

**Proceedings of INAP 2009**

**18<sup>th</sup> International Conference on Applications of  
Declarative Programming and Knowledge  
Management**

Salvador Abreu and Dietmar Seipel (Eds.)

November 5–7, 2009

Évora, Portugal

# Organization

INAP 2009 was organized by the Informatics Department of Universidade de Évora, in cooperation with the Portuguese A. I. Society (APPIA).

## Workshop Chairs

Salvador Abreu	Universidade de Évora, Portugal
Dietmar Seipel	Universität Würzburg, Germany

## Organizing Committee

Vitor Nogueira	University of Évora, Portugal
Vasco Pedro	University of Évora, Portugal
Pedro Salgueiro	University of Évora, Portugal

## Program Committee

Salvador Abreu	University of Évora, Portugal (co-chair)
Sergio Alvarez	Boston College, USA
Philippe Codognet	CNRS/JFLI, Tokyo, Japan
Vitor Santos Costa	University of Porto, Portugal
Daniel Diaz	University of Paris I, France
Ulrich Geske	University of Potsdam, Germany
Gopal Gupta	UT Dallas, USA
Petra Hofstedt	Technical University of Berlin, Germany
Ulrich Neumerkel	Technical University of Vienna, Austria
Vitor Nogueira	University of Évora, Portugal
Enrico Pontelli	New Mexico State University, USA
Irene Rodrigues	University of Évora, Portugal
Carolina Ruiz	Worcester Polytechnic Institute, USA
Dietmar Seipel	University of Wuerzburg, Germany (co-chair)
Terrance Swift	CENTRIA, Portugal
Hans Tompits	Technical University of Vienna, Austria
Masanobu Umeda	Kyushu Institute of Technology, Japan
Armin Wolf	Fraunhofer FIRST, Berlin, Germany
Osamu Yoshie	Waseda University, Japan

# Table of Contents

An Alternative Declarative Approach to Database Interaction and Management ( <i>invited talk</i> ) .....	1
<i>António Porto</i>	
Design Patterns for Tabled Logic Programming ( <i>invited talk</i> ) .....	3
<i>Terrance Swift</i>	
<b>Applications</b> .....	
Elder Care via Intention Recognition and Evolution Prospection .....	5
<i>Luis Moniz Pereira and Anh Han The</i>	
Knowledge Management Strategy and Tactics for Forging Die Design Support .....	21
<i>Masanobu Umeda and Yuji Mure</i>	
Network Monitoring with Constraint Programming: Preliminary Specification and Analysis .....	37
<i>Pedro Salgueiro and Salvador Abreu</i>	
Searching in Protein State Space .....	53
<i>Dietmar Seipel and Jörg Schultz</i>	
The Contact-Center Business Analyzer: a case for Persistent Contextual Logic Programming .....	69
<i>Cláudio Fernandes, Nuno Lopes, Manuel Monteiro and Salvador Abreu</i>	
<b>Databases and Query Languages</b> .....	
An Efficient Logic Programming Approach to Sub-Graph Isomorphic Queries .....	83
<i>Hasan Jamil</i>	
Database NL Dialogues question answering as a Constraint Satisfaction Problem .....	97
<i>Irene Rodrigues, Luis Quintano and Lúcia Silva Ferreira</i>	
Extending XQuery for Semantic Web Reasoning .....	109
<i>Jesus Manuel Almendros-Jimenez</i>	
JSquash: Source Code Analysis of Embedded Database Applications for Determining SQL Statements .....	125
<i>Dietmar Seipel, Andreas Böhm and Markus Fröhlich</i>	

Reference Model and Perspective Schemata Inference for Enterprise Data Integration .....	141
<i>Valéria Magalhães Pequeno and João Moura Pires</i>	
<b>Foundations and Extensions of LP</b> .....	
A Very Compact and Efficient Representation of List Terms for Tabled Logic Programs .....	157
<i>João Raimundo and Ricardo Rocha</i>	
Inspection Points and Meta-Abduction in Logic Programs .....	171
<i>Luis Moniz Pereira and Alexandre Miguel Pinto</i>	
Stable Model implementation of Layer Supported Models by program transformation .....	185
<i>Luis Moniz Pereira and Alexandre Miguel Pinto</i>	
Towards Computing Revised Models for FO Theories .....	199
<i>Johan Wittocx, Broes De Cat and Marc Denecker</i>	
ISTO: a Language for Temporal Organisational Information Systems .....	213
<i>Vitor Nogueira and Salvador Abreu</i>	
Knowledge Representation Using Logtalk Parametric Objects .....	225
<i>Paulo Moura</i>	
Adaptive Reasoning for Cooperative Agents .....	241
<i>Luis Moniz Pereira and Alexandre Miguel Pinto</i>	
Runtime Verification of Agent Properties .....	257
<i>Stefania Costantini, Pierangelo Dell’Acqua, Luis Moniz Pereira and Panagiota Tsintza</i>	

---

# An Alternative Declarative Approach to Database Interaction and Management

António Porto

Faculdade de Ciências  
Universidade do Porto  
`ap@dcc.fc.up.pt`

**Abstract.** Most real-world applications inevitably face the issue of persistence, generally understood as how to design, maintain and interact with a database. The standard approach relies on the mature technology of relational databases, with interaction specified through SQL embedded in the host programming language. Attempts to raise the level of the error-prone interaction code have been in the direction of object-oriented databases or deductive databases, with simpler queries but less mature technology on the database part.

Here we present a different high-level approach, close in spirit to natural language, using variable-free conceptual expressions that are quite concise, natural and easy to read and understand, promoting much better code maintenance than the alternative approaches. This is achieved by using the flexible operator syntax and the deductive capabilities of logic programming in two ways, first to compile the database scheme from a modular structural description into a clausal representation, and then to translate (using the compiled scheme) terms expressing queries and commands into SQL statements.

The approach relies crucially on the use of attributes, whose inheritance and composition avoid many explicit joins. Expressions become natural by choosing the right noun phrases (rather than verbal) for the attributes. A useful feature is the use of global parameters for implicit current values. Our deductive handle on the scheme allows the query translation to automatically split the needed joins into inner and outer joins. The abstraction power is further raised by having manifold attributes, whose values actually vary along a parametric domain, the main examples being the handling of temporal and multi-lingual data.

Commands can also be very high-level, for example the simple statement of update of an identity value (part of the primary key) of an individual results in its replacement in all tuples of all concepts where the individual was referenced, this being done in the correct order to prevent violation of foreign keys.

We present the ideas incrementally, with examples along with a rigorous account of the used syntax and semantics.

---

---

# Design Patterns for Tabled Logic Programming (Abstract)

Terrance Swift

Centro de Inteligência Artificial, Universidade Nova de Lisboa

The past few years have brought an increasing amount of research into Tabled Logic Programming (TLP), so that TLP is now available in several Prolog systems including XSB, YAP, B Prolog, Mercury, ALS, and Ciao. A leading reason for this interest is the demonstrated ability of TLP to solve practical problems. TLP has been used in XSB to implement systems for program analysis, model checking, agent frameworks, semantic web applications, natural language processing, medical informatics, and software engineering. TLP has been extensively used in YAP for machine learning and bioinformatics applications; TLP has also been used B Prolog to implement probabilistic reasoning in the PRISM system. In some of these applications, TLP is used simply as a means to compute transitive closure. In other applications, the uses of TLP are more elaborate: tabling is used to evaluate large sets of mutually recursive predicates, to provide full well-founded negation, to provide a basis for quantitative reasoning, or is combined with logical constraints. A bibliography of such applications is available in [2].

This paper attempts to synthesize approaches to TLP by making use of the organizing principle of software design patterns (cf. [1]). Software design patterns were originally applied to object-oriented programs, but they have since been applied to a variety of areas including workflow systems, stream databases, and enterprise architectures. However, within logic programming the use of design patterns is uncommon, perhaps because of their lack of formality. While software design patterns do not provide a formal basis for program synthesis or development, they can provide a useful framework to highlight programming idioms and to associate these idioms with their uses. Even a partial framework can benefit users by providing a survey of “tricks of the trade”; it can benefit engine implementors by indicating the types of operations that must be made efficient or robust; and it can benefit compiler writers by indicating analysis problems that are characteristic of TLP. Towards that end, this paper makes the first known attempt at using design patterns to classify TLP programming idioms.

## References

1. J. Coplien and D. Schmidt. *Pattern Languages of Program Design*. Addison-Wesley, 1995.
2. T. Swift and D.S. Warren. XSB: Extending the power of prolog using tabling, 2009. available at [www.cs.sunysb.edu/~tswift](http://www.cs.sunysb.edu/~tswift).

---



---

# Elder Care via Intention Recognition and Evolution Prospection

Luís Moniz Pereira and Han The Anh  
lmp@di.fct.unl.pt, h.anh@fct.unl.pt

Centro de Inteligência Artificial (CENTRIA)  
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal

**Abstract.** We explore and exemplify the application in the Elder Care context of the ability to perform Intention Recognition and of wielding Evolution Prospection methods. This is achieved by means of an articulate use of Causal Bayes Nets (for heuristically gauging probable general intentions), combined with specific generation of plans involving preferences (for checking which such intentions are plausibly being carried out in the specific situation at hand). The overall approach is formulated within one coherent and general logic programming framework and implemented system. The paper recaps required background and illustrates the approach via an extended application example.

**Keywords:** Intention Recognition, Elder Care, Causal Bayes Nets, P-Log, Evolution Prospection, Preferences.

## 1 Introduction

In the last twenty years there has been a significant increase of the average age of the population in most western countries and the number of elderly people has been and will be constantly growing. For this reason there has been a strong development of supportive technology for elderly people living independently in their own homes, for example, RoboCare Project [8] – an ongoing project developing robots for assisted elderly people’s living, SINDI – a logic-based home monitoring system [9].

For the Elder Care application domain, in order to provide contextually appropriate help for elders, it is required that the assisting system have the ability to observe the actions of the elders, recognize their intentions, and then provide suggestions on how to achieve the recognized intentions on the basis of the conceived plans. In this paper we focus on the latter two steps in order to design and implement an elder care logic programming based assisting system. The first step of perceiving elders’ actions is taken for granted. For elders’ intention recognition based on their observable actions, we employ our work on Intention Recognition system using Causal Bayes Networks and plan generation techniques, described in [1]. The intention recognition component is indispensable for living-alone elders, in order to provide them with timely suggestions. The next step, that of providing action suggestions for realizing the recognized intention gleaned from the previous stage, is implemented using our Evolution Prospection Agent (EPA) system [2,3]. The latter can prospectively look ahead into the future to choose the best course of evolution whose actions achieve the recognized intention, while being aware

---

of the external environment and of an elder's preferences and already scheduled future events. Expectation rules and a priori preferences take into account the physical state (health reports) information of the elder to guarantee that only contextually safe healthy choices are generated; then, information such as the elder's pleasure, interests, etc. are taken into account by a posteriori preferences and the like. The advance and easiness of expressing preferences in EPA [3] enable to closely take into account the elders' preferences, which we believe, would increase the degree of acceptance of the elders w.r.t. the technological help - an important issue of the domain [10].

Recently, there have been many works addressing the problem of intention recognition as well as its applications in a variety of fields. As in Heinze's doctoral thesis [11], intention recognition is defined, in general terms, as the process of becoming aware of the intention of another agent and, more technically, as the problem of inferring an agent's intention through its actions and their effects on the environment. According to this definition, one approach to tackle intention recognition is by reducing it to plan recognition, i.e. the problem of generating plans achieving the intentions and choosing the ones that match the observed actions and their effects in the environment of the intending agent. This has been the main stream so far [11,14].

One of the main issues of that approach is that of finding an initial set of possible intentions (of the intending agent) that the plan generator is going to tackle, and which must be imagined by the recognizing agent. Undoubtedly, this set should depend on the situation at hand, since generating plans for all intentions one agent could have, for whatever situation he might be in, is unrealistic if not impossible.

In this paper, we use an approach to solve this problem employing so-called *situation-sensitive Causal Bayes Networks* (CBN) - That is, CBNs [18] that change according to the situation under consideration, itself subject to ongoing change as a result of actions. Therefore, in some given situation, a CBN can be configured dynamically, to compute the likelihood of intentions and filter out the much less likely intentions. The plan generator (or plan library) thus only needs, at the start, to deal with the remaining more relevant because more probable or credible intentions, rather than all conceivable intentions. One of the important advantages of our approach is that, on the basis of the information provided by the CBN the recognizing agent can see which intentions are more likely and worth addressing, so, in case of having to make a quick decision, it can focus on the most relevant ones first. CBNs, in our work, are represented in P-log [4,6,5], a declarative language that combines logical and probabilistic reasoning, and uses Answer Set Programming (ASP) as its logical and CBNs as its probabilistic foundations. Given a CBN, its situation-sensitive version is constructed by attaching to it a logical component to dynamically compute situation specific probabilistic information, which is forthwith inserted into the P-log program representing that CBN. The computation is dynamic in the sense that there is a process of inter-feedback between the logical component and the CBN, i.e. the result from the updated CBN is also given back to the logical component, and that might give rise to further updating, etc.

In addition, one more advantage of our approach, in comparison with those using solely BNs [12,13] is that these just use the available information for constructing CBNs. For complicated tasks, e.g. in recognizing hidden intentions, not all information

---

is observable. The approach of combining with plan generation provides a way to guide the recognition process: which actions (or their effects) should be checked whether they were (hiddenly) executed by the intending agent. In practice, one can make use of any plan generators or plan libraries available. For integration's sake, we can use the ASP based conditional planner called ASCP [16] from XSB Prolog using the XASP package [7,24] for interfacing with Smodels [22] – an answer set solver – or, alternatively, rely on plan libraries so obtained.

In the sequel we briefly describe the intention recognition and evolution propection systems, but not the planner. Then we show in detail how to combine them to provide contextual help for elderly people, illustrating with an extended example.

## 2 Intention Recognition

### 2.1 Causal Bayes Networks

A Bayes Network (BN), recapitulated here for convenience in order to help see their realization in P-log, is a pair consisting of a directed acyclic graph (dag) whose nodes represent variables and missing edges encode conditional independencies between the variables, and an associated probability distribution satisfying the assumption of conditional independence (Causal Markov Assumption - CMA), saying that variables are independent of their non-effects conditional on their direct causes [18].

If there is an edge from node  $A$  to another node  $B$ ,  $A$  is called a parent of  $B$ , and  $B$  is a child of  $A$ . The set of parent nodes of a node  $A$  is denoted by  $parents(A)$ . Ancestor nodes of  $A$  are parents of  $A$  or parents of some ancestor nodes of  $A$ . If  $A$  has no parents ( $parents(A) = \emptyset$ ), it is called a top node. If  $A$  has no child, it is called a bottom node. The nodes which are neither top nor bottom are said intermediate. If the value of a node is observed, the node is said to be an *evidence node*. In a BN, associated with each intermediate node of its dag is a specification of the distribution of its variable, say  $A$ , conditioned on its parents in the graph, i.e.  $P(A|parents(A))$  is specified. For a top node, one without parents, the unconditional distribution of the variable is specified. These distributions are called Conditional Probability Distribution (CPD) of the BN.

Suppose the nodes of the dag form a causally sufficient set [17], i.e. no common causes of any two nodes are omitted, then implied by CMA [17], the joint distribution of all node values of the set can be determined as the product of conditional probabilities of the value of each node on its parents

$$P(X_1, \dots, X_N) = \prod_{i=1}^N P(X_i | parents(X_i))$$

where  $V = \{X_i | 1 \leq i \leq N\}$  is the set of nodes of the dag.

Suppose there is a set of evidence nodes in the dag, say  $O = \{O_1, \dots, O_m\} \subset V$ . We can determine the conditional probability of a variable  $X$  given the observed value of evidence nodes by using the conditional probability formula

$$P(X|O) = \frac{P(X, O)}{P(O)} = \frac{P(X, O_1, \dots, O_m)}{P(O_1, \dots, O_m)} \quad (1)$$

---

where the numerator and denominator are computed by summing the joint probabilities over all absent variables with respect to  $V$ , as follows

$$P(X = x, O = o) = \sum_{av \in ASG(AV_1)} P(X = x, O = o, AV_1 = av)$$

$$P(O = o) = \sum_{av \in ASG(AV_2)} P(O = o, AV_2 = av)$$

where  $o = \{o_1, \dots, o_m\}$  with  $o_1, \dots, o_m$  being the observed values of  $O_1, \dots, O_m$ , respectively;  $ASG(Vt)$  denotes the set of all assignments of vector  $Vt$  (with components are variables in  $V$ );  $AV_1, AV_2$  are vectors components of which are corresponding absent variables, i.e. variables in  $V \setminus \{O \cup \{X\}\}$  and  $V \setminus O$ , respectively.

In short, to define a BN specify the structure of the network, its Conditional Probability Distribution (CPD) and the prior probability distribution of the top nodes.

## 2.2 Intention recognition with Causal Bayesian Networks

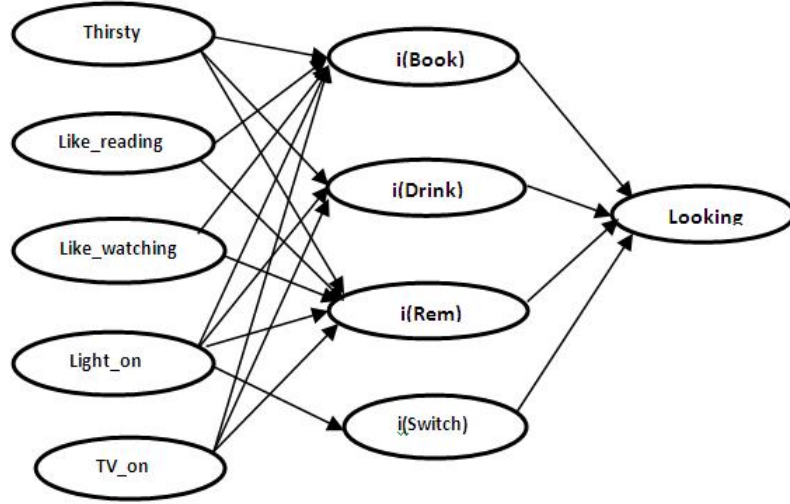
The first phase of the intention recognition system is to find out how likely each possible intention is, based on current observations such as observed actions of the intending agent or the effects its actions (either actually observed, or missed direct observation) have in the environment. It is carried out by using a CBN with nodes standing for binary random variables that represent causes, intentions, actions and effects.

Intentions are represented by intermediate nodes whose ancestor nodes stand for causes that give rise to intentions. Intuitively, we extend Heinze's tri-level model [11] with a so-called pre-intentional level that describes the causes of intentions, used to estimate prior probabilities of the intentions. This additional level guarantees the causal sufficiency condition of the set of nodes of the dag. However, if these prior probabilities can be specified without considering the causes, intentions are represented by top nodes. These reflect the problem context or the intending agent's mental state.

Observed actions are represented as children of the intentions that causally affect them. Observable effects are represented as bottom nodes. They can be children of observed action nodes, of intention nodes, or of some unobserved actions that might cause the observable effects that are added as children of the intention nodes.

The causal relations among nodes of the BNs (e.g. which causes give rise to an intention, which intentions trigger an action, which actions have an effect), as well as their CPD and the distribution of the top nodes, are specified by domain experts. However, they might be learnt mechanically. By using formula 1 the conditional probabilities of each intention on current observations can be determined,  $X$  being an intention and  $O$  being the set of current observations.

*Example 1 (Elder Care).* An elder stays alone in his apartment. The intention recognition system observes that he is looking for something in the living room. In order to assist him, the system needs to figure out what he intends to find. The possible things are: something to read (*book*); something to drink (*drink*); the TV remote control (*Rem*); and the light switch (*Switch*). The CBN representing this scenario is that of Figure 1. Its CPD and the distribution of top nodes will be given directly in P-log code.



**Fig. 1:** Elder's intentions CBN

### 2.3 P-log

The computation in CBNs is automated using P-log, a declarative language that combines logical and probabilistic reasoning, and ASP as its logical and CBNs as its probabilistic foundations. We recap it here for self-containment, to the extent we use it.

The original P-log [4,6] uses ASP as a tool for computing all stable models of the logical part of P-log. Although ASP has been proved to be a useful paradigm for solving a variety of combinatorial problems, its non-relevance property [7] makes the P-log system sometimes computationally redundant. Newer developments of P-log [5] use the XASP package of XSB Prolog [24] for interfacing with Smodels [22] – an answer set solver. The power of ASP allows the representation of both classical and default negation in P-log easily. Moreover, the new P-log uses XSB as the underlying processing platform, allowing arbitrary Prolog code for recursive definitions. Consequently, it allows more expressive queries not supported in the original version, such as meta queries (probabilistic built-in predicates can be used as usual XSB predicates, thus allowing full power of probabilistic reasoning in XSB) and queries in the form of any XSB predicate expression [5]. Moreover, the tabling mechanism of XSB [23] significantly improves the performance of the system.

In general, a P-log program  $\Pi$  consists of the 5 components detailed below: a sorted signature, declarations, a regular part, a set of random selection rules, a probabilistic information part, and a set of observations and actions.

**(i) Sorted signature and Declaration** The sorted signature  $\Sigma$  of  $\Pi$  contains a set of constant symbols and term-building function symbols, which are used to form terms in the usual way. Additionally, the signature contains a collection of special reserved function symbols called attributes. Attribute terms are expressions of the form  $a(\bar{t})$ , where  $a$  is an attribute and  $\bar{t}$  is a vector of terms of the sorts required by  $a$ . A literal is an atomic statement,  $p$ , or its explicit negation,  $neg\text{-}p$ .

---

The declaration part of a P-log program can be defined as a collection of sorts and sort declarations of attributes. A sort  $c$  can be defined by listing all the elements  $c = \{x_1, \dots, x_n\}$ , specifying the range of values  $c = \{L..U\}$  where  $L$  and  $U$  are the integer lower bound and upper bound of the sort  $c$ . Attribute  $a$  with domain  $c_1 \times \dots \times c_n$  and range  $c_0$  is represented as follows:

$$a : c_1 \times \dots \times c_n \dashrightarrow c_0$$

If attribute  $a$  has no domain parameter, we simply write  $a : c_0$ . The range of attribute  $a$  is denoted by  $range(a)$ .

**(ii) Regular part** This part of a P-log program consists of a collection of rules, facts, and integrity constraints (IC) in the form of denials, formed using literals of  $\Sigma$ . An IC is encoded as a rule with the `false` literal in the head.

**(iii) Random Selection Rule** This is a rule for attribute  $a$  having the form:

$$random(RandomName, a(\bar{t}), DynamicRange) :- Body$$

This means that the attribute instance  $a(\bar{t})$  is random if the conditions in *Body* are satisfied. The *DynamicRange*, not used in the particular examples in the sequel, allows to restrict the default range for random attributes. The *RandomName* is a syntactic mechanism used to link random attributes to the corresponding probabilities. If there is no precondition, we simply put *true* in the body. A constant *full* can be used in *DynamicRange* to signal that the dynamic domain is equal to  $range(a)$ .

**(iv) Probabilistic Information** Information about probabilities of random attribute instances  $a(\bar{t})$  taking a particular value  $y$  is given by probability atoms (or simply pa-atoms) which have the following form:

$$pa(RandomName, a(\bar{t}, y), d_-(A, B)) :- Body.$$

meaning that if the *Body* were true, and the value of  $a(\bar{t})$  were selected by a rule named *RandomName*, then *Body* would cause  $a(\bar{t}) = y$  with probability  $\frac{A}{B}$ .

**(v) Observations and Actions** These are, respectively, statements of the forms *obs(l)* and *do(l)*, where  $l$  is a literal. Observations are used to record the outcomes of random events, i.e. of random attributes and attributes dependent on them. The statement *do(a(t, y))* indicates that  $a(t) = y$  is enforced true as the result of a deliberate action, not an observation.

## 2.4 An example: recognizing elders' intentions

To begin with, we need to declare two sorts:

```
bool = {t, f}.
elder_intentions = {book, drink, rem, switch}.
```

where the second one is the sort of possible intentions of the elder. There are five top nodes, named *thirsty(thsty)*, *like\_reading(lr)*, *like\_watching(lw)*, *tv\_on(tv)*, *light\_on(light)*, belonging to the pre-intention level to describe the causes that might give rise to the considered intentions. The values of last two nodes are observed (evidence nodes). The corresponding random attributes are declared as

---

```

thsty:bool. lr:bool. lw:bool. tv:bool. light:bool.
random(rth,thsty,full). random(rlr,lr,full).
random(rlw,lw,full). random(rtv,tv,full).
random(rl, light, full).

```

and their independent probability distributions are encoded with pa-rules as

```

pa(rth,thsty(t),d_(1,2)). pa(rlr,lr(t),d_(8,10)).
pa(rlw,lw(t),d_(7,10)). pa(rtv,tv(t),d_(1,2)).
pa(rl,light(t),d_(1,2)).

```

The possible intentions reading is afforded by four nodes, representing the four possible intentions of the elder, as mentioned above. The corresponding random attributes are coded specifying an attribute with domain *elder\_intentions* and receives boolean values

```

i:elder_intentions --> bool. random(ri, i(I), full).

```

The probability distribution of each intention node conditional on the causes are coded in P-log below. Firstly, for *i(book)*:

```

pa(ri(book),i(book,t),d_(0,1)):-light(f).
pa(ri(book),i(book,t),d_(0,1)):-light(t),tv(t).
pa(ri(book),i(book,t),d_(6,10)):-light(t),tv(f),lr(t),lw(t),thsty(t).
pa(ri(book),i(book,t),d_(65,100)):-light(t),tv(f),lr(t),lw(t),thsty(f).
pa(ri(book),i(book,t),d_(7,10)):-light(t),tv(f),lr(t),lw(f),thsty(t).
pa(ri(book),i(book,t),d_(8,10)):-light(t),tv(f),lr(t),lw(f),thsty(f).
pa(ri(book),i(book,t),d_(1,10)):-light(t),tv(f),lr(f),lw(t).
pa(ri(book),i(book,t),d_(4,10)):-light(t),tv(f),lr(f),lw(f).

```

For *i(drink)*:

```

pa(ri(drink),i(drink,t),d_(0,1)) :- light(f).
pa(ri(drink),i(drink,t),d_(9,10)) :- light(t), thsty(t).
pa(ri(drink),i(drink,t),d_(1,10)) :- light(t), thsty(f).

```

For *i(rem)*:

```

pa(ri(rem),i(rem,t),d_(0,1)):-light(f).
pa(ri(rem),i(rem,t),d_(8,10)):-light(t),tv(t).
pa(ri(rem),i(rem,t),d_(4,10)):-light(t),tv(f),lw(t),lr(t),thsty(t).
pa(ri(rem),i(rem,t),d_(5,10)):-light(t),tv(f),lw(t),lr(t),thsty(f).
pa(ri(rem),i(rem,t),d_(6,10)):-light(t),tv(f),lw(t),lr(f),thsty(t).
pa(ri(rem),i(rem,t),d_(9,10)):-light(t),tv(f),lw(t),lr(f),thsty(f).
pa(ri(rem),i(rem,t),d_(1,10)):-light(t),tv(f),lw(f),lr(t),thsty(t).
pa(ri(rem),i(rem,t),d_(2,10)):-light(t),tv(f),lw(f),lr(t),thsty(f).
pa(ri(rem),i(rem,t),d_(0,1)):-light(t),tv(f),lw(f),lr(f),thsty(t).
pa(ri(rem),i(rem,t),d_(3,10)):-light(t),tv(f),lw(f),lr(f),thsty(f).

```

For *i(switch)*:

```

pa(ri(switch),i(switch,t),d_(1,1)) :- light(f).
pa(ri(switch),i(switch,t),d_(1,100)) :- light(t).

```

---

There is only one observation, namely, that the elder is looking for something (*look*). The declaration of the corresponding random attribute and its probability distribution conditional on the possible intentions are given as follows:

```
look : bool. random(rla, look, full).
pa(rla, look(t), d_(99,100)) :- i(book,t), i(drink,t), i(rem,t).
pa(rla, look(t), d_(7,10)) :- i(book,t), i(drink,t), i(rem,f).
pa(rla, look(t), d_(9,10)) :- i(book,t), i(drink,f), i(rem,t).
pa(rla, look(t), d_(6,10)) :- i(book,t), i(drink,f), i(rem,f).
pa(rla, look(t), d_(6,10)) :- i(book,f), i(drink,t), i(rem,t).
pa(rla, look(t), d_(3,10)) :- i(book,f), i(drink,t), i(rem,f).
pa(rla, look(t), d_(4,10)) :- i(book,f), i(drink,f), i(rem,t).
pa(rla, look(t), d_(1,10)) :- i(book,f), i(drink,f), i(rem,f), i(switch,t).
pa(rla, look(t), d_(1,100)) :- i(book,f), i(drink,f), i(rem,f), i(switch,f).
```

Recall that the two nodes *tv\_on* and *light\_on* are observed. The probabilities that the elder has the intention of looking for *book*, *drink*, *remote control* and *light switch* given the observations that he is looking around and of the states of the light (on or off) and TV (on or off) can be found in P-log with the following queries, respectively:

$$\begin{aligned} ? - pr(i(book, t) \mid (obs(tv(S_1)) \& light(S_2) \& obs(look(t))), V_1). \\ ? - pr(i(drink, t) \mid (obs(tv(S_1)) \& light(S_2) \& obs(look(t))), V_2). \\ ? - pr(i(rem, t) \mid (obs(tv(S_1)) \& light(S_2) \& obs(look(t))), V_3). \\ ? - pr(i(switch, t) \mid (obs(tv(S_1)) \& light(S_2) \& obs(look(t))), V_4). \end{aligned}$$

where  $S_1, S_2$  are boolean values ( $t$  or  $f$ ) instantiated during execution, depending on the states of the light and TV. Let us consider the possible cases

- If the light is off ( $S_2 = f$ ), then  $V_1 = V_2 = V_3 = 0, V_4 = 1.0$ , regardless of the state of the TV.
- If the light is on and TV is off ( $S_1 = t, S_2 = f$ ), then  $V_1 = 0.7521, V_2 = 0.5465, V_3 = 0.5036, V_4 = 0.0101$ .
- If both light and TV are on ( $S_1 = t, S_2 = t$ ), then  $V_1 = 0, V_2 = 0.6263, V_3 = 0.9279, V_4 = 0.0102$ .

Thus, if one observes that the light is off, definitely the elder is looking for the light switch, given that he is looking around. Otherwise, if one observes the light is on, in both cases where the TV is either on or off, the first three intentions *book*, *drink*, *remote control* still need to be put under consideration in the next phase, generating possible plans for each of them. The intention of looking for the light switch is very unlikely to be the case comparing with other three, thus being ruled out. When there is light one goes directly to the light switch if the intention is to turn it off, without having to look for it.

**2.4.1 Situation-sensitive CBNs** Undoubtedly, CBNs should be situation-sensitive since using a general CBN for all specific situations (instances) of a problem domain is unrealistic and most likely imprecise. However, consulting the domain expert to manually change the CBN w.r.t. each situation is also very costly. We here provide a way



---

to construct situation-sensitive CBNs, i.e. ones that change according to the given situation. It uses Logic Programming (LP) techniques to compute situation specific probabilistic information which is then introduced into a CBN which is general for the problem domain.

The LP techniques can be deduction with top-down procedure (Prolog) (to deduce situation-specific probabilistic information) or abduction (to abduce probabilistic information needed to explain observations representing the given situation). However, we do not exclude various other types of reasoning, e.g. including integrity constraint satisfaction, contradiction removal, preferences, or inductive learning, whose results can be compiled (in part) into an evolving CBN.

The general issue of how to update a CBN with new probabilistic information can take advantage of the advances in LP semantics for evolving programs by means of rule updates [19,20,21]. In this paper, however, we don't need such general updates, and make do with a simpler way, shown in the sequel.

*Example 2 (Elder Care, cont'd).* In this scenario, the CBN may vary depending on some observed factors, for example, the time of day, the current temperature, etc. We design a logical component for the CBN to deal with those factors:

```
pa_rule(pa(rlk,lr(t),d_(0,1)),[]):-time(T), T>0, T<5, !.
pa_rule(pa(rlk,lr(t),d_(1,10)),[]):-time(T), T>=5, T<8, !.
pa_rule(pa(rlw,lw(t),d_(9,10)),[]):-time(T), schedule(T,football), !.
pa_rule(pa(rlw,lw(t),d_(1,10)),[]):-time(T), (T>23; T<5), !.
pa_rule(pa(rth,thsty(t),d_(7,10)),[]):-temp(T), T>30, !.
pa_rule(pa(rlk,lr(t),d_(1,10)),[]):-temp(TM), TM >30, !.
pa_rule(pa(rlw,lw(t),d_(3,10)),[]):-temp(TM), TM>30, !.
```

Given P-log probabilistic information by *pa/3* rules, then the corresponding so-called situation-sensitive *pa\_rule/2* predicate takes the head and body of some *pa/3* rule as its first and second arguments, respectively, and includes conditions for its activation in its own body. Now, a situation is given by asserted facts representing it and, in order to find the probabilistic information specific to the given situation, we simply use the XSB Prolog built-in *findall/3* predicate to find all true *pa/3* literals expressed by the *pa\_rule/2* rules with true bodies in the situation. For example, when the time and temperature are defined (the assisting system should be aware of such information), they are asserted using predicates *time/1* and *temp/1*. Note that in this modelling, to guarantee the consistency of the P-log program (there must not be two *pa*-rules for the same attribute instance with non-exclusive bodies) we consider time with a higher priority than temperature, enacted by using XSB Prolog *cut* operator, as can be seen in the *rlk* and *rlw* cases.

## 2.5 Plan Generation

The second phase of the intention recognition system is to generate conceivable plans that can achieve the most likely intentions surviving after the first phase. The plan generation phase can be carried out by ASCP – an ASP logic programming based conditional planner [16], as in our work [1], which details the method and the language for expressing actions. For lack of space, we will simply assume here that the system has been

---

furnished a plan library that provides recipes for achieving the recognized intentions in the situation. This alternative method has also been employed in several works of plan recognition, e.g. [11,14].

### 3 Elder Assisting with Evolution Prospection

#### 3.1 Preliminary

We next describe constructs of the evolution prospection system that are necessary for representation of the example. A full presentation can be found in [2]. The separate formalism for expressing actions can be found in [1] or [16].

**3.1.1 Language** Let  $\mathcal{L}$  be a first order language. A domain literal in  $\mathcal{L}$  is a domain atom  $A$  or its default negation  $not\ A$ . The latter is used to express that the atom is false by default (Closed World Assumption). A domain rule in  $\mathcal{L}$  is a rule of the form:

$$A \leftarrow L_1, \dots, L_t \quad (t \geq 0)$$

where  $A$  is a domain atom and  $L_1, \dots, L_t$  are domain literals. An integrity constraint in  $\mathcal{L}$  is a rule with an empty head. A (logic) program  $P$  over  $\mathcal{L}$  is a set of domain rules and integrity constraints, standing for all their ground instances.

**3.1.2 Active Goals** In each cycle of its evolution the agent has a set of active goals or desires. We introduce the *on\_observe/1* predicate, which we consider as representing active goals or desires that, once triggered by the observations figuring in its rule bodies, cause the agent to attempt their satisfaction by launching all the queries standing for them, or using preferences to select them. The rule for an active goal AG is of the form:

$$on\_observe(AG) \leftarrow L_1, \dots, L_t \quad (t \geq 0)$$

where  $L_1, \dots, L_t$  are domain literals. During evolution, an active goal may be triggered by some events, previous commitments or some history-related information. When starting a cycle, the agent collects its active goals by finding all the *on\_observe(AG)* that hold under the initial theory without performing any abduction, then finds abductive solutions for their conjunction.

**3.1.3 Preferring abducibles** Every program  $P$  is associated with a set of abducibles  $\mathcal{A} \subseteq \mathcal{L}$ . These, and their default negations, can be seen as hypotheses that provide hypothetical solutions or possible explanations to given queries. Abducibles can figure only in the body of program rules. An abducible  $A$  can be assumed only if it is a considered one, i.e. if it is expected in the given situation, and, moreover, there is no expectation to the contrary

$$consider(A) \leftarrow expect(A), not\ expect\_not(A), A$$

The rules about expectations are domain-specific knowledge contained in the theory of the program, and effectively constrain the hypotheses available in a situation. To

---

express preference criteria among abducibles, we envisage an extended language  $\mathcal{L}^*$ . A preference atom in  $\mathcal{L}^*$  is of the form  $a \triangleleft b$ , where  $a$  and  $b$  are abducibles. It means that if  $b$  can be assumed (i.e. considered), then  $a \triangleleft b$  forces  $a$  to be assumed too if it can. A preference rule in  $\mathcal{L}^*$  is of the form:

$$a \triangleleft b \leftarrow L_1, \dots, L_t \ (t \geq 0)$$

where  $L_1, \dots, L_t$  are domain literals over  $\mathcal{L}^*$ . This preference rule can be coded as follows:

$$expect\_not(b) \leftarrow L_1, \dots, L_n, not\ expect\_not(a), expect(a), not\ a$$

In fact, if  $b$  is considered, the consider-rule for abducible  $b$  requires  $expect\_not(b)$  to be false, i.e. every rule with the head  $expect\_not(b)$  cannot have a true body. Thus,  $a \triangleleft b$ , that is if its body in the preference rule holds, and if  $a$  is expected, and not counter-expected, then  $a$  must be abducted so that this particular rule for  $expect\_not(b)$  also fails, and the abduction of  $b$  may go through if all the other rules for  $expect\_not(b)$  fail as well.

*A priori* preferences are used to produce the most interesting or relevant conjectures about possible future states. They are taken into account when generating possible scenarios (abductive solutions), which will subsequently be preferred amongst each other *a posteriori*.

**3.1.4 A posteriori Preferences** Having computed possible scenarios, represented by abductive solutions, more favorable scenarios can be preferred *a posteriori*. Typically, *a posteriori* preferences are performed by evaluating consequences of abducibles in abductive solutions. An *a posteriori* preference has the form:

$$A_i \ll A_j \leftarrow holds\_given(L_i, A_i), holds\_given(L_j, A_j)$$

where  $A_i, A_j$  are abductive solutions and  $L_i, L_j$  are domain literals. This means that  $A_i$  is preferred to  $A_j$  *a posteriori* if  $L_i$  and  $L_j$  are true as the side-effects of abductive solutions  $A_i$  and  $A_j$ , respectively, without any further abduction when testing for the side-effects. Optionally, in the body of the preference rule there can be any Prolog predicate used to quantitatively compare the consequences of the two abductive solutions.

**3.1.5 Evolution result a posteriori preference** While looking ahead a number of steps into the future, the agent is confronted with the problem of having several different possible courses of evolution. It needs to be able to prefer amongst them to determine the best courses from its present state (and any state in general). The *a posteriori* preferences are no longer appropriate, since they can be used to evaluate only one-step-far consequences of a commitment. The agent should be able to also declaratively specify preference amongst evolutions through quantitatively or qualitatively evaluating the consequences or side-effects of each evolution choice.

*A posteriori* preference is generalized to prefer between two evolutions. An *evolution result a posteriori* preference is performed by evaluating consequences of following some evolutions. The agent must use the imagination (look-ahead capability) and

---

present knowledge to evaluate the consequences of evolving according to a particular course of evolution. An *evolution result a posteriori preference* rule has the form:

$$E_i \lll E_j \leftarrow \text{holds\_in\_evol}(L_i, E_i), \text{holds\_in\_evol}(L_j, E_j)$$

where  $E_i, E_j$  are possible evolutions and  $L_i, L_j$  are domain literals. This preference implies that  $E_i$  is preferred to  $E_j$  if  $L_i$  and  $L_j$  are true as evolution history side-effects when evolving according to  $E_i$  or  $E_j$ , respectively, without making further abductions when just checking for the side-effects. Optionally, in the body of the preference rule there can be recourse to any Prolog predicate, used to quantitatively compare the consequences of the two evolutions for decision making.

### 3.2 Evolution Prospecction as An Intention Consumer

Having recognized the intention of another agent, EPA system can be used to provide the best courses of evolution for that agent to achieve its own intention. These courses of evolution might be provided to the other agent as suggestions.

In Elder Care domain, assisting systems should be able to provide contextually appropriate suggestions for the elders based on their recognized intentions. The assisting system is supposed to be better aware of the environment, the elders' physical states, mental states as well as their scheduled events, so that it can provide good and safe suggestions, or simply warnings. We continue with the Elder Care example from a previous section for illustration.

*Example 3 (Elder Care, cont'd).* Suppose in Example 1, the final confirmed intention is that of looking for a drink. The possibilities are natural pure water, tea, coffee and juice. EPA now is used to help the elder in choosing an appropriate one. The scenario is coded with the program in Figure 2 below.

The elder's physical states are employed in *a priori* preferences and expectation rules to guarantee that only choices that are contextually safe for the elder are generated. Only after that other aspects, for example the elder's pleasure w.r.t. to each kind of drink, are taken into account, in *a posteriori* preferences.

The information regarding the environment (current time, current temperature) and the physical states of the elder is coded in the Prolog part of the program (lines 9-11). The assisting system is supposed to be aware of this information in order to provide good suggestions.

Line 1 is the declaration of program abducibles: *water, coffee, tea*, and *juice*. All of them are always expected (line 2). Line 3 picks up a recognized intention verified by the planner. The counter-expectation rules in line 4 state that *coffee* is not expected if the elder has high blood pressure, experiences difficulty to sleep or it is late; and *juice* is not expected if it is late. Note that the reserved predicate *prolog/1* is used to allow embedding prolog code in an EPA program. More details can be found in [2,3]. The integrity constraints in line 5 say that is is not allowed to have at the same time the following pairs of drink: tea and coffee, tea and juice, coffee and juice, and tea and water. However, it is the case that the elder can have coffee or juice together with water at the same time.

---

```

1. abds([water/0, coffee/0, tea/0, juice/0]).
2. expect(coffee). expect(tea). expect(water). expect(juice).
3. on_observe(drink) <- has_intention(elder, drink).
   drink <- tea. drink <- coffee. drink <- water. drink <- juice.
4. expect_not(coffee) <- prolog(blood_high_pressure).
   expect_not(coffee) <- prolog(sleep_difficulty).
   expect_not(coffee) <- prolog(late).
   expect_not(juice) <- prolog(late).
5. <- tea, coffee. <- coffee, juice.
   <- tea, juice. <- tea, water.
6. coffee '<|' tea <- prolog(morning_time).
   coffee '<|' water <- prolog(morning_time).
   coffee '<|' juice <- prolog(morning_time).
7. juice '<|' coffee <- prolog(hot). juice '<|' tea <- prolog(hot).
   juice '<|' water <- prolog(hot). water '<|' coffee <- prolog(hot).
   water '<|' tea <- prolog(hot).
8. tea '<|' coffee <- prolog(cold). tea '<|' juice <- prolog(cold).
   tea '<|' water <- prolog(cold).
9. pleasure_level(3) <- coffee. pleasure_level(2) <- tea.
   pleasure_level(1) <- juice. pleasure_level(0) <- water.
10. sugar_level(1) <- coffee. sugar_level(1) <- tea.
    sugar_level(5) <- juice. sugar_level(0) <- water.
11. caffein_level(5) <- coffee. caffein_level(0) <- tea.
    caffein_level(0) <- juice. caffein_level(0) <- water.
12. Ai << Aj <- holds_given(pleasure_level(V1), Ai),
    holds_given(pleasure_level(V2), Aj), V1 > V2.

13. on_observe(health_check) <- time_for_health_check.
    health_check <- precise_result.
    health_check <- imprecise_result.
14. expect(precise_result) <- no_high_sugar, no_high_caffein.
    expect(imprecise_result).
    no_high_sugar <- sugar_level(L), prolog(L < 2).
    no_high_caffein <- caffein_level(L), prolog(L < 2).
15. Ei <<< Ej <- holds_in_evol(precise_result, Ei),
    holds_in_evol(imprecise_result, Ej).

beginProlog.
:- assert(scheduled_events(1, [has_intention(elder, drink)])),
   assert(scheduled_events(2, [time_for_health_check])).
late :- time(T), (T > 23; T < 5).
morning_time :- time(T), T > 7, T < 10.
hot :- temperature(TM), TM > 32.
cold :- temperature(TM), TM < 10.
blood_high_pressure :- physical_state(blood_high_pressure).
sleep_difficulty :- physical_state(sleep_difficulty).
endProlog.

```

**Fig. 2:** Elder Care: Suggestion for a Drink

---

The *a priori* preferences in line 6 say in the morning coffee is preferred to tea, water and juice. And if it is hot, juice is preferred to all other kinds of drink and water is preferred to tea and coffee (line 7). In addition, the *a priori* preferences in line 8 state if the weather is cold, tea is the most favorable, i.e. preferred to all other kinds of drink.

Now let us look at the suggestions provided by the Elder Care assisting system modelled by this EPA program, considering some cases:

1. time(24) (*late*); temperature(16) (*not hot, not cold*); no high blood pressure; no sleep difficulty: there are two *a priori* abductive solutions: *[tea]*, *[water]*. Final solutions: *[tea]* (since it has greater level of pleasure than water, which is ruled out by the *a posteriori* preference in line 12).
2. time(8) (*morning time*); temperature(16) (*not hot, not cold*); no high blood pressure; no sleep difficulty: there are two abductive solutions: *[coffee]*, *[coffee, water]*. Final: *[coffee]*, *[coffee, water]*.
3. time(18) (*not late, not morning time*); temperature(16) (*not cold, not hot*); no high blood pressure; no sleep difficulty: there are six abductive solutions: *[coffee]*, *[coffee, water]*, *[juice]*, *[juice, water]*, *[tea]*, and *[water]*. Final: *[coffee]*, *[coffee, water]*.
4. time(18) (*not late, not morning time*); temperature(16) (*not cold, not hot*); high blood pressure; no sleep difficulty: there are four abductive solutions: *[juice]*, *[juice, water]*, *[tea]*, and *[water]*. Final: *[tea]*.
5. time(18) (*not late, not morning time*); temperature(16) (*not cold, not hot*); no high blood pressure; sleep difficulty: there are four abductive solutions: *[juice]*, *[juice, water]*, *[tea]*, and *[water]*. Final: *[tea]*.
6. time(18) (*not late, not morning time*); temperature(8) (*cold*); no high blood pressure; no sleep difficulty: there is only one abductive solution: *[tea]*.
7. time(18) (*not late, not morning time*); temperature(35) (*hot*); no high blood pressure; no sleep difficulty: there are two abductive solutions: *[juice]*, *[juice, water]*. Final: *[juice]*, *[juice, water]*.

If the *evolution result a posteriori preference* in line 15 is taken into account and the elder is scheduled to go to the hospital for health check in the second day: the first and the second cases do not change. In the third case: the suggestions are *[tea]* and *[water]* since the ones that have *coffee* or *juice* would cause high caffeine and sugar levels, respectively, which can make the checking result (health) imprecise (lines 13-15). Similarly for other cases ...

Note future events can be asserted as Prolog code using the reserved predicate *schedule\_events/2*. For more details of its use see [2,3].

As one can gather, the suggestions provided by this assisting system are quite contextually appropriate. We might elaborate current factors (time, temperature, physical states) and even consider more factors to provide more appropriate suggestions if the situation gets more complicated.

## 4 Conclusions and Future Work

We have shown a coherent LP-based system for assisting elderly people based on an intention recognizer and Evolution Prospection system. The recognizer is to figure out

---

intentions of the elders based on their observed actions or the effects their actions have in the environment, via a combination of situation-sensitive Causal Bayes Nets and a planner. The implemented Evolution Prospection system, being aware of the external environment, elders' preferences and their note future events, is then employed to provide contextually appropriate suggestions that achieve the recognized intention. The system built-in expectation rules and *a priori* preferences take into account the physical state (health reports) information of the elder to guarantee that only contextually safe healthy choices are generated; then, information such as the elder's pleasure, interests, scheduled events, etc. are taken into account by *a posteriori* and *evolution result a posteriori* preferences.

We believe to have shown the usefulness and advantage of our approach of combining several needed features to tackle the application domain, by virtue of an integrated logic programming approach.

One future direction is to implement meta-explanation about evolution prosppection. It would be quite useful in the considered setting, as the elder care assisting system should be able to explain to elders the whys and wherefores of suggestions made.

Moreover, it should be able to produce the abductive solutions found for possible evolutions, keeping them labeled by the preferences used (in a partial order) instead of exhibiting only the most favorable ones. This would allow for final preference change on the part of the elder.

## References

1. L. M. Pereira, H. T. Anh. *Intention Recognition via Causal Bayes Networks plus Plan Generation*, Procs. 14th Portuguese Conf. on AI (EPIA'09), Springer LNAI, October 2009 (to appear).
2. L. M. Pereira, H. T. Anh. *Evolution Prospection*, in: K. Nakamatsu (ed.), Procs. Intl. Symposium on Intelligent Decision Technologies (KES-IDT'09), pages 51-63, Springer Studies in Computational Intelligence 199, 2009.
3. L. M. Pereira, H. T. Anh. *Evolution Prospection in Decision Making*. Intl. Journal of Intelligent Decision Technologies, IOS Press (to appear in 2009).
4. C. Baral, M. Gelfond, and N. Rushton. *Probabilistic reasoning with answer sets*. In Procs. Logic Programming and Nonmonotonic Reasoning (LPNMR 7), pages 21–33, Springer LNAI 2923, 2004.
5. H. T. Anh, C. K. Ramli, C. V. Damásio. *An implementation of extended P-log using XASP*. In Procs. Intl. Conf. Logic Programming, Springer LNCS 5366, 2008.
6. C. Baral, M. Gelfond, N. Rushton. *Probabilistic reasoning with answer sets*. Theory and Practice of Logic Programming, 9(1): 57-144, January 2009.
7. L. Castro, T. Swift, and D. S. Warren. *XASP: Answer set programming with xsb and smodels*. Accessed at <http://xsb.sourceforge.net/packages/xasp.pdf>
8. A. Cesta, F. Pecora. *The Robocare Project: Intelligent Systems for Elder Care*. AAAI Fall Symposium on Caring Machines: AI in Elder Care, USA 2005.
9. A. Mileo, D. Merico, R. Bisiani. *A Logic Programming Approach to Home Monitoring for Risk Prevention in Assisted Living*. In Procs. Intl. Conf. Logic Programming, Springer LNCS 5366, 2008.
10. M. V. Giuliani, M. Scopelliti, F. Fornara. *Elderly people at home: technological help in everyday activities*. IEEE International Workshop on In Robot and Human Interactive Communication, pp. 365-370, 2005.

- 
11. C. Heinze. *Modeling Intention Recognition for Intelligent Agent Systems*, Doctoral Thesis, the University of Melbourne, Australia, 2003. Online available: [http : //www.dsto.defence.gov.au/publications/scientific\\_record.php?record = 3367](http://www.dsto.defence.gov.au/publications/scientific_record.php?record=3367)
  12. K. A. Tahboub. *Intelligent Human-Machine Interaction Based on Dynamic Bayesian Networks Probabilistic Intention Recognition*. J. Intelligent Robotics Systems, vol. 45, no. 1, pages 31-52, 2006.
  13. O. C. Schrempf, D. Albrecht, U. D. Hanebeck. *Tractable Probabilistic Models for Intention Recognition Based on Expert Knowledge*, In Procs. 2007 IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS 2007), pages 1429–1434, 2007.
  14. H. A. Kautz and J. F. Allen. *Generalized plan recognition*. In Procs. 1986 Conf. of the American Association for Artificial Intelligence, AAAI 1986: 32-37, 1986.
  15. T. Eiter, W. Faber, N. Leone, G. Pfeifer, A. Polleres. *A Logic Programming Approach to Knowledge State Planning, II: The  $DLV^K$  System*. Artificial Intelligence 144(1-2): 157-211, 2003.
  16. P. H. Tu, T. C. Son, C. Baral. *Reasoning and Planning with Sensing Actions, Incomplete Information, and Static Causal Laws using Answer Set Programming*. Theory and Practice of Logic Programming, 7(4): 377-450, July 2007.
  17. C. Glymour. *The Mind's Arrows: Bayes Nets and Graphical Causal Models in Psychology*. MIT Press, 2001.
  18. J. Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge U.P., 2000.
  19. J. J. Alferes, A. Brogi, J. A. Leite, L.M. Pereira. *Evolving logic programs*. Procs. 8th European Conf. on Logics in AI (JELIA'02), pages 50–61, Springer LNAI 2424, 2002.
  20. J. J. Alferes, F. Banti, A. Brogi, J. A. Leite. *The Refined Extension Principle for Semantics of Dynamic Logic Programming*, Studia Logica 79(1): 7-32, 2005.
  21. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, T. C. Przymusinski. *Dynamic updates of non-monotonic knowledge bases*. J. Logic Programming, 45(1-3):4370, 2000.
  22. I. Niemelä, P. Simons. *Smodels: An implementation of the stable model and well-founded semantics for normal logic programs*. 4th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning, Springer LNAI 1265, pages 420–429, 1997.
  23. T. Swift. *Tabling for non-monotonic programming*. Annals of Mathematics and Artificial Intelligence, 25(3–4):201-240, 1999.
  24. *The XSB System Version 3.0 Vol. 2: Libraries, Interfaces and Packages*. July 2006.



---

# Knowledge Management Strategy and Tactics for Forging Die Design Support

Masanobu Umeda<sup>1</sup> and Yuji Mure<sup>2</sup>

<sup>1</sup> Kyushu Institute of Technology, 680-4 Kawazu, Iizuka 820-8502, Japan  
umerin@ci.kyutech.ac.jp

<sup>2</sup> Kagoshima Prefectural Institute of Industrial Technology, 1445-1 Hayato-Oda,  
Kirishima, Kagoshima 899-5105, Japan  
mure@kagoshima-it.go.jp

**Abstract.** The design tasks of cold forged products are very important because they have great influence on the quality of final products and the forging die life. The design expertise required for the design is, however, about to be scattered and lost, and therefore a framework for enabling the accumulation, utilization, and evolution of expertise is greatly needed. The authors have been developing a knowledge-based system to support process planning and the design of dies and die-sets for cold forged products. The knowledge base contains design knowledge regarding cold forging, and the design knowledge is systematized based on the formal definition of die design problems and on methods with high generality for resolving these problems. These features allow the improvement of product quality by an exhaustive search of process plans and die structures, and the improvement of design performance by automating design work. They also make it easier to pass on and evolve expertise by allowing design experts to audit and maintain a knowledge base by themselves. The proposed system has been applied to several forged products. Experiments show that it can generate multiple forging process plans, dies, and die-sets based on the design knowledge, and that feasible solutions can be obtained within a practical amount of time.

## 1 Introduction

Parts formed by cold forging are popularly used in many products such as automobiles, aircrafts, and home electrical appliances. This is because they have the advantages of productivity in mass production, dimensional accuracy, and improvements in mechanical properties. The design tasks of cold forged products are very important because they have great influence on the quality of final products and the forging die life. In these design tasks, many kinds of expertise in forging process planning and the design of dies and die-sets are required for high-quality and low-cost production. Such expertise is, however, about to be scattered and lost because of the impending retirement of experienced experts. This expertise has been little documented and has been transferred as individual skills through informal communication. In order to systematically pass on this

---

expertise, it is essential to realize a framework which enables the design knowledge of experienced experts to be accumulated as important software assets, and continuously utilized and evolved in an organization or industrial community.

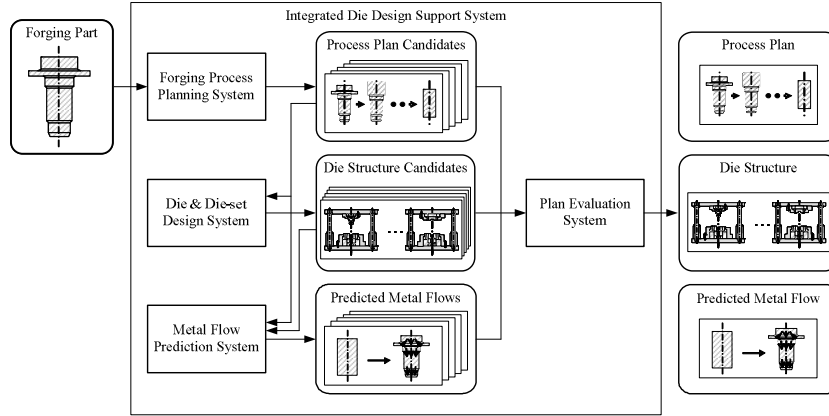
A number of knowledge-based systems have been developed to ensure the productivity of forging process planning and die design. They can be roughly classified into the following two categories. Case-based or pattern-based systems [1,2] formulate past successful design results as design cases or patterns, and derive a solution from a similar case. Although these systems have the advantage of obtaining adequate results, the variety of applicable products strongly depends on the cases or patterns that have been accumulated. Conventional rule-based systems [3–7] represent design guidelines or expertise as production rules, and have the advantages of adaptability to new products. However, production rules are difficult to maintain and extend, because interference between rules, and combinatorial explosion and execution control of rules become serious problems if a knowledge base becomes large [8].

The authors have been developing a knowledge-based system to support process planning and the design of dies and die-sets for cold forged products. The knowledge base contains design knowledge regarding cold forging, and the design knowledge is systematized based on the formal definition of die design problems and on methods with high generality for resolving these problems. The knowledge representation language of the knowledge base is suitable for representing design procedures involving trial and error without any side-effects, and can be used to comprehensively enumerate all possible solutions of forging processes and die structures. These features allow the improvement of product quality by an exhaustive search of process plans and die structures and the improvement of design performance by automating design work. They also make it easier to inherit the available expertise and evolve it by allowing design experts to audit and maintain a knowledge base by themselves.

This paper first describes an overview of the integrated die design support system for cold forging, design object models of a forged product and a die, and the formalization and systematization methodology of forging die design knowledge. The implementation of the system is then briefly introduced. Finally, experimental results applied to several forged products are shown.

## 2 System Overview

Generally, the design process of a forged product is as follows. First, the shape of a forged product is designed based on a final product design by considering post machining processes. Then, forging process planning is performed to decide the stage number of the forming process, the shape of an intermediate forged product, the forming methods, etc. In the planning, various design constraints, such as the load capacity of press machines and the forming limits and metal flow of materials are taken into account. Finally, dies and die-sets are designed for each stage of the forming processes by considering the product quality and the die life. If necessary, the metal flow is evaluated using experimental analysis or FEM



**Fig. 1.** Overview of the integrated die design support system

analysis. Although the degree of geometric deformation freedom is generally high in forging process planning, design constraints, such as the forming limits and metal flow of materials, are strong. Therefore, it is difficult even for experienced experts to find an appropriate solution from countless process plans and the die structures.

The integrated die design support system is intended to consistently support this design process. It consists of the following four subsystems shown in Fig. 1: forging process planning, die and die-set design, metal flow prediction, and plan evaluation.

The forging process planning system is to support the planning of a cold forging process for a given axisymmetrical forged product. It can propose multiple deformable process plans using the Adjustment of Stepped Cylinder (ASC) method [9]. Because the ASC method is based on a combination of simple geometric deformation and fundamental design constraints, it can generate appropriate plans even if no similar best practices can be found in a case-based knowledge base.

The die and die-set design system generates multiple die structures and detailed shapes of their components, such as die inserts, rings, and knock-out pins, for the generated process plans. Because dies are subjected to high stress, they are prone to breakage, such as cracks. Therefore, the die life has to be taken into account by considering empirical actions, such as partitioning a die insert into pieces and stacking pressure pads that receive the pressure on dies.

Forming limits of materials, such as the reduction in area, are taken into account by these two subsystems. It is, however, still necessary to evaluate the metal flow using experimental analysis or FEM analysis in a case of an unknown or unfamiliar shape, because it is not sufficient to evaluate the surface and internal defects. It is, however, time consuming or impractical to evaluate the metal flow of all generated solutions using the usual analysis methods. The metal flow











prediction system [10] is designed to help make a rough and quick quantitative evaluation of metal flow by using the knowledge of experienced experts and deformation characteristics.

Candidates of forging process plans and die structures are often obtained from these two subsystems. Therefore, it is not easy to choose an optimal candidate manually, because many aspects of the candidates must be kept in mind. The plan evaluation system is designed to help choose an optimal candidate by evaluating the candidates overall in terms of product quality, production cost, and production lead time.

### 3 Modeling of Forged Product and Die

#### 3.1 Shape Representation

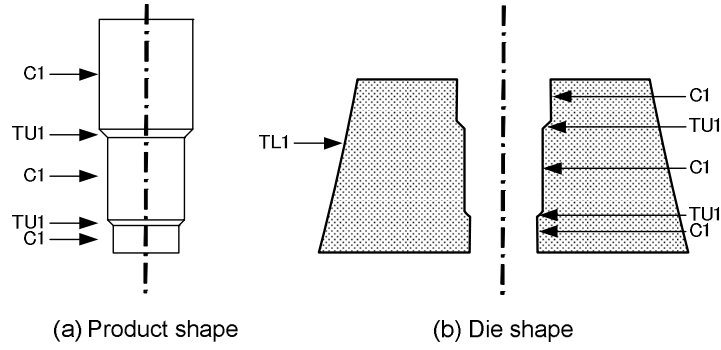
Because the shape of a forged product is axisymmetric, the principal shapes of companion forging dies are also axisymmetric. Therefore, both a forged product shape and a die shape can be represented as a series of axisymmetrical representation units called **basic elements** (BEs). Figure 2 shows a list of the basic elements. Each basic element has geometrical attributes, such as element type, diameter, and height. Figure 3 shows examples of the shape representations of a forged product and a die. In these examples, the shape of a forged product can be represented by an outer shape [C1,TU1,C1,TU1,C1]. The shape of a hollow die can be represented by the combination of an outer shape [TL1] and an inner shape [C1,TU1,C1,TU1,C1]. This uniform representation simplifies the shape representation of a forged product and a die, and the knowledge description needed for forging process planning and die design can then easily be integrated.

	1	2	3	4
C				
TU				
TL				

**Fig. 2.** Basic elements for shape representation

#### 3.2 Object Model Representation

A forged product, a die, and a die-set are constructively represented as a design object model called an **assembly structure**, which is simplified for use with an



**Fig. 3.** Shape representations of a product and a die using basic elements

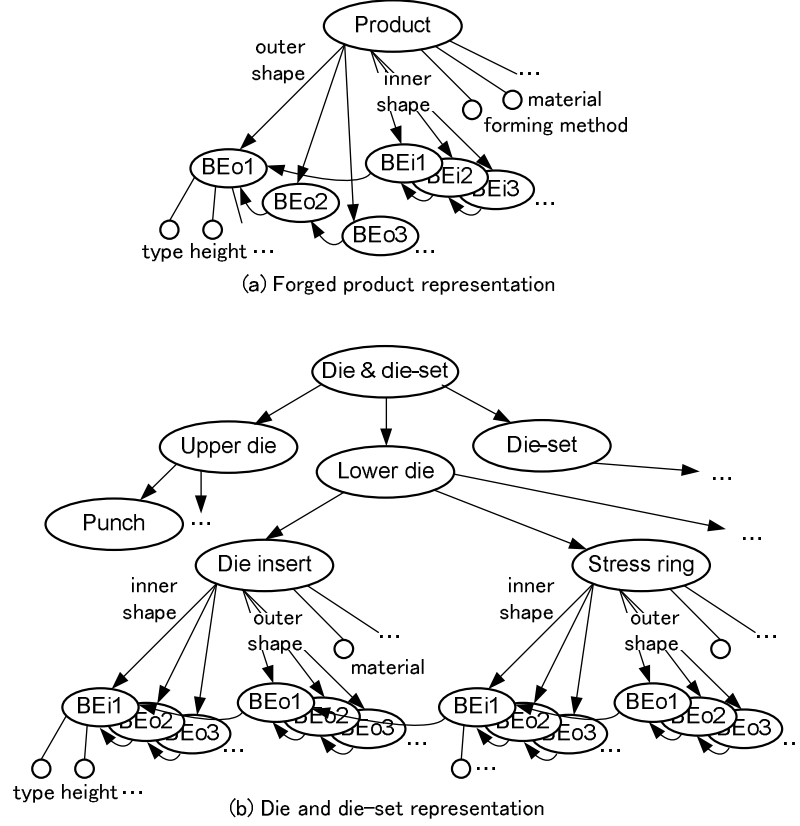
axisymmetrical object based on the model proposed for mechanism parts [11]. Figure 4 illustrates the assembly structure of a forged product and that of a die and a die-set, where the arrows indicate the dependency between elements. The assembly structure of the forged product in Fig. 4 (a) represents a billet, an intermediate forged product, or a final forged product. The object model of a forged product has outer and inner shapes represented by a series of basic elements, and other attributes, such as material, forming method, and forming load. On the other hand, the assembly structure of the die and die-set in Fig. 4 (b) represents a whole structure of an upper die, a lower die, and a die-set. The object model of a part, which constructs an upper die, a lower die, or a die-set, has outer and inner shapes and other similar attributes to that of a forged product.

The model based on the assembly structure is easy to extend to deal with dimensional and geometrical tolerance. This feature is very important because accuracy management of a forged product and a die is essential in cold forging.

## 4 Formalization of Die and Die-set Design Problems

Generally speaking, most design tasks can be seen as activities to decide structural attributes from required functional attributes by solving an inverse problem<sup>3</sup>. Because these kinds of problems are often nonlinear, it is difficult to solve them analytically, and they thus used to be solved by trial and error based on experiences or intuition. Deriving approximate solutions from past design results or design constraints using back calculation is an example of how these problems have been solved. These heuristic methods are, however, usually unique to each designer or organization. The design result, productivity, and product quality of a designer usually differ from those of other designers, even for the

<sup>3</sup> On the other hand, a direct problem is solved by deriving functional attributes from structural attributes. This is often called analysis.



**Fig. 4.** Assembly structures of a forged product and a die and die-set

same problem. For this reason, even if pieces of heuristic knowledge are collected from experienced experts through interviews and described as production rules, the generality of this knowledge is quite low and its applicability is limited to a narrow range of problems. A knowledge base developed in this way is difficult to maintain and extend and is thus likely to become out of date. In addition, it would obviously be more effective and efficient for experienced experts to develop and maintain a knowledge base by themselves.

For this reason, it is essential to provide a formalization methodology to allow design experts to reorganize, describe, audit, and maintain their design knowledge by themselves, and to also enable them to represent individual heuristic knowledge in as general a form as possible.

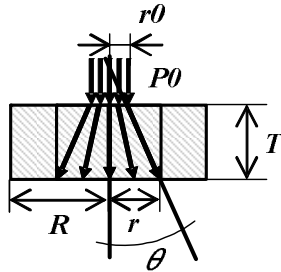
#### 4.1 Formal Definition of Design Problem

The authors have applied the design space [12] to the formal definition of die and die-set design problems. The design space  $Ds$  is a set of design solutions

intentionally defined by necessary and sufficient design attributes and the design constraints between them. It is represented as  $\langle P, D, R, C \rangle$ , where  $P$  is a set of design attributes,  $D$  is a set of the domains of the attributes,  $R$  is a set of the relationships between the attributes, and  $C$  is a set of the constraints on the attributes.

Figure 5 illustrates the pressure pads and design space of the pressure pads design. The pads belong to a die-set, and are used for absorbing the forming pressure and preventing the deformation of a press machine. This design problem is to decide the number  $N$ , thickness  $T$ , and radius  $R$  of the pads from other attributes, such as the receiving radius  $R0$ , receiving pressure  $P0$ , and pressure propagation angle  $\theta$ , where  $P$  is the propagated pressure and  $r$  is the radius where the pads actually receive the pressure. The solutions must satisfy that  $P$  is smaller than a load limit **LoadLimit** of a press machine.

If a design problem is large and complicated, designers often solve it by dividing it into several subproblems and solving them step by step. Such a complicated design problem can be represented as a concatenation of design spaces representing subproblems. The concatenation indicates the order in which the subproblems need to be solved, and a set of its solutions can be expressed as a result of several join operations over the design spaces. The join operation, such as natural join and direct product, is determined by how a preceding design space and a following design space is combined and the dependency between them. An example of the concatenation of design spaces is shown in section 5.



(a) Pressure pads

$$\begin{aligned}
 Ds &= \langle P, D, R, C \rangle \\
 P &= \{r0, \theta, P0, r, T, P, R, N\} \\
 D &= \{r0, \theta, P0, r, T, P, R \in \text{real}, \\
 &\quad N \in \text{int}\} \\
 R &= \{r = r0 * (3 / (3 - 2 * \tan(\theta)))^{(N-1)}, \\
 &\quad T = (r - r0) / \tan(\theta), \\
 &\quad R = 1.5 * r, \\
 &\quad P = (r0 / r)^2 * P0 \\
 &\quad \} \\
 C &= \{P < \text{LoadLimit}\}
 \end{aligned}$$

(b) Design space of pressure pads design

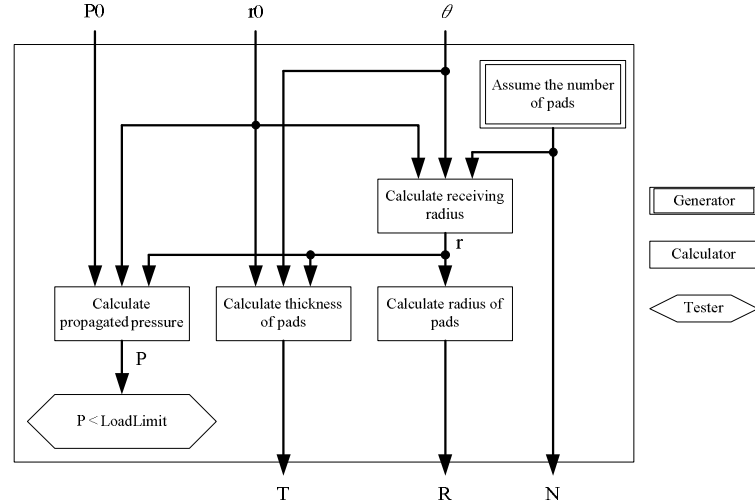
Fig. 5. Design space of pressure pads design

## 4.2 Resolving Method of the Design Space

If a design problem is difficult to solve analytically, a trial and error process is usually necessary. Traditional AI techniques for combinatorial optimization

problems are ideally applicable to such design problems in many cases, but most of these techniques are not adequate for describing design knowledge in a manageable and extendable way. This is because some are limited in applicable scope and may become obstacles to the future extension of a knowledge base, and other techniques are difficult for design experts to use them to describe, audit, and maintain a knowledge base by themselves.

The generate and test method is a very simple but general method for resolving such a problem in a manageable and extendable way. A design problem defined by a design space can be easily formulated using the generate and test method. That is, for a given design space  $\langle P, D, R, C \rangle$ , the solutions can be obtained by assuming the values of base attributes within  $D$ , where base attributes are members of a necessary and sufficient subset of  $P$  for specifying a unique design solution, calculating values of other dependent attributes from the base attributes using  $R$ , and verifying that  $C$  is satisfied. Figure 6 illustrates a data-flow diagram of the resolving method based on the generate and test method for the design space of pressure pads design in Fig. 5.



**Fig. 6.** Data-flow diagram for pressure pads design

## 5 Systematization of Die and Die-set Design Knowledge

Parts composing a die and a die-set can be roughly classified into a forming portion, a pressure receiver, and a mounting portion according to their roles. The design procedure of a die and a die-set is standardized based on this classification. The forming portion which is prone to breakage is first designed according to the



results of forging process planning, and then the pressure receiver is designed to withstand the forming pressure. Finally, the mounting portion, by which the forming portion and pressure receiver are mounted on a press machine, is designed. Figure 7 illustrates the overall structure of these portions. In this section, the systematization methodology of design knowledge of a die and a die-set is introduced using examples of the forming portion.

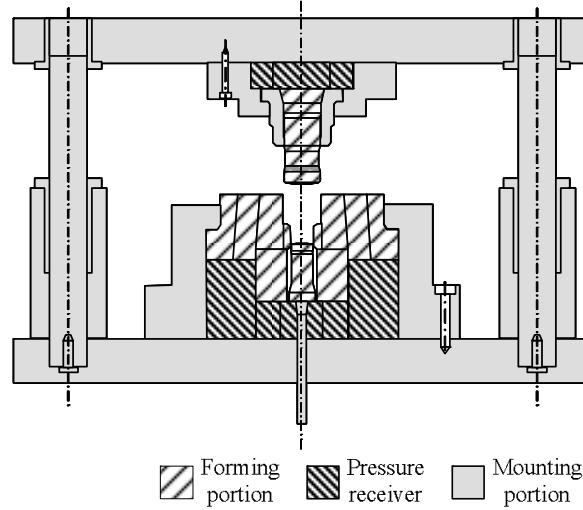


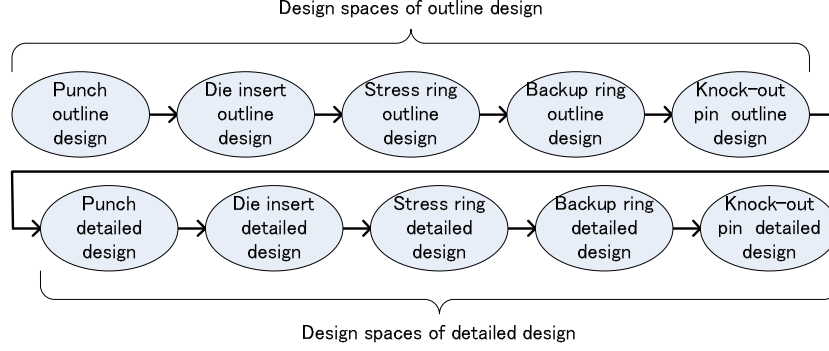
Fig. 7. Overall structure of a die and a die-set

### 5.1 Design Space of Forming Portion Design

The forming portion, which consists of a punch, a die insert, stress rings, backup rings, and knock-out pins, is directly related to the forming of forged products. Because it is subjected to high stress, the shapes of its parts have to be designed with forming methods and loads in mind in order not to be broken. The design procedure of the forming portion was divided into an outline design and a detailed design. First, the outline shapes of the forming portion are decided according to a process plan by taking into account design constraints, such as the intensity and press fit ratio of a die. A number of solutions with different die structures may be available for the same forged or intermediate product if multiple forming methods are applicable. The detailed shapes are then determined by taking into account design constraints, such as the deformation characteristics of materials and the reduction in area.

The design space of the forming portion design can be represented as a concatenation of ten design spaces. Figure 8 illustrates these design spaces. Five

of them from the beginning correspond to the outline design, and the rest to the detailed design. In the following, the design space of the die insert detailed design is briefly explained as an example.



**Fig. 8.** Concatenated design spaces of the forming portion design

## 5.2 Design Space of Die Insert Detailed Design

In the detailed design of a die insert, the partition of a die insert, creation of lands, and position alignment of the upper and lower dies have to be solved. For the sake of simplicity, this paper focuses on the partition of a die insert.

Because the forming portion is subjected to high stress as described before, a die insert and a punch are often damaged by metal fatigue. In such a case, it is usually possible to avoid breakage by changing their geometries in order to reduce the concentration of the stress. In the case of a die insert, it is effective to partition the die insert into two parts around a stress concentration zone. Repairing the partition of a die insert is low cost, because it is possible to replace only the part of the partitioned die insert that is broken. On the other hand, the partition has disadvantages, because it is costly to produce due to its increased number of components and complicated geometry.

There are roughly two parting methods of a die insert, as illustrated in Fig. 9. The vertical partition (a) is suitable for upsetting because a die insert is subjected to especially high stress. The horizontal partition (b) is, instead, suitable for forward extrusion where the reduction in area is not very large, but it is applicable to other forming methods. Figure 10 is a data-flow diagram for die insert partition design in case of forward extrusion. In this diagram, either of the parting methods is assumed only when the constraints on forming load and the reduction in area are satisfied, and the shapes of partitioned die inserts are decided according to the assumed parting method.

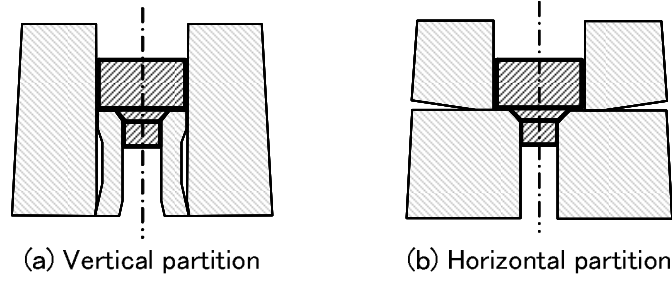


Fig. 9. Parting methods of a die insert

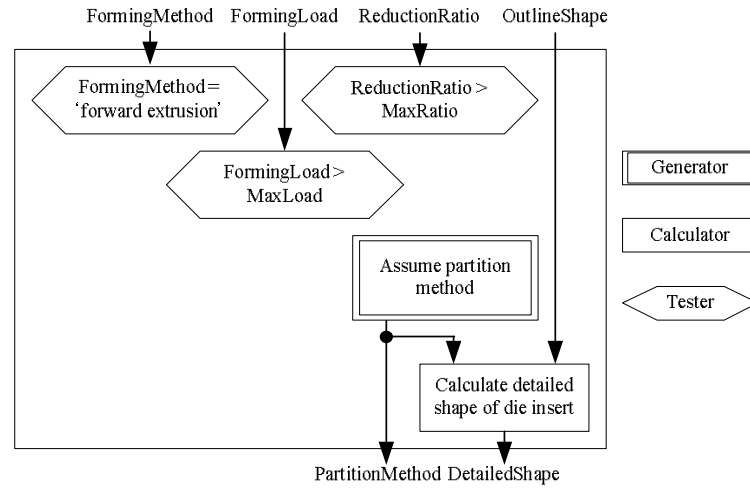
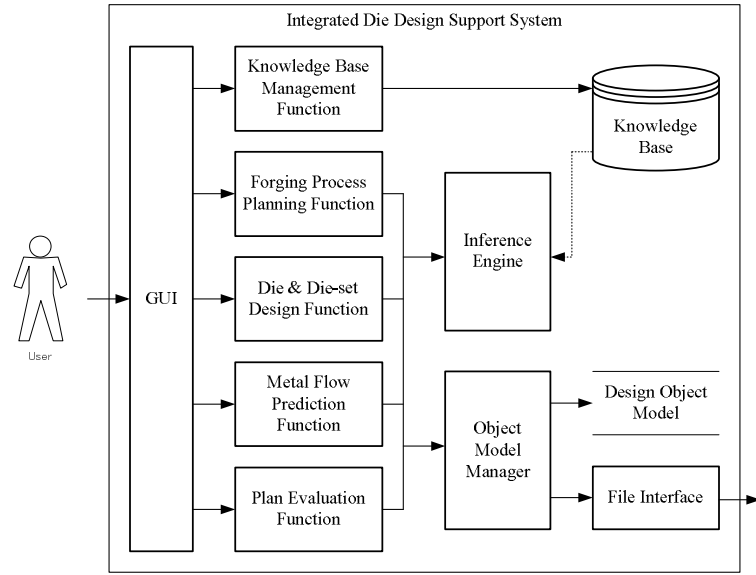


Fig. 10. Data-flow diagram for die insert partition design

## 6 System Implementation

The authors have been developing an integrated die design support system based on the proposed methodology. Figure 11 illustrates its overall system architecture. The knowledge base stores systematized knowledge for forging process planning, die and die-set design, metal flow prediction, and plan evaluation. The inference engine performs reasoning using the knowledge base and elicits all possible solutions. The forging process planning function, die and die-set design function, metal flow prediction function, and plan evaluation function utilize the knowledge base through the inference engine, and present the solutions to the user.

The contents of the knowledge base can be classified into two: inference rules which represent processes, tasks, and procedures of design work, and engineering information, such as design constraints and design standards. The former are



**Fig. 11.** System architecture of the integrated die design support system

described using a knowledge representation language named DSP [13], which is based on attribute grammars [14]. DSP is a functional programming language having the search capability using the generate and test method. Because the language is capable of representing trial and error without any side-effects or loop constructs and knowledge descriptions using the language can be declaratively read and understood, it is suitable for representing design procedures according to systematized knowledge using the design space. The resolving method represented by data-flow diagrams can be translated into executable DSP programs with almost one-to-one correspondence. On the other hand, the engineering information is described in a knowledge representation language named PUBLIB [15], which is suitable for representing relationships between attributes, such as dimensions of standard parts and characteristics of materials, in a structured manner. Separating the contents of the knowledge base into two helps maintain the knowledge base and make it portable, because the latter type of knowledge has more locality or individuality between different organizations than the former.

All components of the system except for the knowledge base are implemented using an integrated development environment called Inside Prolog [16]. Knowledge descriptions stored in the knowledge base are translated into Prolog programs by the knowledge base management function and executed by the inference engine. Because Inside Prolog provides standard Prolog functionality and a large variety of application programming interfaces which are essential for practical application development, it is easy to integrate the inference engine and other

supporting functions in one development environment. The knowledge base and supporting functions for metal flow prediction and plan evaluation are still under development, while prototypes of the knowledge base and supporting functions for the others have been implemented. A primary user interface of the system is shown in Fig. 12.

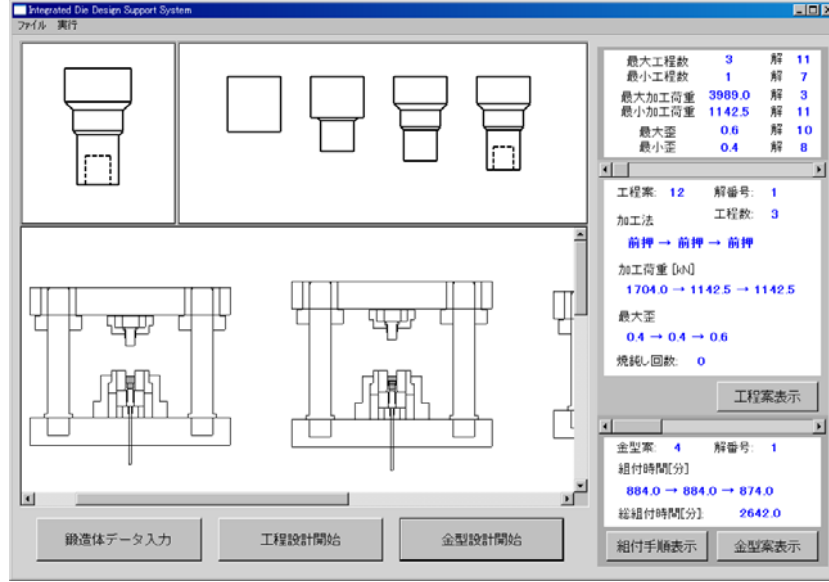


Fig. 12. Screen shot of the integrated die design support system

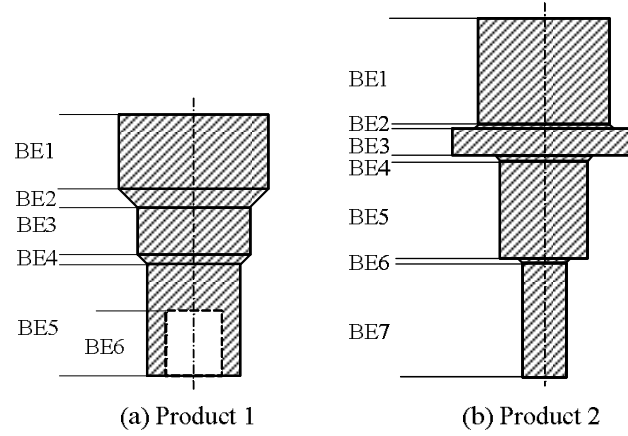
## 7 Experimental Results

The integrated die design support system was applied to several examples of forged products. Figure 13 shows two of them, and Fig. 12 shows the results applied to product 1. In this example, 12 forging process plans are generated within a second, and 4 solutions for the die structure are generated for all stages of one of the forging process plans within a second. Figure 14 shows one of the process plans, and Fig. 15 shows the corresponding result of the die and die-set design. The differences between the 4 solutions of die structure come from the applied parting methods of the die inserts.

These experiments show that the system can generate multiple forging process plans and die structures based on design knowledge, and that feasible solutions can be obtained within a practical amount of time. The solutions have different characteristics in terms of the stage number of forming processes, maximum forming load, maximum strain, the number of annealing processes, the

---

partition of die inserts, etc., and an appropriate solution can be chosen from them.



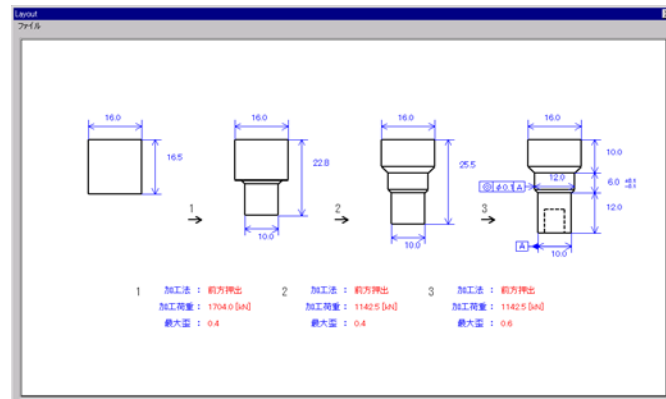
**Fig. 13.** Forged product examples

## 8 Conclusions and Future Work

A framework for enabling the accumulation, utilization, and evolution of expertise is strongly needed for supporting process planning and the design of dies and die-sets of cold forged products, because such essential expertise is about to be scattered and lost. This paper describes the formalization and systematization methodology of forging die design knowledge to enable designers to develop, audit, maintain and extend a knowledge base by themselves. It also describes an overview and brief implementation of the integrated die design support system based on the proposed methodology.

In this framework, design problems of forging dies and die-sets are formally defined by using the design space, and the resolving method of the design space can be represented by a data-flow diagram based on the generate and test method. The resolving method represented by data-flow diagrams can be easily translated into executable programs with almost one-to-one correspondence using a functional programming language based on attribute grammars.

The integrated die design support system is designed to consistently support process planning and the design of dies and die-sets. It can comprehensively enumerate all possible solutions of forging processes and die structures, and help choose an optimal solution from all of the solutions. The system was applied to examples of forged products, and it was shown that the system could generate multiple forging process plans and die structures based on the design knowledge, and that feasible solutions could be obtained within a practical amount of time.



**Fig. 14.** Example of a forging process plan

As was briefly mentioned in section 3, accuracy management is essential in cold forging. It is essential to incorporate tolerance information of forged products into the system and produce process plans and die structures so that the accuracy of forged products is satisfied. The metal flow prediction system and plan evaluation system are still under development, and further research is necessary for practical use.

## ACKNOWLEDGMENT

The authors are grateful to Dr Osamu Takata, who developed the foundation of this system before he passed away in the prime of life.

## References

1. Lange, K., Du, G.: A formal approach to designing forming sequences for cold forging. Trans. of the NAMRI/SME (1989) 17–22
2. Takata, O., Nakanishi, K., Yamazaki, T.: Forming-sequence design expert system for multistage cold forging: Forest-d. In: Proc. of Pacific Rim International Conference on Artificial Intelligence '90. (1990) 101–113
3. Sevenler, K., Raghupathi, P.S., Altan, T.: Forming-sequence design for multistage cold forging. J. of Mechanical Working Technology **14** (1987) 121–135
4. Mahmood, T., Lengyel, B., Husband, T.M.: Expert system for process planning in the cold forging of steel. Expert Planning Systems **322** (1990) 141–146
5. Kim, H.S., Im, Y.T.: An expert system for cold forging process design based on a depth-first search. Journal of Materials Processing Technology **95** (1999) 262–274
6. Kumar, S., Singh, R.: A low cost knowledge base system framework for progressive die design. Journal of Materials Processing Technology **153** (2004) 958–964
7. Xuewen, C., Siyu, Z., Jun, C., Xueyu, R.: Research of knowledge-based hammer forging design support system. The International Journal of Advanced Manufacturing Technology **27** (2005) 25–32

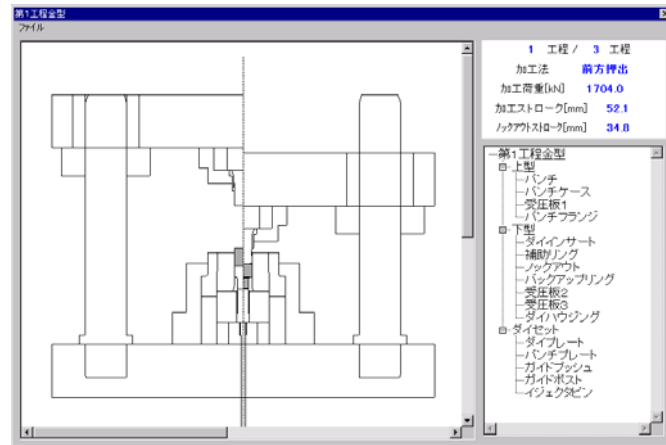


Fig. 15. Example of a die and die-set

8. Ueno, H.: Expert systems. *Journal of Information Processing Society of Japan* **28** (1987) 147–157
9. Takata, O., Mure, Y., Nakashima, Y., Ogawa, M., Umeda, M., Nagasawa, I.: A knowledge-based system for process planning in cold forging using the adjustment of stepped cylinder method. In: *Declarative Programming for Knowledge Management*. Volume 4369 of *Lecture Notes in Computer Science*., Springer Science+Business Media (2006) 161–174
10. Umeda, M., Yamazumi, S., Mure, Y.: Development of knowledge-based metal flow prediction system in cold forging. In: *The Proceedings of the 59th Japanese Joint Conference for the Technology of Plasticity*. (2008) 325–326
11. Nagai, T., Nagasawa, I., Umeda, M., Higuchi, T., Nishidai, Y., Kitagawa, Y., Tsurusaki, T., Ohhashi, M., Takata, O.: A design product model for mechanism parts by injection molding. In: *Declarative Programming for Knowledge Management*. Volume 4369 of *Lecture Notes in Computer Science*., Springer Science+Business Media (2006) 148–160
12. Yamaguchi, H., Nagasawa, I., Umeda, M., Mochizuki, M., Zhihua, Z.: Knowledge representation model for basic design of power plant equipment. *Transactions of Information Processing Society of Japan* **41** (2000) 3180–3192
13. Umeda, M., Nagasawa, I., Higuchi, T.: The elements of programming style in design calculations. In: *Proceedings of the Ninth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*. (1996) 77–86
14. Deransart, P., Jourdan, M., eds.: *Attribute Grammars and their Applications*. Number 461 in *Lecture Notes in Computer Science*. Springer-Verlag (1990)
15. Umeda, M., Nagasawa, I., Ito, M.: Knowledge representation model for engineering information circulation of standard parts. *Transactions of Information Processing Society of Japan* **38** (1997) 1905–1918
16. Katamine, K., Umeda, M., Nagasawa, I., Hashimoto, M.: Integrated development environment for knowledge-based systems and its practical application. *IEICE Transactions on Information and Systems* **E87-D** (2004) 877–885



---

# Network Monitoring with Constraint Programming: Preliminary Specification and Analysis

Pedro Salgueiro and Salvador Abreu

Departamento de Informática  
Universidade de Évora  
`{pds,spa}@di.uevora.pt`

**Abstract.** Network Monitoring and Intrusion Detection Systems plays an important role in today's computer networks health, allowing the diagnosis and detection of anomalous situations on the network that could damage the performance and put the security of users data in risk if not detected or diagnosed in time to take any necessary measures. In this paper we present a preliminary specification and analysis of a network monitoring and intrusion detection concept system based on constraint programming, implemented on several constraint solver systems. This concept allows to describe the desirable network situations through constraints on network entities, allowing a more expressive and clear way of describing network situations.

## 1 Introduction

Computer networks are composed of numerous complex and heterogeneous systems, hosts and services. Often, large numbers of users rely on such networks to perform their regular job and have to interrupt it when, for some reason, the network has a problem. In order to prevent computer network degradation or even any failure preventing users from using it, computer networks should be constantly monitored in order to diagnose its functionality as well as the security of the network and its users data.

Constraint Programming [1,2,3] is a powerful programming paradigm for modeling and problem solving and it has been widely used with success on a variety of problems such as scheduling, vehicle routing, planning and bio-informatics. Using constraint programming, the modeling of complex problems is done by asserting a number of constraints over a set of variables in a natural way, simplifying the modelling of complex problems. The constraints are then solved by a constraint solver which assigns values to each of the variables, solving the problem.

---

On a computer network, there are some aspects that must be verified in order to maintain the quality and integrity of the services provided by the network as well as to ensure its safety. The description of those conditions, together with a verification that they are met can be seen as network performance monitoring combined with an intrusion detection task.

These two aspects are usually intertwined, from a network point view, so that it makes sense to provide a single specification for the desired system and network properties which can be used to produce network activity monitoring agents, with knowledge about conditions which must or must not be met. These conditions, specified in terms of properties of parts of the (observed) network traffic, will amount to a specification of a desired or an unwanted state of the network, such as that brought about by a system intrusion or another form of a malicious access.

It is our claim that the proper operation of a network can be described as a *constraint satisfaction problem* (CSP) [1], by using variables which represent parts of the network traffic as well as structural properties of the network itself.

## 2 Constraint Programming

Constraint Programming (CP) is a declarative programming methodology which consists in the formulation of a solution to a problem as a *constraint satisfaction problem* (CSP) [1], in which a number of variables are introduced, with well-specified domains and which describe the state of the system. A set of relations, called *constraints*, is then imposed on the variables which make up the problem. These constraints are understood to have to hold true for a particular set of bindings for the variables, resulting in a *solution* to the CSP.

Finite Domain Constraint Programming is one of the most practical and widely used approaches to Constraint Programming [1]. Many systems and libraries have been developed to perform Finite Domain Constraint Programming and rely on constraint propagation involving variables ranging over some finite set of integers to solve the problems.

Some of the more well known FD constraint programming systems include GNU Prolog [4] and SICStus Prolog [5] which extend a Prolog virtual machine with capabilities of constraint programming. Besides those systems, there are libraries which provide Constraint Programming capabilities to well known programming languages, such as Gecode [6] a constraint solver library implemented in C++. Gecode is particular, in that it is not designed to be used directly for modeling but is meant to be interfaced with other systems, such as Gecode/R [7], a Ruby interface to Gecode. CaSPER [8] (Constraint Solving Programming Environment for Research) is a new and open C++ library for generic constraint solving, that still needs to be matured.

---

The results presented in this paper were obtained using the previously mentioned constraint system solvers.

### 3 Modeling Network Monitoring with Constraint Programming

Computer Network Monitoring with Constraint Programming relies on the description of situations that should or should not be seen on a network traffic. The description of such situations is done by stating constraints over network entities, such as network packets, ip address, ports, hosts, among others. Network packet is the entity more relevant in the description of a network situation, since it uses most of the other entities and allows to describe all the traffic on a given network, which is composed by 19 individual fields.

The solution to such a problem that describes network situations is a set of network packets that satisfies all the constraints that describes the network situation. If that set of packets is seen on a network traffic, then that network situation exists on that network.

Our approach to network monitoring works offline, instead of working with real time traffic, it uses network traffic logs and looks for the specified network situations in those logs, which were generated by the packet sniffer tcpdump[9].

Using constraint programming, the variables that are used to model the problem can take any value that satisfies the stated constraints. In the case of network monitoring, the domain of the packet variables is huge, since each packet is composed by 19 individual fields, each one having its own large domain. If the domain of such variables were completely free, but respecting the constraints, the amount of valid values that could be assigned to a variable which satisfies the constraints is huge, so it makes sense that the packet variables can only be assigned with values that satisfies the stated constraints and that exists on the network traffic log. In Sect. 4 are described the several approaches that we used to force the network packet variables to only take values from a network traffic log.

### 4 Network traffic log as variable domain

Network monitoring based with constraint programming relies on stating rules and relations between network packets. Such network packets are composed by 19 fields, almost all of them important do describe a network situation. Such fields have a very wide domain and most of the values that can be assigned to such fields make no sense on the computer network that is being monitored. The only values that would make sense to be assigned to each field of each network

---

packet involved in a description of a network situation would be the actual values that exist on the network traffic.

The values of such fields should be chosen from the network log, but not the individual fields of the packet, since the values of each field on the network traffic log only make sense when combined with all the other fields of a network packet.

So, instead of each individual field of a packet be allowed to take values from its correspondent field of any packet on the traffic log, a tuple representing a network packet and composed with all the fields of a network packet can only take values from the set of tuples available on the network traffic log, this way, the network packet variables used on a network description can only take values that really exists, eliminating any unnecessary values.

#### 4.1 Approaches

In order to force the network packet variables to only be allowed to take values that belong to a network traffic log, several approaches were taken.

In this implementation of a network monitoring concept based on constraint programming, the network packet variables are represented as an array of integer variables, and the traffic logs were converted into a matrix that represents all the packets available on the log, each line of the matrix corresponding to a packet of the log.

**Using Extensional constraints** The first approach to this problem was to use *extensional* constraints. *Extensional* constraints receives as its input a tuple of variables and a list of tuples of values or variables. In this application domain, the tuple of variables is the array representing a network variable, which is composed by several variables, and the list of tuples is the list with all packets found on the network log. This constraint forces the tuple of variables to take values from the list of tuples, which on this case forces the packet variables to only take values from the network traffic log. The results obtained when using this approach are described in Sect. 5.1.

The use of the extensional constraint can be described as follows:

$$P = (P_1, \dots, P_{19}),$$

$$M = \{(V_{(1,1)}, \dots, V_{(1,19)}), \dots, (V_{(k,1)}, \dots, V_{(k,19)})\},$$

$$\forall P_i \in P, \forall M, \text{extensional}(P, M) \Rightarrow P \in M$$

where  $P$  represents a network packet variable and  $M$  a set of packets representing all possible values that packet  $P$  can take.

---

**Using Extensional and Element constraints** A second approach to the problem combines the *extensional* constraint with the *element* constraint. The *element* constraint allows to use an un-instantiated variable as an index to an array of values or variables. This allows to translate the matrix representing the network packets into a matrix of indexes to an array with all values that exists on the original matrix. By adopting this approach, one can use the *extensional* constraint on this translated matrix, which only have indexes, which have much smaller values than the original matrix, which could improve the performance in a great scale. Using the *extensional* with the translated matrix, the values assigned to the packet variables are indexes to an array containing all the values of the original matrix, instead of the values that compose the original packets. This poses a problem, since after applying the constraints to state that the packet variables should only take values from the network traffic log, it is also necessary to state other constraints over such variables using the values of the original matrix. To solve this problem, the variables are *bonded* back to its original values by using the *element* constraint, allowing to use constraints over the original values. The results obtained by this approach are described in Sect. 5.2.

The use of the extensional constraint combined with the element constraint can be described as follows:

$$\begin{aligned}
P &= (P_1, \dots, P_{19}), \\
M &= \{(V_{(1,1)}, \dots, V_{(1,19)}), \dots, (V_{(k,1)}, \dots, V_{(k,19)})\}, \\
T &= \{V_{(1,1)} \cup \dots \cup V_{(1,19)} \cup \dots \cup V_{(k,1)} \cup \dots \cup V_{(k,19)}\}, \\
N &= \{(X_{(1,1)}, \dots, X_{(1,19)}) \dots, (X_{(k,1)}, \dots, X_{(k,19)})\}, \\
\\
\forall P_j \in P, \forall M, \forall X_{(k,i)} \in N, \\
\text{extensional\_element}(P, M, N) &\Rightarrow P \in N, T_{X_{(k,i)}} = M_{(k,i)}
\end{aligned}$$

where  $P$  represents a network packet variable,  $M$  a set of packets representing all possible values that packet  $P$  can take,  $T$  all the values in  $M$  and  $N$  the translated matrix  $M$ .

All these approaches were experimented using several constraint system solvers. The chosen constraint systems to make the experiments were Gecode 3.1.0, Gecode/R 1.1.0, GNU Prolog 1.3.0-6, SICStus Prolog 4.0.4-x86-linux-glibc2.6 and CaSPER 0.1.4. Each experiment used the same problem with the same set of data where possible.

The problem that was used to analyse the performance of each constraint system solver was the description of a portscan attack. This network situation

---

can be detected by a set of constraints between two network packets, the packet that initiates a TCP/IP connection and the packet that closes it. In order to detect if there is a portscan attack, there is the need to decide how many sets of two of these packets have to appear on a network traffic to consider it a portscan attack. These number of sets of packets used to describe the network situation affects the performance of our approach as each set adds more constraint variables to the problem. With this fact in mind, we made several experiments with the same data sets, but using different number of sets of packets to describe the problem in order to see how the performance was affected when the size of the problem rises.

We also made some runs using different data sets in order to see the behaviour of the performance when the size of the matrix varies and also when the values of the matrix varies.

On some cases we were forced to use smaller matrices with smaller values because some constraint system couldn't handle some matrices while using some constraints.

## 4.2 Modeling a portscan attack as a CSP

A portscan attack can be modeled as a CSP in the form of  $\mathcal{P} = (X, D, C)$ , where  $X$  is a set with all the variables that compose the problem,  $D$  the domain of each variable and  $C$  the constraints of the CSP. The set of variables  $X$  is a set of integer variables representing all the packets described by the CSP.  $X$  can be defined as follows, where  $i$  represents the number of packets that are being looked:

$$X = \{(P_{(1,1)}, \dots, P_{(1,19)}), \dots, (P_{(i,1)}, \dots, P_{(i,19)})\}$$

Follows the definition of the domain  $D$ , where  $D_{(i,n)}$  is the domain of the corresponding variable  $P_{(i,n)}$ ,  $M$  the set of all packets in the network traffic log and  $k$  the number of packets seen on the network traffic log.

$$\begin{aligned} D &= \{D_{(1,1)}, \dots, D_{(1,19)}, \dots, D_{(i,1)}, \dots, D_{(i,19)}\}, \\ M &= \{(V_{(1,1)}, \dots, V_{(1,19)}), \dots, (V_{(k,1)}, \dots, V_{(k,19)})\}, \\ \forall D_{(i,n)} \in D, \quad \forall P_{(i,n)} \in X, \quad P_i &= (P_{(i,1)}, \dots, P_{(i,19)}) \Rightarrow P_i \in M \end{aligned}$$

The constraints  $C$  that compose a such a CSP are defined as follows:

---


$$\begin{aligned}
&\forall i \geq 0 \Rightarrow P_{(i,14)} = 1, \\
&\quad P_{(i,18)} = 0, \\
&\quad P_{(i+1,13)} = 1, \\
&\quad (((P_{(i,2)} = P_{(i+1,2)}) \wedge \dots \wedge (P_{(i,6)} = P_{(i+1,6)})) \vee \\
&\quad ((P_{(i,2)} = P_{(i+1,6)}) \wedge \dots \wedge (P_{(i,6)} = P_{(i+1,11)}))), \\
&\quad ((P_{(i,0)} < P_{(i+1,0)}) \vee ((P_{(i,0)} = P_{(i+1,0)}) \wedge (P_{(i,1)} < P_{(i+1,1)}))), \\
&\quad (((P_{(i,0)} = P_{(i+1,0)}) \wedge (P_{(i+1,1)} - P_{(i,1)} < 500)) \vee \\
&\quad ((P_{(i+1,0)} = P_{(i,0)} + 1) \wedge (1000000 - P_{(i,1)} + P_{(i+1,1)} < 500))), \\
&\quad P \in M
\end{aligned}$$

$$\forall i \geq 2 \Rightarrow ((P_{(i,0)} > P_{(i-2,0)}) \vee ((P_{(i,0)} = P_{(i-2,0)}) \wedge (P_{(i,1)} > P_{(i-2,1)})))$$

## 5 Experimental Results

In this Section we present the execution times obtained on each experiment while using each of the constraint solver as well as a description of the behaviour of the performance of each experiment.

The experiments all use the same problem modeling with the same data set in order to compare the results obtained in each of the constraint solver. The data sets we have used are composed by a set of packets, each one being composed by 19 individual values. For each constraint solver we used 4 matrices, three of them composed by similar values, varying the number of packets in each matrix, and one composed by smaller values. These were the matrices used in most of the constraint solvers, except in Gecode/R and on CaSPER for reasons explained above. With those constraint solvers we used matrices similar to the ones described above, but transformed in order to contain smaller values, still respecting the order that existed between all the values on the original matrix.

For each experiment with each matrix, several runs were made with similar problem modeling but with different number of variables in order to understand the behaviour of the performance of each constraint solver. For each constraint solver two experiments are presented, one using two packet sets and another composed by six packet sets, which allows to analyse the behaviour of the performance of each constraint solver when the number of variables to model the problem varies.

On each of these runs, we also made several measurements of the execution time, being measured the time the system takes to build the constraints, and the time the system needs to solve the problem after it has been initialized.

---

This network monitoring concept works with log files, so this initialization of the constraint solvers can be considered as making part of the process of the log file reading process.

On the experiments with GNU Prolog and SICStus Prolog were made two measurements of the total time to solve the problem, one using no heuristics and another using the *first\_fail*[2] heuristic, where the alternative that most likely leads to a failure is selected.

In Fig. 1 is made a comparison of the time in milliseconds needed to solve the problem modeled by 6 sets of packets varying the size of matrix between Gecode, GNU Prolog and SICStus Prolog, the experiments that used the same matrices. In Fig. 2 is made a comparison of the solve time in milliseconds using the matrix *portscan2* and varying the number of packet sets to model the problem using the same constraint solvers.

All the experiments were run on a dedicated computer, an HP Proliant DL380 G4 with two Intel(R) Xeon(TM) CPU 3.40GHz and with 4 GB of memory, running Debian GNU/Linux 4.0 with Linux kernel version 2.6.18-5.

### 5.1 Using Extensional Constraints

This approach forces a network packet, represented by a tuple of variables, implemented as a list or array of variables to only take values from a set of tuples of values that represent all the packets on a network log, implemented as a matrix.

**Using Gecode** When using the extensional constraint on Gecode, the performance tend to degrade very much when the values on the matrix range from very small values to very high values. The performance also tends to degrade when the number of packet variables are being used to describe the network situation. The size of the matrix that represents all traffic log doesn't seem to affect considerably the performance, since modeling the problem with the same amount or network packet variables with different matrices with a different number of packets doesn't affect its performance. In Table 1 are presented the execution times of the several runs using GNU Prolog. On this experiment were made 10 runs, and the times presented are the *usertime* average of each run in milliseconds.

**Using GNU Prolog with *fd\_relation/2*** Using the equivalent extensional constraint from Gecode on GNU Prolog, *fd\_relation/2*, the performance is very similar to Gecode if the matrix don't have very big values. Also, similar to Gecode, the performance is affected badly when the number of packets used to model the network situation rises. The size of the matrix doesn't seem to affect the performance of GNU Prolog, since the times obtained to the same situation using matrices with different size are practically the



---

same. One big difference to Gecode, is that GNU Prolog doesn't seem to be affected by matrices that are composed by values that ranges from very low to very high values, since the same test using matrices of the same size, one with smaller values and other with much grater values, produces very similar times. Table 2 presents the execution times of the several runs using GNU Prolog. On this experiment 100 runs were made, and the times presented are the *usertime* average of each run in milliseconds.

**Using Gecode/R** The performance of using this approach in Gecode/R is very similar to the performance of Gecode, mean while Gecode/R has a big limitation, as it segfaults when the values of the matrix are too big, so, in order to make the experiments, we were able to use the values of the matrix *portscan\_small* in order to have some kind of comparison. The other matrices *portscan*, *portscan1* and *portscan2* had to be transformed to smaller values in order to Gecode/R be able to work with them. As expected, the behaviour of the performance of Gecode/R using the extensional constraint is quite similar to the behaviour of the performance of Gecode. As the values of the matrix rises, the performance starts to decay. The performance is also affected by the size of the matrix but in a small factor. The number of network packet variables needed to represent the problem has a big influence in the performance, degrading when the number of variables rise. In Table 3 are presented the execution times of the several runs using Gecode/R. On this experiment 10 runs were made, and the times presented are the *usertime* average of each run in milliseconds.

**Using SICStus Prolog** Using SICStus Prolog with the extensional constraint *table/2* revealed to be an interesting solution, presenting good execution times. The performance of SICStus Prolog didn't seem to be affected by the size of the values that compose the matrix. On the other hand, and opposite to the other approaches, the performance is quite affected by the size of the matrix, getting degraded as the its size grows. Similar to all other approaches, the execution times get worst as the size of the problem grows, getting slower when more variables are needed to model the situation. In Table 4 are presented the execution times of several runs using SICStus Prolog. On this experiment 100 runs were made, and the times presented are the *usertime* average of each run in milliseconds.

**Using CaSPER** Using the extensional constraint *table* in CaSPER shows it doesn't handle very good matrices composed by high values, degrading the performance when its values get higher. The performance of CaSPER gets worst when the number of variables needed to represent the problem rise, just as with all other constraint solvers experimented. It also degrades in a great

scale when the size of the matrix rises. CaSPER revealed very sensitive to the size of the values that compose the matrix, having problems handling big matrices composed by big values, so, in order to make some experiments, we transformed the matrices used on other approaches to contain smaller values, being the same that were used on the experiments with Gecode/R. The results obtained with CaSPER are presented in Table 5. On this experiment 10 runs were made, and the times presented are the *usertime* average of each run in milliseconds.

Matrix	N° of Packets	2 sets ( setup )	2 sets ( solve )	6 sets ( setup )	6 sets ( solve )
portscan_small	20	441	48	1281	135
portscan	20	3735	409	10927	1100
portscan1	40	3754	395	10984	1126
portscan2	60	3748	401	11025	1082

**Table 1.** Gecode execution times using extensional constraint

Matrix	N° of Packets	2 sets (setup)	2 sets (solve)	2 sets (solve ff)	6 sets setup	6 sets (solve)	6 sets (solve ff)
portscan_small	20	363.3	2.1	2.2	1075.7	226.5	225.1
portscan	20	366.0	3.0	1.3	1078.1	229.8	227.7
portscan1	40	366.9	4.3	1.6	1079.4	261.7	217.5
portscan2	60	370.08	4.5	1.7	1090.1	262.5	220.1

**Table 2.** GNU Prolog execution times using *fd\_relation extensional* constraint

Matrix	N° of Packets	2 sets (setup)	2 sets (solve)	6 sets (setup)	6 sets (solve)
portscan_small	20	130	492	158	1482
portscan_smaller	20	126	453	156	1380
portscan1_smaller	40	130	460	162	1399
portscan2_smaller	60	133	461	190	1395

**Table 3.** Gecode/R execution times using the *extensional* constraint

---

Matrix	N° of Packets	2 sets (setup)	2 sets (solve)	2 sets (solve ff)	6 sets (setup)	6 sets (solve)	6 sets (solve ff)
portscan_small	20	16.1	0.2	0.2	48.0	1.5	1.6
portscan	20	16.0	0.4	0.3	48.1	1.4	1.6
portscan1	40	33.6	0.4	0.2	100.7	2.7	2.6
portscan2	60	52.3	0.4	0.4	157.4	3.6	3.0

**Table 4.** SICStus Prolog execution using the *extensional* constraints

Matrix	N° of Packets	2 sets (setup)	2 sets (solve)	6 sets (setup)	6 sets (solve)
portscan_small	20	640	52410	1910	123810
portscan_smaller	20	570	51670	1750	121420
portscan1_smaller	40	580	106470	1770	264040
portscan2_smaller	60	590	194080	1800	499440

**Table 5.** CaSPER execution times while using the *extensional* constraint

## 5.2 Using Extensional Constraints with Element Constraints

The approach using *extensional* constraints described in Sect. 5.1 shows that some of the tested constraint solvers don't perform very well when using the extensional constraint with a matrix of possible values composed by values that ranges from very low to very high values. In order to solve this problem, the matrix with all possible values was translated into a matrix that contains indexes to an array that contains all the values of the matrix. This matrix is then used along with the extensional constraint in order to constraint the network packets to the values that exist on the matrix. This packet will have values from the matrix of indexes, so it is needed to create a new packet variable and channel that variable to the values on the array with the original values, which is done using the *element* constraint.

**Using Gecode** Using this approach in Gecode revealed to be quite good, having an excellent performance when compared to the approach that only uses the *extensional* constraint. Although this excellent performance, it behaves much like when using only the extensional constraint, as it degrades as the number of packet variables needed to model a network situation rises. Also, the performance of this approach doesn't seem to be affected by the size of the matrix, since performing the same test using similar matrices, but with different size doesn't change the execution times. One main advantage of this approach, is that its performance is not affected by the values of the matrix, having the same results in terms of performance when using a matrix with very big values or using a matrix with smaller values. In Table 6

---

are presented the execution times of the several runs using Gecode. On this experiment 1000 runs were made, and the times presented are the *usertime* average of each run in milliseconds.

**Using GNU Prolog with *fd\_relation/2*** Using such approach in GNU Prolog didn't reveal to be very promising, since the execution times obtained were actually slower than the version using only the *fd\_relation/2* constraint. This approach tends to degrade just like the approach that used only the constraint *fd\_relation/2*, not being influenced by the values on the matrix or by the size of the matrix, degrading only when the number of packet variables needed to describe a network situation rises. In Table 7 are presented the execution times of the several runs using GNU Prolog.

**Using Gecode/R** Using the extensional constraint combined with the element constraint improved the performance of Gecode/R in a great factor, still, its not as good as the performance obtained by using the same approach with Gecode. Such as Gecode, the performance of Gecode/R is affected in a small factor by the size of the matrix containing all possible values neither by the size of the values on the matrix, still, more noticeable than when using Gecode, since the overall performance of Gecode/R is worse than the overall performance of Gecode. The performance of Gecode/R is affected by the number of the packet variables needed to represent the network situation, degrading when the number of variables rises, just as in Gecode and GNU Prolog. In Table 8 are presented the execution times of the several runs using Gecode/R.

**Using SICStus Prolog** Using SICStus Prolog with the combination of the extensional constraint *table/3* and the constraint element *relation/3* didn't reveal any advantage over the approach that used only the constraint *table/3* just as it happened on GNU Prolog. In Table 9 are presented the execution times of several runs using SICStus Prolog where can be seen the main differences between the other approaches. On this experiment 100 runs were made, and the times presented are the *usertime* average of each run.

**Using CaSPER** Using CaSPER with the *extensional* constraint combined with the *element* revealed very good results when compared to the approach using only the *extensional* constraint. This fact is due to the use of a translated matrix that uses much smaller values than the ones on the original values. The performance of CaSPER tends to degrade when the number of variables needed to model the problem rises as well as when the size of the matrix rises. It seems not to be affected by the size of the values on the original matrix. The execution times of this approach using CaSPER are presented in Table

---

10. On this experiment 10 runs were made and the times presented are the *usertime* average of each run in milliseconds.

Matrix	N° of Packets	2 sets ( setup )	2 sets ( solve )	6 sets ( setup )	6 sets ( solve )
portscan_small	20	4.36	0.46	7.33	1.15
portscan	20	3.86	0.49	6.14	1.24
portscan1	40	4.77	0.57	8.82	1.20
portscan2	60	5.89	0.79	12.15	1.26

**Table 6.** Gecode execution times using *extensional* and *element* constraints

Matrix	N° of Packets	2 sets (setup)	2 sets (solve)	2 sets (solve ff)	6 sets setup	6 sets (solve)	6 sets (solve ff)
portscan_small	20	379.7	3.2	2.8	1127.1	285.6	282.6
portscan	20	381.5	3.4	3.1	1128.7	284.1	281.3
portscan1	40	381.1	3.2	3.1	1132.5	296.3	283.0
portscan2	60	389.1	3.3	3.1	1148.2	294.0	278.8

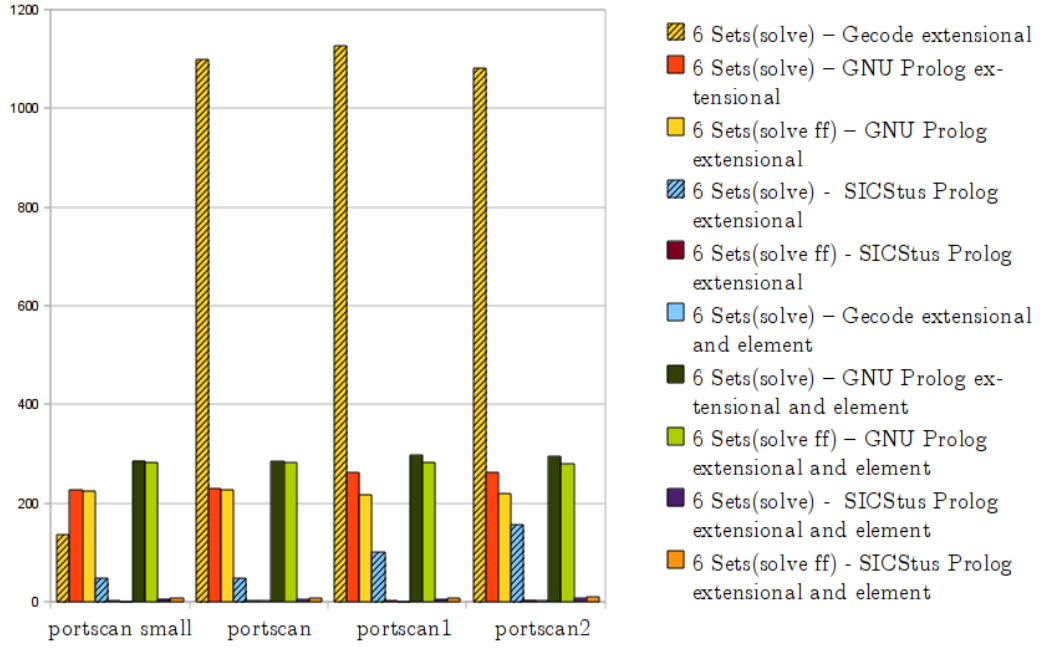
**Table 7.** GNU Prolog execution times using fd\_relation *extensional* and *element* constraint

Matrix	N° of Packets	2 sets ( setup )	2 sets ( solve )	6 sets ( setup )	6 sets ( solve )
portscan_small	20	144	88	235	267
portscan	20	148	82	238	262
portscan1	40	158	95	251	333
portscan2	60	155	121	274	382

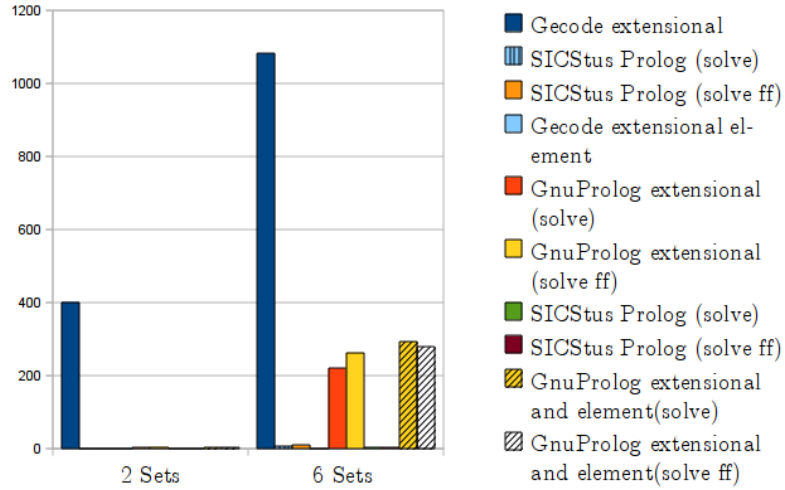
**Table 8.** Gecode/R execution times using the *extensional* and *element* constraints

## 6 Conclusions and Future Work

In this work we have tested several implementations of a network monitoring concept based on constraint programming, using several constraint solver systems in order to analyze the performance of each one.



**Fig. 1.** Solve time comparison in milliseconds, varying the matrix size



**Fig. 2.** Solve time comparison in milliseconds, varying the number of packets to model the problem

---

Matrix	N° of Packets	2 sets (setup)	2 sets (solve)	2 sets (solve ff)	6 sets (setup)	6 sets (solve)	6 sets (solve ff)
portscan_small	20	57.8	0.5	1.8	167.7	5.5	6.8
portscan	20	58.0	0.2	1.0	167.3	4.1	6.5
portscan1	40	58.1	0.4	0.6	167.4	5.1	6.8
portscan2	60	58.5	0.2	0.6	167.6	6.6	9.0

**Table 9.** SICStus Prolog execution using *extensional* and *element* constraints

Matrix	N° of Packets	2 sets (setup)	2 sets (solve)	6 sets (setup)	6 sets (solve)
portscan_small	20	13	54	40	157
portscan_smaller	20	12	56	40	158
portscan1_smaller	40	18	118	58	343
portscan2_smaller	60	24	221	81	647

**Table 10.** CaSPER execution times using the *extensional* and *element* constraints

The results obtained in some constraint solvers are quite impressive, mainly when using Gecode with the *extensional* constraint combined with *element* constraint. With this approach the performance is affected in a very small scale, either by varying the size of the matrix, the size of the values that compose the matrix or by number or variables used to model the problem. As expected Gecode/R is slower than Gecode, as a new layer of modeling is being added on top of Gecode.

While Gecode and Gecode/R have a huge gain of performance by combining the *Extensional* and *Element* constraint, implementing this concept in GNU Prolog or SICStus Prolog actually degrades the performance of the system. An interesting conclusion for the several experiment, is that on such constraint solvers, the performance is not affected by the values that compose the matrices. Also on GNU Prolog, an interesting fact is that the number of variables used to model the problem influences the performance on a very small scale.

The results obtained by Gecode shows that modeling network situations with constraints in order to perform network monitoring and intrusion detection tasks can be done in an efficient way by combining *extensional* and *element* constraints.

This work shows that the performance of each approach varies in a big scale, ranging from very fast execution times when using Gecode with the combination of the *extensional* and *element* constraints to a much lower performance when using an *extensional* constraint with CaSPER. We haven't yet identified a valid

---

explanation for the different performance, as the objective of this work was to experiment different approaches on several constraint solvers. An important future step will be the study in detail of the reasons that lead to such a performance difference in order to optimize the performance of this concept.

We are currently implementing new network constraints as well as making use of new data types in order to allow for the description of more network situations. As the modeling and performance aspects progress, we will extend the application domain to include live network monitoring. Another significant planned addition is the development of a Domain Specific Language (DSL)[10] that will use these network constraints to conveniently perform network monitoring tasks.

Work is also being carried out in our research group, in the development of a new parallel constraint programming system, focused on solving constraints in a distributed environment. Naturally we plan on extending the work presented herein to make use of the new constraint solving library.

## References

1. F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier Science, 2006.
2. K.R. Apt. *Principles of constraint programming*. Cambridge Univ Pr, 2003.
3. Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.
4. D. Diaz and P. Codognet. Design and implementation of the gnu prolog system. *Journal of Functional and Logic Programming*, 6(2001):542, 2001.
5. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. *Lecture notes in computer science*, pages 191–206, 1997.
6. C. Schulte and P.J. Stuckey. Speeding up constraint propagation. *Lecture Notes in Computer Science*, 3258:619–633, 2004.
7. Gecode/R Team. Gecode/R: Constraint Programming in Ruby. Available from <http://gecoder.org/>.
8. M. Correia and P. Barahona. Overview of the CaSPER\* Constraint Solvers. *Third International CSP Solver Competition*, page 15, 2008.
9. tcpdump web page at <http://www.tcpdump.org/>, April, 2009.
10. A. Van Deursen and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.



---

# Searching in Protein State Space

Dietmar Seipel

Department of Computer Science

seipel@informatik.uni-wuerzburg.de

Jörg Schultz

Department of Bioinformatics, Biozentrum

joerg.schultz@biozentrum.uni-wuerzburg.de

University of Würzburg, Am Hubland, D-97074 Würzburg, Germany

**Abstract.** The increasing complexity of protein interaction networks makes their manual analysis infeasible. Signal transduction processes pose a specific challenge, as each protein can perform different functions, depending on its state.

Here, we present a PROLOG and XML based system which explores the protein state space. Starting with state based information about the function of single proteins, the system searches for all biologically reasonable states that can be reached from the starting point. As facts of general molecular biology have been integrated, novel reasonable states, not encoded in the starting set, can be reached. Furthermore, the influence of modifications like mutations or the addition of further proteins can be explored. Thus, the system could direct experiments and allow to predict their outcome.

## 1 Introduction

Current large scale projects like the sequencing of genomes and the unravelling of protein *interaction networks* produce a wealth of data and give new insights into the molecular architecture of cells. Still, it is a huge step from data generation to the understanding of the molecular details of cellular networks. One of the primary challenges is the structured representation of a proteins function, a prerequisite for any computational analysis. Although many protein and nucleotide databases still use mainly human readable text as annotation, different approaches have been developed to structure this knowledge. These were as straightforward as defining specific keywords or as complicated as setting up an *ontology* for different aspects of protein function [Rzhetsky *et al.*, 2000]; probably, the most widely used of the latter approaches is from the Gene Ontology project [Consortium, 2008].

An additional level of complexity arises when describing not only the function of single proteins, but their interplay within the cellular network. Here, the function of one protein has to be seen in the context of its interactions or the pathways it is involved in. In the case of metabolic networks, methods for the representation of knowledge and the computation with function have been developed [Keseler *et al.*, 2009, Karp, 2001]. Here, the function of a protein can be seen as more or less constant. This contrasts typical eukaryotic signaling pathways, where the function of a protein and therefore

---

its influence on the cellular networks changes for different input signals. This flexibility in function is usually implemented by describing the state of a protein when performing specific actions [Duan *et al.*, 2002, Schacherer *et al.*, 2001, Ratsch *et al.*, 2003]. Systems like the  $\pi$ -calculus have been developed, which allow to encode signaling pathways and to explore the effects of *mutations* [Regev *et al.*, 2001]. Still, these approaches as well as Petri nets [Grafahrend-Belau *et al.*, 2008] represent rigid networks and do not allow to find states not given to the system. The Biocham system [Fages *et al.*, 2004] offers automated reasoning tools for querying the temporal properties of interaction networks under all possible behaviours using Computation Tree Logic (CTL). [Baral, 2004] uses an action-style, knowledge based approach for supporting various tasks of reasoning (including planning, hypothetical reasoning, and explanation) about signaling networks in the presence of incomplete and partial information.

Here, we describe a *simulation approach* which is based on a logic representation of protein states. Its goal is to find all possible and biological acceptable pathways which can be derived from the given starting situation. As the system is aware of general concepts of molecular biology, it can 1) generate states not known to the system and 2) simulate the outcome of modifications like mutations or addition of inhibitors.

The rest of this paper is organized as follows: In Section 2, we present the concepts and techniques for manipulating XML structures in PROLOG. Section 3 provides formal representations of protein function, general biological concepts, and then it describes the central PROLOG search algorithm for protein networks. Two practical applications are shown in Section 4. Finally, Section 5 discusses the new approach.

## 2 Manipulation of XML Structures in PROLOG

The logic programming language PROLOG is very useful for representing, manipulating and reasoning about *complex structures*, such as the states obtained during the simulation of biological processes [Bratko, 2001, Clocksin & Mellish, 2003]. In particular, *search algorithms* can be implemented very nicely based on PROLOG's built-in concept of backtracking, and the handling and manipulation of the relatively complex intermediate data structures obtained during the simulation can be done in a very declarative way.

XML has been increasingly used for representing various forms of complex, semi-structured data, and for exchanging these data between different applications. E.g., in molecular biology, XML languages like SBML [Hucka *et al.*, 2003], MAGE-ML [Spellman *et al.*, 2002] and PSI MI [Hermjakob *et al.*, 2004] have been developed. XML data are used by many tools in different application domains, and to some extent they are human-readable.

Thus, we have used XML as the data format for the complex states of our PROLOG based search algorithm. For querying and manipulating XML elements in PROLOG, we have developed the XML query, transformation, and update language FNQUERY [Seipel, 2002]. This combination of XML and PROLOG facilitates knowledge representation and software engineering drastically. The elegant data access to and update of XML data by path expressions in connection with the powerful reasoning facilities and the declarative programming style of PROLOG makes our application compact, flexible,

---

and extensible. Another PROLOG based XML query language has, e.g., been proposed in [Alemendros-Jiménez *et al.*, 2007], without the necessary update functionality, however. Our approach is also superior to standard XML processing tools such as XQuery or XSLT, since it is fully interleaved with the declarative – and general purpose – programming language PROLOG, and since the update capabilities are better.

## 2.1 XML Representations and Path Expressions

There exist various ways of representing XML structures in PROLOG. We have developed a PROLOG term representation for XML elements, which we call field notation, and a fact representation, which we call graph notation. These internal representations are described in [Seipel, 2002]; for the purposes of this paper it is sufficient to read XML data from files and to write them to XML files.

We use path expressions for localizing sub-structures of an XML structure. The path language FNPATH, which we have developed, is an extension of the well-known path language XPATH, and XPATH expressions can be mapped to equivalent FNPATH expressions. Moreover, we can allow for more general path expressions, since we use PROLOG – including backtracking and unification – for their evaluation. In the following we will describe the fragment of FNPATH which we have used in this paper. The path expressions in this fragment directly correspond to equivalent XPATH expressions.

A path expression  $\pi = \sigma_1 \sigma_2 \dots \sigma_n$  consists of one or more steps  $\sigma_i$  for localizing child elements or attributes of XML elements. E.g., the path expression

`/proteins/protein::[@name=Protein]`

has two location steps. When this path expression is applied to an XML element *Cell*, then the first step  $\sigma_1 = /proteins$  selects the child element *Ps* of *Cell* with the tag *proteins*. Subsequently, the second step  $\sigma_2 = /protein::[@name=Protein]$  of the path expression selects a child element *P* of *Ps* with the tag *protein*, such that *P* has an attribute *name* with the value *Protein*.

In general, a location step can be of the forms  $\sigma = @ \nu$ ,  $\sigma = / \nu$ , and  $\sigma = / \nu :: \tau$ , where  $\nu$  is a node test and  $\tau$  is a list of predicates. The second form is a short hand for the third, if the list  $\tau$  is empty. The predicate *@name=Protein* in the example above also contains a location step *@name*, which selects the value of the attribute *name* of the element *P*; the predicate succeeds, if this value is *Protein*.

## 2.2 The Query Language

The PROLOG based XML query, transformation, and update language FNQUERY is based on FNPATH, and it has three sublanguages:

- FNSELECT: selection of subelements/attributes,
- FNTRANSFORM: transformations like XSLT,
- FNUPDATE: updates (insertion, deletion) of elements/attributes.

In this paper, we have extensively used FNSELECT and FNUPDATE for computing successor states during the simulation process. For this purpose, FNUPDATE has been extended by insertion and deletion operations, which are very useful here.

---

**Selection of Substructures.** If  $X$  is an XML element in PROLOG representation and  $\pi$  is a path expression, then the assignment  $Y := X \pi$  selects a substructure of  $X$  and assigns it to  $Y$ . E.g., the following rule selects the location `Loc` of a state element for a given protein with the name `Protein` within a given cell `Cell`. The path expression  $\pi$  consists of 5 location steps, which are applied successively to `Cell`. The predicate `@name=Protein` in the second location step assures that the selected child has the proper value for the attribute `name`:

```
get_location(Protein, Cell, Loc) :-
    Loc := Cell/proteins/protein::[@name=Protein]
        /current_state/state@location.
```

**Modification/Update of Substructures.** If  $X$  is an XML element in PROLOG representation and  $\pi$  is a path expression, then the assignment  $Y := X * [\pi : v]$  modifies the substructure of  $X$  that is selected by  $\pi$  to the new value  $v$ . E.g., the assignment statement in the following rule changes the value of the attribute `location` to `Loc`.

```
set_location(Protein, Loc, Cell_1, Cell_2) :-
    Cell_2 := Cell_1 * [
        /proteins/protein::[@name=Protein]
        /current_state/state@location:Loc].
```

**Insertion and Deletion of Substructures.** The following PROLOG rules are used for modifying a given cell `Cell_1`. The first rule adds a new interaction element with the attribute value `Int` for `protein` within the `protein` element with the name `P`. The second rule deletes such an interaction element.

```
add_interaction(P, Int, Cell_1, Cell_2) :-
    Cell_2 := Cell_1 <+> [
        /proteins/protein::[@name=P]
        /current_state/state/interactions
        /interaction:[protein:Int]:[] ].

delete_interaction(P, Int, Cell_1, Cell_2) :-
    Cell_2 := Cell_1 <-> [
        /proteins/protein::[@name=P]
        /current_state/state/interactions
        /interaction::[@protein=Int] ].
```

### 3 Representation and Searching

In the following, we provide formal representations of protein function and general biological concepts, and then we describe the central PROLOG search algorithm for protein networks.

---

### 3.1 Formal Representation of Protein Function

It is a fundamental feature of proteins, especially those involved in signal transduction, that their function is tightly regulated. Depending, for example, on its own phosphorylation status, a protein kinase might phosphorylate different target proteins. Therefore, any representation of a proteins function has to allow different functions for a single protein, depending on its status. This status can be described by three features, namely the localisation, modifications, and interactions [Duan *et al.*, 2002, Ratsch *et al.*, 2003].

In our approach, we combine these features with a detailed description of the function performed within this state. Currently, the *localisation* mainly describes the organelle a protein can be found in, for example the cytoplasm or the nucleus. Within the list of *modifications*, the type, but also the position in the sequence is stored. Finally, the *interactions* describe a list of bound proteins. Depending on the value of these three features, the protein might perform different functions. These are encoded in a list of actions. Each action itself consists of the type of the action, the name of the involved protein and possible parameters.

Type	Subtypes	Parameters
moving	none	protein new location
interaction	binding dissociation	protein <sub>1,2</sub>
modification	phosphorylation dephosphorylation	protein <sub>1,2</sub> position

**Table 1.** Implemented Actions

Currently, three general types of actions are implemented, where each type can contain different subtypes and parameters (Table 1). Within this concept it is important to note that we do not use any descriptions like *activation*, *stimulation* or *inhibition*, as these are interpretations and depend on the experiment performed. We describe the molecular details of a function a protein performs when in a given state. This might be interpreted as an activation to perform a function within a given state.

Having defined the state as the core structure, different states and their actions have to be put together with additional information to describe a proteins function. First, there are constant, that is state independent, features which have to be represented. Second, as we aim for qualitative simulation, the actual state of a protein has to be stored. Following, all states which trigger a specific action are listed. Finally, we allow for so called forbidden states. Especially in molecular biology, the knowledge of a proteins function frequently contains negative information like the fact that a protein will not be localised within the nucleus if bound to a defined other protein. By introducing forbidden states, we allow the handling of this information. Figure 1 shows the XML structure describing the function of a single protein.

---

```

<protein name="receptor" sequence="MDCQLSILLLLSCSVLD...">
  <current_state>
    <state location="transmembrane">
      <modifications/>
      <interactions>
        <interaction protein="jak"/> </interactions>
      <actions/>
    </state>
  </current_state>
  <action_states>
    <state location="transmembrane">
      <modifications/>
      <interactions mode="identical">
        <interaction protein="ligand"/>
        <interaction protein="jak"/> </interactions>
      <actions>
        <action type="interaction" subtype="binding">
          <protein name="receptor"/>
          <parameter protein="receptor"/>
          <status state="todo"/> </action> </actions>
        </state>
      </action_states>
    <forbidden_states>
      <state location="transmembrane">
        <modifications/>
        <interactions>
          <interaction protein="Grb2"/> </interactions>
        </state>
      </forbidden_states>
    </protein>

```

**Fig. 1.** XML Representation of a Proteins Function – To represent all functions of a protein and allow simulation, the actual state of a protein (`current_state`) as well as all states leading to an action (`action_states`) have to be known. Furthermore, states not acceptable for the protein can be stored (`forbidden_states`). Features invariant to the protein, like the name or its sequence, are given as parameter to the protein tag.

### 3.2 Representation of General Biological Concepts

If one aims at qualitatively simulating signal transduction processes, it is of vital importance to take into account general biological concepts which can help to restrain the search space and keep the derived states consistent as well as biologically correct. Therefore, we have implemented the following concepts into our system.

**Localisation.** To describe different cellular localisations, we first have defined a set of accepted keywords describing localisations. In a second step, we have built, comparably to the GeneOntology, a hierarchy of these localisations using `subclasses/2` facts.

---

As we are modelling changes of localisations (moves), we furthermore define between which compartments direct moves are possible using `allowed/2` facts. For example, although nucleus and mitochondrion are both organelles embedded in the cytoplasm, no direct transitions between them are possible. A protein is assigned to one localisation for each state. A special case are transmembrane proteins, which reside in up to three different localisations, for example extracellular, membrane and cytoplasmic; this is represented by a `contains/2` fact.

---

```
localisations([
    localisation, intra, extra,
    membrane, transmembrane,
    cytoplasm, nucleus, ... ]).

subclasses(localisation, [
    intra, extra, membrane ]).
subclasses(transmembrane, [membrane]).

allowed(intra, extra).
allowed(cytoplasm, nucleus).
allowed(X, X).

contains(transmembrane, [
    extra, membrane, cytoplasm ]).

...
```

---

**Interaction.** As protein interactions build the backbone of any signal transduction process, a substantial amount of rules was generated to assure, that the modelled interactions are biologically sound. First, only proteins not already interacting (`unbound/2`) are allowed to start a new interaction and only interacting proteins can dissociate. Second, the predicate `matching_location_/3` assures that both interacting proteins reside in the same localisation; the values of the localisations do not have to be identical. Third, reciprocity of binding as well as dissociation events is assured by `add_interaction/3`. If one of multiple interacting proteins is moving, that is its localisation changes, all interacting proteins have to move. This assures consistency of localisation information; furthermore a translocation will not happen if any of the interacting proteins is not allowed within the new localisation. E.g., the PROLOG module `interaction` contains the following rule for bindings:

---

```
binding(S1, Action, S2) :-
    P1 := Action/protein@name,
    P2 := Action/parameter@protein,
    Proteins = [P1, P2],
    unbound(S1, Proteins),
```

---

---

```

matching_location_(S1, Proteins, SA),
add_interaction(SA, Proteins, SB),
!,
\+ forbidden_state(Proteins, SB),
add(SB, Action, Proteins, S2).

```

---

Based on the facts implemented for localisation, we can handle cases where the information about the localisation of one protein is more specific than the other. In an example scenario, it might be known for protein A, that it is localised within the cytoplasm whereas protein B is only known to be somewhere within the cell (intracellular). Our system does not only allow this interaction, as the cytoplasm is a specification of intracellular, it updates the localisation information for protein B. This is also done iteratively for all interacting proteins in the case of complexes. Thereby, the system can improve on current knowledge.

**Modification.** Currently, the only implemented modification is phosphorylation. Here, our rules assure that within the phosphorylated position resides a correct amino acid, S, T or Y. Furthermore, it is assured, that the side which becomes phosphorylated is not already phosphorylated. Complementarily, only phosphorylated sites can become de-phosphorylated. In both cases, a modification includes the interaction (predicate `interaction:unbound/2` and first call of predicate `interaction/4`), the actual modification (predicate `phosphorylation_/3`) and a dissociation (second call of predicate `interaction/4`). Therefore, only proteins which are able to interact can be involved in a phosphorylation or de-phosphorylation reaction.

---

```

phosphorylation(S1, Action, S2) :-
    Protein_1 := Action/protein@name,
    Protein_2 := Action/parameter@protein,
    Proteins = [Protein_1, Protein_2],
    interaction:unbound(S1, Proteins),
    interaction(S1, Proteins, binding, SA),
    phosphorylation_(SA, Action, SB),
    interaction(SB, Proteins, dissociate, S2).

```

---

### 3.3 Searching Through Protein States

Thus far, we have described a static system for the representation of protein function based on states and some general rules concerning biological concepts. As it is the aim of the system to search through protein state space and to allow for qualitative simulations, the function of each protein has to be connected to and depend on the state of other proteins. This has been implemented for example by LiveDIP [Duan *et al.*, 2002] and the Molecule Pages database [Saunders *et al.*, 2008] by linking states with state transitions. Here, no state transitions are encoded. That is, no information like *protein*



---

A gets transferred from state 1 to state 2 when interacting with protein C has to be provided. Rather, we perform actions on proteins and let the protein itself *decide* whether its new state might trigger further actions. Thereby, we allow each protein to reach states, which are not explicitly encoded in its state list and are able to unravel novel pathways intrinsic to the analyzed protein interaction network.

**Comparing States.** A fundamental concept of the presented approach is the ability to compare different states. This is, for example, of importance for deciding, whether the current state of a protein *matches* a state which is connected to a new action. Obviously, the most straightforward approach would be to check for identity within all features of the state (localisation, interactions and modifications). This would lead to the same results as explicitly coding state transitions and would not allow to detect states not encoded within the system. To allow for a more explorative searching, we have defined a *subsumption* relationship for states.

```
subsumes_localisation(S, G) :-
    SLoc := S@location,
    GLoc := G@location,
    localisation:specifies(SLoc, GLoc) .

subsumes_interactions(S, G) :-
    identical := G/interactions@mode,
    !,
    get_interactions(S, SInt),
    get_interactions(G, GInt),
    subset(GInt, SInt),
    subset(SInt, GInt) .

subsumes_modifications(S, G) :-
    identical := G/modifications@mode,
    !,
    get_modifications(S, SMod),
    get_modifications(G, GMod),
    subset(GMod, SMod), subset(SMod, GMod) .
```

In the case of localisations, the subsuming state needs to have a localisation which is either identical to or more specific than the general state. For example, a state with a cytoplasmic localisation will subsume a state with a intracellular localisation. Here, the general biological knowledge about the compartments of the cell is substantial. For the interactions and modifications, the general state has to be a subset of the subsuming state. To allow more restrictive comparisons, both interaction and modification can be set to identical, which enforces identity between the general and the subsuming state. These rules allow to decide whether a protein is in a state, which can trigger a specified action.

---

**Search Algorithm and Implementation.** Having defined a subsumption relationship between states and actions associated with given states, which are called *soups* below, the system can now be used to search through protein state space. To start the search process, an action as well as the starting states of all proteins are needed. In the first step, the system tries to perform the action, taking into account the biological background given above. If the action can be performed, at least one protein will be in a novel state. The system now compares the current state of all proteins with all action states of the protein. If no current state is found which subsumes an action state (`soup_node_subsumes/2`), then the system tries to perform the action. If successful, a new round of search is initiated (`find_next_soup/4`). In each step, applications of the general biological concepts assure, that only correct states are reached. For example, a protein A might be in a state triggering an action to bind to protein B. This action will fail, if protein B in its current state is located in a different cellular compartment than protein A. The search process iterates until no protein is in a state triggering a new action. Here, the first solution of the search process has been found.

```
protein_state_search(S, _, S).
protein_state_search(S1, Actions_1, S3) :-
    M := S1@id, not(soup_node_subsumes(_, M)),
    find_next_soup(S1, Actions_1, S2, Actions),
    find_candidate_actions(S2, Actions, Cs),
    add_candidate_actions(Cs, Actions, Actions_2).
protein_state_search(S2, Actions_2, S3).

find_next_soup(S1, As1, S2, As2) :-
    A := As1/action::[@id=Id, /status@state=todo],
    [Type, SubType] := A-[@type, @subtype],
    once(apply(Type:SubType, [S1, A, S])),
    As2 := As1 * [
        /action::[@id=Id]/status@state:done ],
    remember_search_edge(S1, S, S2).
```

Natural numbers are assigned as identifiers to Soups, and the edges between two soups are asserted in the PROLOG database:

```
remember_search_edge(S1, S, S2) :-
    M := S1@id,
    soup_to_number_(S, N),
    S2 := S * [@id:N],
    assert(soup_edge(M, N)),
    !.
```

The system finds all solutions using PROLOG's *backtracking* mechanism. In each intermediate step, the system checks, whether another action than the one already performed might be triggered. If so, another search process is started, leading to further

---

solutions of the system. Thus, a *depth first search* through the space of all possible protein states is performed.

Within the search process, care is taken to detect possible cycles which would lead to infinite searches. Therefore, the search is stopped if an action already performed is triggered.

```
find_candidate_actions(Soup, Actions, Candidates) :-
    findall( Action,
        ( Protein := Soup/proteins/protein,
          check_action(Protein, Action),
          \+ performed(Action, Soup),
          \+ action_already_found(Actions, Action) ),
        Candidates_2 ),
    sort(Candidates_2, Candidates).

check_action(Protein, Action) :-
    C_State := Protein/current_state/state,
    A_State := Protein/action_states/state,
    protein_state_subsumes(C_State, A_State),
    Action := A_State/actions/action.

performed(Action, Soup) :-
    Test_Action := Soup/pathway/step/action,
    action:identical(Action, Test_Action).

action_already_found(Actions, Action) :-
    Action_2 := Action * [@id:_],
    Action_2 := Actions/action.
```

**Interface / Visualization.** Already in comparatively small biological systems, many end states can be reached. Obviously, any two of these end states can be identical, as each can be reached by different combinations of actions. To allow for the evaluation of the results and decrease the redundancy, we have developed a visualization in SWI-PROLOG [Wielemaker, 2009], cf. Figures 2 and 3. Here, each combination of protein states, which can be reached is represented by a circle. If two states are identical, then the nodes representing these states are combined leading to a collapse of the original search tree.

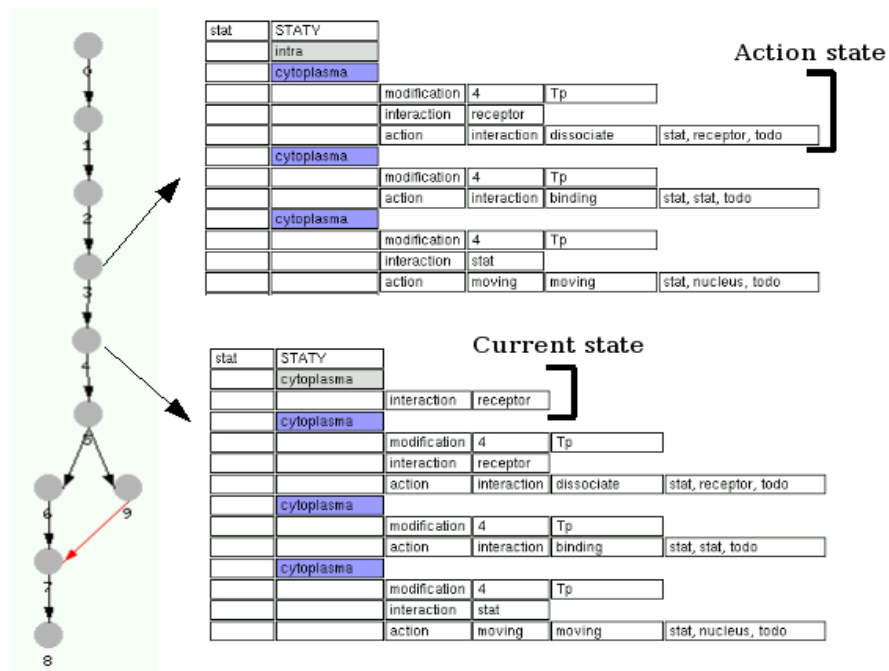
For each node, not only the state of all proteins, but also all pathways leading to the node are given. As the representation is based on the same XML structure as the input, it might be used as input for other programs, e.g. visualizing the pathways in a more detailed way.

## 4 Applications

To show, that the proposed system is indeed able to represent signaling pathways, we evaluate two test cases, the Jak–Stat and the MAP kinase pathway. In addition to finding novel states, it was a fundamental goal of the presented system to allow for *qualitative simulations* of modifications of the signal transduction pathways; here, the implemented general biological concepts are of major importance.

### 4.1 Jak–Stat Pathway

The Jak–Stat signalling pathway provides one of the fastest routes from a cell surface receptor to the nucleus [O’Shea *et al.*, 2002]. Still, all major principles of signalling pathways, dimerization, phosphorylation and translocation are implemented within this short pathway. The involved proteins are the receptor, a ligand, the Jak protein kinases and the Stats. For each of the proteins we have defined a starting state as well as states which lead to further actions. An example is the rule that a phosphorylated Stat can homodimerize. The state searching procedure was started with an interaction of the ligand with the receptor.



**Fig. 2.** Representation of the Jak–Stat Pathway

---

Figure 2 shows all soups (combinations of proteins states) which have been reached from this initial condition. Most interestingly, there are two ways how node 7 can be reached. Inspection of this node reveals, that the difference between the two pathways is the order in which the dissociation of Stat from the receptor and the dimerization of two Stats are handled.

- With the knowledge given to system, two pathways have been found leading to node 7. Whereas in the first (via node 6), the phosphorylated Stat dissociates from the receptor before dimerization with another phosphorylated Stat, the pathway via node 9 reverses these steps.
- The detailed information underlying nodes 3 and 4 is selected and shown in an HTML table. The annotation is automatically improved. After binding to the receptor, the localisation of Stat is specified from *intracellular* to *cytoplasmic* (shown in grey).

It is an important feature of our system, that states which are not explicitly given in the *knowledge base* can be generated. This happens for example at node 9, which describes a protein complex of the ligand, a dimerized receptor, Jak and Stat. Additionally, Stat itself is a dimer. The existence of such a protein complex was not given within any state described in the knowledge base.

In addition to the generation of novel states, the system can also help in refining current knowledge. At the beginning of the search process, Stats location is annotated as *intracellular*. Within node 4, Stat binds to the receptor, which is annotated as *trans-membrane* protein. As in the current implementation transmembrane proteins consist of an extracellular, a transmembrane and a cytoplasmic region, the system deduces, that Stat has to be cytoplasmic, if it binds as an intracellular protein to a transmembrane protein.

This example illustrates, that information about important states is not only sufficient to reconstruct pathways, also novel states as well as improvements of current knowledge can be automatically generated.

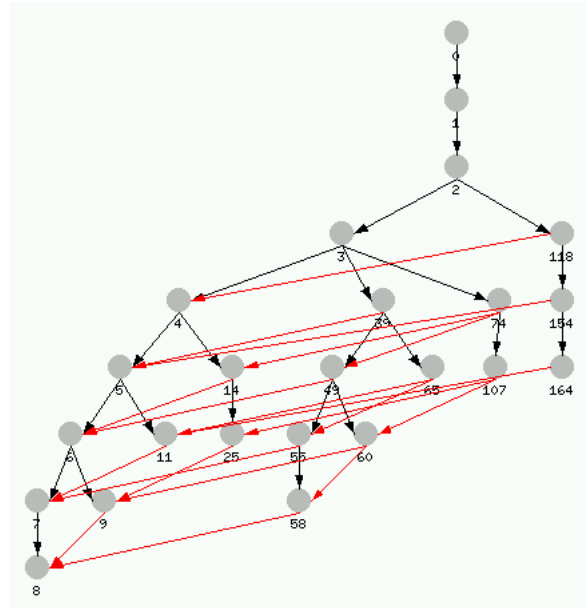
## 4.2 MAP Kinase Pathway

The MAPK pathway represents one of the oldest signalling cascades conserved between yeast and human [Bardwell, 2005, Wang & Dohlman, 2004]. Its core is a cascade of phosphorylation of protein kinases.

In total, 167 possible soups (combinations of protein states) were found. As the integration of these states and the pathways leading to them shows (Figure 3), there are many different pathways leading to identical combinations of protein states. Directed by this analysis, experimental research might unravel which of these pathways are actually implemented by the yeast cell.

## 5 Discussion

Unraveling the molecular details of signal transduction processes is an important step towards understanding the regulation of cellular networks. Although it is the ultimate



**Fig. 3.** Representation of the Yeast Mating Pheromone Response Pathway

goal of systems biology to quantitatively model the whole cell, especially in signal transduction we are still far from the enumeration of all involved molecular reactions. The identification of these reaction as well as the delineation of possible pathways is therefore a crucial prerequisite.

We have described a PROLOG system which, given a defined set of proteins and their functions, searches for all possible states encoded within a system and thereby finds all possible pathways using backtracking. The fundamental concept of the approach is the state of a protein, which is defined by its location, modifications and interactions. The whole system consists of three components: first an XML encoded state based representation of protein function. Second, a rule based representation of general biological concepts and third, a search algorithm implemented in PROLOG. The resulting system is not only able to search through protein state space, but furthermore to simulate the outcome of manipulations to the proteins. Declarative logic programming in PROLOG greatly simplified the system despite the *complex logic* behind the search algorithm and the complexity of the underlying *data structures*.

Contrasting other approaches [Saunders *et al.*, 2008,Zheng *et al.*, 2008], we do not encode any explicit state transitions. Instead, the protein *knows* about states which trigger further actions. If the protein is in a state that subsumes one of these states, an action is started, which might transform other proteins, leading to further actions. Thus, proteins can reach states, which have not been given to the system beforehand. These novel deduced states might be a good starting point for further experimental exploration of signal transduction networks.

---

A critical point within the approach is the collection of the functions of the involved proteins. Although manual *knowledge acquisition* will assure the highest quality, it is comparably slow. One might overcome this hindrance by adding automated function prediction for the involved proteins based for example on sequence analysis.

So far, the system has been tested only on smaller signal transduction networks. One possible challenge might be the increasing computational resources needed for larger system. To date, we rather see the collection of biological knowledge in the level of detail needed for the system as the limitation. More on the users point of view, the huge amount of detail might become hard to digest in the current representation. Thus, a more *graphical*, possibly *animated interface*, which might even allow to interfere with the system, would be the long term goal.

## References

- [Almendros–Jiménez *et al.*, 2007] Almendros–Jiménez, J.M., Becerra–Terón, A., Enciso–Baños, F.J. (2007) Integrating XQUERY and Logic Programming. *Proc. 17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007) and 21st Workshop on (Constraint) Logic Programming (WLP 2007)*, 136–147.
- [Baral, 2004] Baral, C., Chancellor, K., Tran, N., Tran, N.L., Joy, A., Berens, M. (2004) A knowledge based approach for representing and reasoning about signaling networks, *Bioinformatics*, 20 (Suppl. 1), i15–i22.
- [Bardwell, 2005] Bardwell, L. (2005) A walk-through of the yeast mating pheromone response pathway. *Peptides*, 26 (2), 339–350.
- [Bratko, 2001] Bratko, I. (2001) PROLOG–Programming for Artificial Intelligence, 3rd Edition, Addison–Wesley, 2001.
- [Clocksin & Mellish, 2003] Clocksin, W. F., Mellish, C. S. (2003) Programming in PROLOG. 5th Edition, Springer, 2003.
- [Consortium, 2008] Consortium, G. O. (2008) The Gene Ontology project in 2008. *Nucleic Acids Res.*, 36, D440–444.
- [Duan *et al.*, 2002] Duan, X. J., Xenarios, I. & Eisenberg, D. (2002) Describing biological protein interactions in terms of protein states and state transitions: the LiveDIP database. *Mol Cell Proteomics*, 1 (2), 104–116.
- [Fages *et al.*, 2004] Fages, F., Soliman, S., Chabrier, N. (2004) Modelling and querying interaction networks in the biochemical abstract machine BIOCHAM, *Journal of Biological Physics and Chemistry*, 4, 64–73.
- [Grafahrend–Belau *et al.*, 2008] Grafahrend–Belau, E., Schreiber, F., Heiner, M., Sackmann, A., Junker, B. H., Grunwald, S., Speer, A., Winder, K. & Koch, I. (2008) Modularization of biochemical networks based on classification of Petri net t–invariants. *BMC Bioinformatics*, 9, 90.
- [Hermjakob *et al.*, 2004] Hermjakob, H., Montecchi–Palazzi, L., Bader, G., Wojcik, J., Salwinski, L., Ceol, A., Moore, S., Orchard, S., Sarkans, U., von Mering, C., Roechert, B., Poux, S., Jung, E., Mersch, H., Kersey, P., Lappe, M., Li, Y., Zeng, R., Rana, D., Nikolski, M., Husi, H., Brun, C., Shanker, K., Grant, S. G. N., Sander, C., Bork, P., Zhu, W., Pandey, A., Brazma, A., Jacq, B., Vidal, M., Sherman, D., Legrain, P., Cesareni, G., Xenarios, I., Eisenberg, D., Steipe, B., Hogue, C. & Apweiler, R. (2004) The HUPO PSI’s molecular interaction format – a community standard for the representation of protein interaction data. *Nat. Biotechnology*, 22 (2), 177–183.

- 
- [Hucka *et al.*, 2003] Hucka, M., Finney, A., Sauro, H., Bolouri, H., Doyle, J., Kitano, H., Arkin, A., Bornstein, B., Bray, D., Cornish-Bowden, A., Cuellar, A., Dronov, S., Gilles, E., Ginkel, M., Gor, V., Goryanin, I., Hedley, W., Hodgman, T., Hofmeyr, J.-H., Hunter, P., Juty, N., Kasberger, J., Kremling, A., Kummer, U., NovÁlre, N. L., Loew, L., Lucio, D., Mendes, P., Minch, E., Mjolsness, E., Nakayama, Y., Nelson, M., Nielsen, P., Sakurada, T., Schaff, J., Shapiro, B., Shimizu, T., Spence, H., Stelling, J., Takahashi, K., Tomita, M., Wagner, J., Wang, J. & Forum, S. (2003) The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19 (4), 524–531.
- [Karp, 2001] Karp, P. (2001) Pathway databases: a case study in computational symbolic theories. *Science*, 293 (5537), 2040–2044.
- [Keseler *et al.*, 2009] Keseler, I. M., Bonavides-Martínez, C., Collado-Vides, J., Gama-Castro, S., Gunsalus, R. P., Johnson, D. A., Krummenacker, M., Nolan, L. M., Paley, S., Paulsen, I. T., Peralta-Gil, M., Santos-Zavaleta, A., Shearer, A. G. & Karp, P. D. (2009) EcoCyc: a comprehensive view of *Escherichia coli* biology. *Nucleic Acids Research*, 37, D464–470.
- [O’Shea *et al.*, 2002] O’Shea, J. J., Gadina, M. & Schreiber, R. D. (2002) Cytokine signaling in 2002: new surprises in the Jak/Stat pathway. *Cell*, 109 Suppl., S121–131.
- [Ratsch *et al.*, 2003] Ratsch, E., Schultz, J., Saric, J., Lavin, P., Wittig, U., Reyle, U. & Rojas, I. (2003) Developing a protein–interactions ontology. *Comparative and Functional Genomics*, 4, 85–89.
- [Regev *et al.*, 2001] Regev, A., Silverman, W. & Shapiro, E. (2001) Representation and simulation of biochemical processes using the  $\pi$ -calculus process algebra. In *Pacific Symposium on Biocomputing*, 459–470.
- [Rzhetsky *et al.*, 2000] Rzhetsky, A., Koike, T., Kalachikov, S., Gomez, S., Krauthammer, M., Kaplan, S., Kra, P., Russo, J. & Friedman, C. (2000) A knowledge model for analysis and simulation of regulatory networks. *Bioinformatics*, 16 (12), 1120–1128.
- [Saunders *et al.*, 2008] Saunders, B., Lyon, S., Day, M., Riley, B., Chenette, E., Subramaniam, S. & Vadivelu, I. (2008) The Molecule Pages database. *Nucleic Acids Research*, 36, D700–706.
- [Schacherer *et al.*, 2001] Schacherer, F., Choi, C., Götze, U., Krull, M., Pistor, S. & Wingender, E. (2001) The TRANSPATH signal transduction database: a knowledge base on signal transduction networks. *Bioinformatics*, 17 (11), 1053–1057.
- [Seipel, 2002] Seipel, D. (2002) Processing XML–documents in PROLOG. In *Proc. 17th Workshop on Logic Programming (WLP 2002)*.
- [Spellman *et al.*, 2002] Spellman, P. T., Miller, M., Stewart, J., Troup, C., Sarkans, U., Chervitz, S., Bernhart, D., Sherlock, G., Ball, C., Lepage, M., Swiatek, M., Marks, W., Goncalves, J., Markel, S., Iordan, D., Shojatalab, M., Pizarro, A., White, J., Hubley, R., Deutsch, E., Senger, M., Aronow, B. J., Robinson, A., Bassett, D., Stoeckert, C. J. & Brazma, A. (2002) Design and implementation of microarray gene expression markup language (MAGE-ML). *Genome Biology*, 3 (9), RESEARCH0046.
- [Wang & Dohlman, 2004] Wang, Y. & Dohlman, H. G. (2004) Pheromone signaling mechanisms in yeast: a prototypical sex machine. *Science*, 306 (5701), 1508–1509.
- [Wielemaker, 2009] Wielemaker, J. (2009) SWI-PROLOG 5.0 Reference Manual and Wielemaker, J., Anjewierden, A. (2009) Programming in XPCE/PROLOG, <http://www.swi-prolog.org/>
- [Zheng *et al.*, 2008] Zheng, S., Sheng, J., Wang, C., Wang, X., Yu, Y., Li, Y., Michie, A., Dai, J., Zhong, Y., Hao, P., Liu, L. & Li, Y. (2008) MPSQ: a web tool for protein–state searching. *Bioinformatics*, 24, 2412–2413.



---

# The Contact-Center Business Analyzer: a case for Persistent Contextual Logic Programming

Claudio Fernandes<sup>1</sup>, Nuno Lopes<sup>\*2</sup>, Manuel Monteiro<sup>3</sup>, and Salvador Abreu<sup>1</sup>

<sup>1</sup> Universidade de Évora and CENTRIA, Portugal  
`{cff,spa}@di.uevora.pt`

<sup>2</sup> Digital Enterprise Research Institute, National University of Ireland, Galway  
`nuno.lopes@deri.org`

<sup>3</sup> xseed, Lda., Portugal  
`manuel.monteiro@xseed.pt`

**Abstract.** This article presents CC/BA, an integrated performance analyzer for contact centers, which has been designed and implemented using Persistent Contextual Logic Programming methods and tools. We describe the system’s architecture, place it in perspective of the existing technology and argue that it provides interesting possibilities in an application area in which timely and accurate performance analysis is critical.

## 1 Introduction

*The problem:* The majority of the current contact center solutions provide extensive reporting, and some of them already provide even more sophisticated business intelligence solutions, allowing their users to analyze thoroughly the operational aspects of the contact center activity [6]. Performance indicators [5] normally made available are those related with how efficient the automatic call distributors (ACDs) are in distributing the calls (waiting times, abandoned calls, etc.) and how efficient the agents are in handling them (e.g. handled calls, call duration).

As there is normally little or even no integration with the surrounding business systems and applications, and there is no integration with the business data related with the various interactions processed by the contact center, these solutions can not correlate the operational data with the business data in order to provide more business oriented key performance indicators, like costs, profits and related ratios (operation margins, segment or customer value, ...).

---

<sup>\*</sup> This author has been funded in part by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2).

---

Our purpose in developing CC-BA, the Business Analyzer tool, is to overcome the identified lack of systems integration and to allow to model and integrate business data with the existing operational data. Systems that might need to be integrated are workforce management tools, CRM and ERP tools.

*Contexts:* The idea of Contextual Logic Programming (CxLP) was introduced in the late 1980s by Monteiro and Porto [8] in the ALPES II project, and is related to similar efforts such as Miller’s  $\lambda$ Prolog module system, described in [7].

The purpose of Contextual Logic Programming (CxLP) was initially to deal with Prolog’s traditionally flat predicate namespace, a feature which seriously hindered the language’s usability in larger scale projects. The impact of these extensions has mostly failed to make it back into the mainstream language, as the most widely distributed implementations only provide a simple, SICStus-like module mechanism, if any.

A more recent proposal [2] rehabilitates the ideas of Contextual Logic Programming by viewing contexts not only as shorthands for a modular theory but also as the means of providing dynamic attributes which affect that theory: we are referring to unit arguments, as described in Abreu and Diaz’s work. It is particularly relevant for our purposes to stress the *context-as-an-implicit-computation* aspect of CxLP, which views a context as a first-class Prolog entity – a term, which behaves similarly to objects in OOP languages.

*Persistence:* Having persistence in a Logic Programming language is a required feature if one is to use it to construct actual information systems; this could conceivably be provided by Prolog’s internal database but is quite adequately accounted for by software designed to handle large quantities of factual information efficiently, as is the case in relational database management systems. The semantic proximity between relational database query languages and logic programming have made the former privileged candidates to provide Prolog with persistence, and this has long been recognized.

ISCO [1] is a proposal for Prolog persistence which includes support for multiple heterogeneous databases and which access to technology beyond relational databases, such as LDAP directory services or DNS. ISCO has been successfully used in a variety of real-world situations, ranging from the development of a university information system to text retrieval or business intelligence analysis tools.

ISCO’s approach for interfacing to DBMSs involves providing Prolog declarations for the database relations, which are equivalent to defin-

---

ing a corresponding predicate, which is then used as if it were originally defined as a set of Prolog facts. While this approach is convenient, its main weakness resides in its present inability to relate distinct database goals, effectively performing joins at the Prolog level. While this may be perceived as a performance-impairing feature, in practice it is not the show-stopper it would seem to be because the instantiations made by the early database goals turn out as restrictions on subsequent goals, thereby avoiding the filter-over-cartesian-product syndrome.

*Contexts and persistence:* Considering that it is useful to retain the regular Prolog notation for persistent relations which is an ISCO characteristic, we would like to explore the ways in which contexts can be taken advantage of, when layered on top of the persistence mechanisms provided by ISCO. In particular we shall be interested in the aspects of common database operations which would benefit from the increase in expressiveness that results from combining Prolog's declarativeness and the program-structuring mechanisms of Contextual Logic Programming.

We shall illustrate the usefulness of this approach to Contextual Logic Programming by providing examples taken from a large scale application, written in GNU Prolog/CX, our implementation of a Contextual Constraint Logic Programming language. More synthetic situations are presented where the constructs associated with a renewed specification of Contextual Logic Programming are brought forward to solve a variety of programming problems, namely those which address compatibility issues with other Prolog program-structuring approaches, such as existing module systems. We also claim that the proposed language and implementation mechanisms can form the basis of a reasonably efficient development and production system.

The remainder of this article is structured as follows: section 2 introduces ISCO and Contextual Logic Programming as application development tools, proposing a unified language featuring both persistence and contexts. Section 3 addresses design issues for a performance analysis tool and introduces CC-BA. The impact of some design issues and environmental requirements is further explored in section 4, where some experimental results are presented. Finally, section 5 draws some conclusions and points at interesting directions for future work.

## 2 Persistent Contextual Logic Programming

Logic Programming and in particular the Plug language has long been recognized as a powerful tool for declaratively expressing both data and

---

processes, mostly as a result of two characteristics: the logic variable with unification and nondeterminism as a result of the built-in backtracking search mechanism.

While the Prolog is a consensual and relatively simple language, it is lacking in a few aspects relevant to larger-scale application development; some of these include:

- The lack of sophistication of its program structuring mechanisms: the ISO modules [4] proposal is representative of what has been done to take Prolog beyond the flat predicate space. The standard adds directives for managing sets of clauses and predicates, which become designated as *modules*. There are several areas where modules complicate matters w.r.t. regular Prolog: an example is the metacall issue, in which predicate arguments are subsequently treated as goals inside a module.
- The handling of persistent evolving data: one of Prolog’s most emblematic characteristics is the way in which programs could easily be manipulated in runtime, by use of the `assert` and `retract` built-ins. These can be used to alter the program database, by adding or removing clauses to existing predicates. There are several issues with these, namely how do these interact with goals which have an active choice-point on a predicate being modified.

One interesting feature provided by traditional Prolog implementations is the possibility of asserting non-factual clauses, which effectively endows the language with a very powerful self-modification mechanism. Nevertheless, the most common uses for clause-level predicate modification primitives deal with factual information only, this is true both for predicates which represent state information (which are typically updated in a way which preserves the number of clauses) and for database predicates, which will typically have a growing number of clauses.

Both of these aspects are addressed by the ISCO programming system, when deployed over GNU Prolog/CX, which compiles Prolog to native code. This implementation platform is discussed in [3] and briefly summarized in the rest of the present section.

## 2.1 Modularity with GNU Prolog/CX

Contextual Logic Programming was initially introduced with the intention of addressing the modularity issue. At present and in the GNU Prolog/CX implementation, CxLP provides an object-oriented framework for

---

the development of Logic programs; this is achieved through the integration of a passive, stateful part (the context) with a dynamic computation (a Prolog goal). The context is made up of instances of *units* which are similar to objects with state in other languages.

The current implementation has support for dynamic loading and unloading of units, which are implemented as OS-level shared libraries, or DLLs.

## 2.2 Persistence and large Database Predicates

Relational Database Management Systems provide efficient implementations for data storage, tailored to certain ways of accessing the information. The access is done via the SQL query language and the efficiency comes partly as a consequence of the availability of certain optimization features, such as multiple indexes on database relations. Prolog systems have long incorporated similar optimizations which map onto abstract machine-level features, such as clause indexing. Most prolog systems implement forms of indexing which are less general than the possible counterparts in an RDBMS.

Maintaining multiple indices – i.e. indexing predicates based on the analysis of anything other than the first argument – is intrinsically expensive and implies a significant runtime overhead to decide which index to use for a given call. As a consequence, very few Prolog systems incorporate any form of multi-argument indexing, usually requiring special annotations to indicate the fact.

Moreover, the combination of indices with dynamic predicates requires a dynamic rebuild of those indices, in particular one which does not survive the process in which it occurred. This means that, even if a Prolog database was appropriate for representing a given fact base, its use incurs the overhead of rebuilding the index each time:

- the predicate is updated (via `assert` or `retract`),
- the Prolog program is started

## 3 Application Design

*The Contact Center Business Analyser* is a web based application to analyse and control the performances of the numerous operations realized through managing a call center. Relying in data fetched from the company clients, it builds detailed financial scopes to be used by different management positions at the center.

---

A call center is a company department which purpose is to materialize personal communication between a company commercial campaign and their clients. The talking, realized by phone calls, is the job of the various agents working at the center, although mail and faxes are used sometimes.

The CC-BA software presents different levels of usage, divided in two groups: one for the administrator and the other for the end-users of the application. As expected, running a contact center requires many “eyes” over the operations, each pair monitoring different subjects. CC-BA maps that concept by providing several kinds of analysis, each one oriented for a particular profile. Each profile can then obtain different views of the data, grouped as several *KPIs* - *key performance indicators*.

Every analysis has a time window. This time period can take two forms: an interval period between two days, weeks or months, and a fixed hour schedule, also included in a time period.

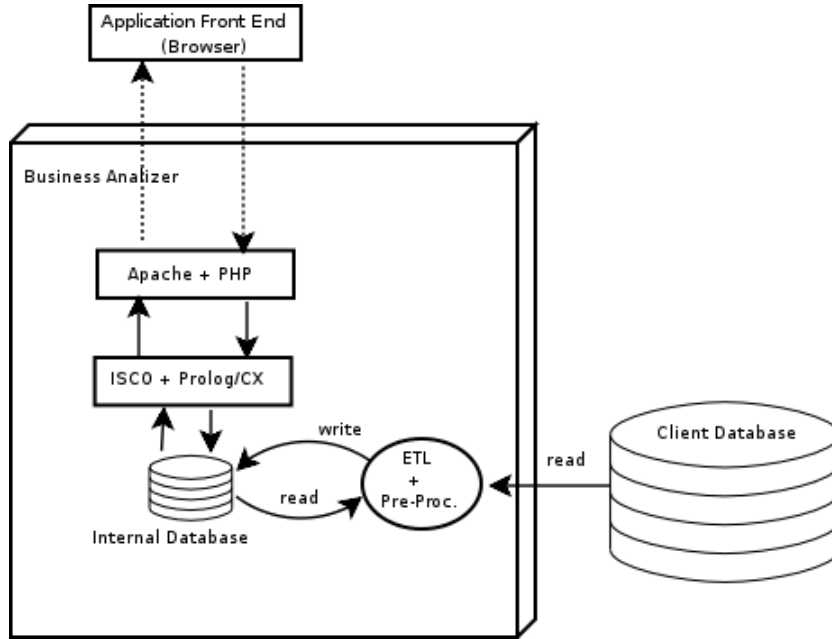
### 3.1 Architecture

The CC-BA structure is similar to the traditional three layer web based applications, but with some slight differences. As shown in figure 1 (see page 7), we have a local database, an ISCO + GNU Prolog/CX Logical layer, and finally a small PHP layer responsible for the Prolog and Browser communication. When comparing to the traditional three layer approach, the ISCO + GNU Prolog/CX layer implements both the Logic and Presentation Layers. The Data layer is provided by a PostgreSQL database, which is populated in the ETL [9] process.

The data used by the application must be fetched from the database of the call center. However, only a small fraction of that data is relevant, and before it is loaded some transformations are needed to be done over it. This process must be done periodically and is called ETL (**E**xtraction, **T**ransformation and **L**oading).

The internal database is compact compared with the ones where the data originated from, however it is a very important component of the application and it must be extremely well tuned to perform the best it can over the queries. The tables are created through the definition of ISCO classes. Those classes, when compiled, will generate Prolog predicates that access the data as if they were regular clauses in a Logic Programming environment.

The core of the application are the available outputs formed by groups of *key performance indicators*. With the goal of achieving a modular system, the implementation of those outputs were designed in a way that adding new outputs formed by any aggregation of KPIs is a linear task.



**Fig. 1.** CC-BA Architecture

Each *KPI* and output are coded in specific and different contextual Prolog units, such that to create a new output one only has to aggregate the *KPIs* desired into the new output unit. Any needed input parameters are pushed in through the GNU Prolog/CX context mechanism.

### 3.2 Example

Figure 2 (on page 8) depicts the unit that implements one use-case: the *Database Information* output. Just to name a few, we have the *Database Length* that is just the length of the contact list of a campaign, the “Successful Contacts”, “Reschedules” and “Unsuccessful Contacts” that are based on the “Contact Numbers”, switching the input parameter as it needed (lines 13, 16 and 19). The context mechanism of GNU Prolog/CX can also be used to specify unit parameters. In the example, the unit `param/3` is used to specify the arguments passed by, accordingly to what comes instantiated. In this case, a Client, a Campaign or an Agent.

Of all the parameters passed by context, there is one which requires special attention: the dates. Every analysis falls in a window period, and therefore all queries have time boundaries. Since the system had strict

---

```

1  :- unit(trat_bd).
2
3  output(Output):-
4      var('campaign', CP),
5      campaign@(code=CP, list_size=Total),
6      format(d, Total):format(Total_DB),
7
8      param(_,CP,_).proc_regs:value(Regs),
9      format(d, Regs):format(ProcRegs),
10
11     rate(Regs, Total):value(ProcRate),
12
13     param(_,CP,_).contact_numbers([successful]):value(Succ),
14     format(d, Succ):format(SuccCont),
15
16     param(_,CP,_).contact_numbers([reschedule]):value(Resc),
17     format(d, Resc).format(RescCont),
18
19     param(_,CP,_).contact_numbers([unsuccessful]):value(Unsucc),
20     format(d, Unsucc):format(UnsuccCont),
21
22     Use is Succ + Unsucc,
23     format(d, Use):format(UseContact),
24
25     rate(Use, Total):value(PenetRate),
26     rate(Succ, Use):value(SuccRate),

```

**Fig. 2.** Unit “trat\_bd”

performance requirements, we had to explore alternative representations for some information. In this case, every date is mapped to an integer value, over which it is possible to set finite-domain constraints in GNU Prolog/CX.

### 3.3 ETL

The ETL (Extraction, Transform and Load) process consists of loading the database with the information provided about the Contact Center operations. This is an “offline” process, i.e. it can be performed at any time, without human intervention.

The extracted data is to be provided to the system in a predefined format, which can be outputted by the operational system of the Contact Center.



---

In the transformation process, the extracted data is converted into data that can be directly loaded into CC-BA. This process starts by checking the date of the last valid ETL and proceeds to loading the interactions performed on a later date. This will include:

- loading new campaigns and agents
- delete older data from the tables “interaction” and “agentWork”
- manage the agents and campaign states

The existence of new campaigns or agents impose the input of data from the system administrator, such as the income of the agent and the costs of each interaction of a campaign. This will leave those agents and campaigns in an unprocessed state until such information can be entered, once the required information is defined the calculations can be performed. The ETL process can also recognize new interactions in an inactive campaign or agent and automatically active the campaign or agent.

## 4 Performance Evaluation

One of the most sensitive facts about the CCBA is its efficiency. It was necessary to guarantee that all the available analysis were computed in a reasonable time. An initially identified problem involved some of the analysis having a large search space. Several relations in the data model will contain, at some point and for the time intervals being considered, tens of millions of records. This fact makes it necessary to ensure that all queries are made in a quick and efficient way.

Since the results of the raw data model were not satisfactory, we had to find other approaches to improve performance. Also, due to some technological restrictions (described later) it was necessary to reduce the number of joins in the low-level (SQL) queries. We opted for the use of data accumulation together with some pre-processing; the following sections provide more detail on this.

### 4.1 Data Accumulation

With the exception of one output, all queries have a minimal granularity of one day. This observation allows us to make daily accumulated values of the records in the database, thereby obtaining very significant performance gains w.r.t. the raw data. To achieve this, the following relations were created, along with the system data model:

---

**agentWork\_daily:** This relation consists of the daily accumulates of a agent's work, i.e. the number of minutes an agent works per day and per campaign as well as the cost associated with that work.

**interaction\_daily:** Here we collect the daily cumulates of all the information pertaining to the interactions made by the agents, grouped by the outcome of the interaction, the number of interactions of that type, the total duration of the calls and the total invoicing value.

**interaction\_daily\_results:** This relation is similar to the previous one but, here, we do not include the agent information. This allows for a smaller search space to be used in queries that do not need agent information, for example searching by campaigns or clients.

## 4.2 Data Pre-processing

Data pre-processing is the action of inserting the correct data in the accumulator tables. This action needs to be done in two separate situations:

- After the ETL phase (see section 3.3);
- After the manager changes the values of a given campaign, or new entities are configured.

As a consequence of the changes made by the manager to the invoice model of a campaign or to the costs of the agents, the accumulator tables have to be updated. Since all changes affect all the data in the database (not only after the change is made), all data in the accumulator tables has to be invalidated and recomputed. This process is transparent to the user.

## 4.3 Computations using Pre-Processed Data

We present a few representative computations and discuss how they are implemented using cumulative data tables.

**Agent, Campaign and Client Costs.** Using the pre-processed data, the cost of an agent, campaign or client over a time period can be computed resorting to the **agentWork\_daily** table, where the total value of the cost is the sum of each day within the time period.

**Total Work Time by Agent.** The amount of time an Agent worked over a specified time period can be computed using the pre-processed information in the **agentWork\_daily** table.

---

**Contact duration.** Using the daily pre-processed information `interaction_daily`, computing the average duration of the contacts of a certain type, performed by an agent, becomes more efficient than the previous implementation. Should it be necessary to request the contact duration for a certain campaign, this information is available in the table `interaction_daily_results`.

**Invoice of an agent or campaign.** Using the table `interaction_daily`, the calculation of the invoicing total for an agent within a time period can be done with the sum of each day in the period, taking the value stored in the `invoice` field. In the same way, the invoice for a campaign can be calculated using the table `interaction_daily_results`.

**Total talk-time.** The total talk-time (in minutes) for an agent or campaign, within a time period, can be computed using the tables `interaction_daily` and `interaction_daily_result`.

**Answered contacts and total number of contacts.** The number of daily answered contacts in a campaign can be calculated more efficiently using the table `interaction_daily_results`, which is also used, together with `interactions_daily`, to obtain the total number of contacts performed by an agent.

#### 4.4 Views

With the correct use of indexes in the database fields and the cumulative tables in place, almost all queries were completed in a satisfactory time, given the desired operating conditions.<sup>1</sup> Nevertheless, some situations remained where the performance goals were not being achieved, namely in computing the number of contacts made by an agent and the total costs of the contact center.

In these situations, we chose to implement an SQL view for each one which is then mapped to an ISCO class. This allows the fields of the query to represent the desired result, while avoiding most further processing of the tuple data at the ISCO/application level. Each view is a *filter* for a larger table (possibly performing some computations), so

---

<sup>1</sup> In the range of 1-3 million events to treat, over a period of one month. The computational resources are very limited, our goal being that a simple workstation-style PC could be enough to support CC-BA.

---

that the result is computed by the DBMS instead of retrieving all the records and performing the filter in the application.

To achieve this goal, the view is designed to be similar to the table it applies to, replacing all non relevant fields by similarly named fields with a fixed value of the appropriate range and type (for instance the minimum value in the records). All the fields needed to perform the query are left *unbound*. An additional field with the result of the computation is introduced, in general this is an aggregator such as **sum** or **count**.

The definition of these views in ISCO is made in the same manner as for a regular database table. It extends the table it applies to by adding the computed field. The ISCO Prolog access predicates are limited to the “lookup” ones, i.e. a class which maps to a view is considered **static**.

The following are examples of situations which we dealt with by defining views at the database level:

**Number of contacts performed by an agent in a hour:** This query cannot be computed by using the accumulator tables since its granularity is smaller than one day (the granularity of the *daily* accumulator tables.) As it was not feasible to use the simple table **interaction** to perform this query, a view was created to perform the computation at the DBMS level.

**Processed Records:** The total number of processed records of a campaign in a given time interval consists, in SQL terms, in a **count(distinct \*)** from the contacts present in the **interaction** table. However, we know that **interaction** is the largest table present in the database with well above one million records per month of activity for a medium sized contact center, and computing this query cannot be directly done over it and remain computationally reasonable.

The introduction of a view decreased the execution time required for answering the query: in our first approach, this query was one of the slowest in the entire application, due to the amount of data that needed to be processed by ISCO. The use of the defined view, which left all the computations in the SQL side improved this situation dramatically.

The SQL code used of this view is shown in figure 3 and the definition of the view in ISCO is presented in figure 4. The ISCO side consists of extending the already defined class “**interaction**” with a new subclass which adds a “regs” field that is calculated in the view SQL code. Note that the SQL view must reproduce the fields which exist in the parent ISCO class.

---

```
1 CREATE VIEW "interaction_proc_regs" as
2     select
3         0 as oid,
4         campaign,
5         min(agent) as agent,
6         min(contact) as contact,
7         datetime,
8         min(duration) as duration,
9         min(result) as result,
10        count(distinct contact) as regs
11    from
12        interaction
13    group by campaign, datetime;
```

**Fig. 3.** Definition of “interaction\_proc\_regs” view

```
1 class interaction_proc_regs: interaction.
2     regs: int.
```

**Fig. 4.** ISCO definition of “interaction\_proc\_regs” view

When processing the records for 1.000.000 interactions, the times presented are for the first approach (all the calculations done in Prolog) are approximately 10 times higher than when using the defined view and cumulative data.

## 5 Conclusions and Directions for Future Work

Achieving the kind of data integration which CC-BA provides, coupled with the optimization capabilities to process these potentially very large amounts of data, results in great benefits for the call center management, meaning that the business oriented key performance indicators can be made available promptly, allowing the different manager roles to fine tune the systems to better reach their business goals, not only the operational ones.

One aspect of ISCO that turned out very useful is the ability to work with multiple indexes for database-stored data. For the quantities of data involved this turned out to be a critical aspect, as the same (factual) predicates are being accessed with different instantiation patterns.

---

We found out the hard way that the ETL process still requires a lot of manual effort and tuning. It would be worthwhile to investigate the automation of the process and the design of the internal database, paying particular attention to the unification of several heterogeneous data sources, to the selection of the hierarchical cumulative sums, as well as to the generation of the internal database schema. Some of these aspects are already being worked on, in particular the design of declarative tools to achieve a measure of automation of the ETL process is the object of ongoing work.

On a finishing note, it is worth mentioning that our implementation of CC-BA is able to adequately function inside a virtual machine running Debian Linux with very low resource requirements: a simple Pentium-M based laptop was enough to account for tracking a medium-sized contact center.

## References

1. Salvador Abreu. Isco: A practical language for heterogeneous information system construction. In *Proceedings of INAP'01*, Tokyo, Japan, October 2001. Prolog Association of Japan.
2. Salvador Abreu and Daniel Diaz. Objective: in Minimum Context. In Catuscia Palamidessi, editor, *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer-Verlag, 2003. ISBN 3-540-20642-6.
3. Salvador Abreu and Vitor Nogueira. Using a Logic Programming Language with Persistence and Contexts. In Osamu Takata, Masanobu Umeda, Isao Nagasawa, Naoyuki Tamura, Armin Wolf, and Gunnar Schrader, editors, *Declarative Programming for Knowledge Management, 16th International Conference on Applications of Declarative Programming and Knowledge Management, INAP 2005, Fukuoka, Japan, October 22-24, 2005. Revised Selected Papers.*, volume 4369 of *Lecture Notes in Computer Science*, pages 38–47. Springer, 2006.
4. ISO/IEC JTC1/SC22/WG17. Information technology – Programming languages – Prolog – Part 2: Modules. Technical Report DIS 13211, ISO, 2000.
5. Ger Koole. Performance analysis and optimization in customer contact centers. In *QEST*, pages 2–5. IEEE Computer Society, 2004.
6. Pierre L’Ecuyer. Modeling and optimization problems in contact centers. In *QEST*, pages 145–156. IEEE Computer Society, 2006.
7. Dale Miller. A logical analysis of modules in logic programming. *The Journal of Logic Programming*, 6(1 and 2):79–108, January/March 1989.
8. L. Monteiro and A Porto. Contextual logic programming. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 284–299, Lisbon, 1989. The MIT Press.
9. Panos Vassiliadis, Alkis Simitsis, and Spiros Skiadopoulos. Conceptual modeling for etl processes. In *DOLAP '02: Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP*, pages 14–21, New York, NY, USA, 2002. ACM.

---

# An Efficient Logic Programming Approach to Sub-Graph Isomorphic Queries\*

Hasan Jamil

Department of Computer Science  
Wayne State University, USA  
jamil@cs.wayne.edu

**Abstract.** In emerging applications such as biological networks and evolving social networks, computing sub-graph isomorphism is a necessary artifact for answering many interesting questions. Efficient computation of sub-graph isomorphism, however, has remained a difficult problem in artificial intelligence and databases. In logic programming, isomorphic sub-graph search has met with limited success mainly due to representation hurdles, large memory needs and lack of a suitable procedure. In this paper, we demonstrate that a simple representation of undirected graphs enables logic based computation of isomorphic sub-graphs very efficiently. We also present a graph and query rewriting technique to compute such queries using logic programming languages such as XSB or Prolog. The space and memory requirement for this framework is extremely low even for very large networks due to the representation technique, and the manner in which the procedure functions.

## 1 Introduction

Graph and graph isomorphism is an extensively studied subject in mathematics and many areas of computer science. Graphs model powerful concepts in artificial intelligence, business, science and the social sciences that cannot be captured easily using traditional data representation models such as relational models. Although data models such as XML come close to graph representations, they do not necessarily support graphs as the primary object. Recent applications in Life Sciences such as in [6, 10, 11], in Social Networks such as in [12, 5], and XML Data Management [15, 2, 16] inspired researchers to take a fresh look at the graph data model. In this paper, we propose a new graph representation model, and using this model, we show that efficient computing of sub-graph isomorphism for generalized undirected graphs using logic programming is possible.

Sub-graph isomorphism involves finding a mapping  $\mu$  from the nodes of a given query graph  $Q$  to the nodes in the target graph  $T$  such that the set of edges in  $Q$  and  $\mu(T)$  are identical, where  $\mu(T)$  denotes the set of all edges of  $T$  under the mapping<sup>1</sup>  $\mu$ . While there have been many algorithms for computing shortest

---

\* Research supported in part by National Science Foundation grants CNS 0521454 and IIS 0612203.

<sup>1</sup> Here the assumption is that the vertices in the two graphs are disjoint.

---

paths using Dijkstra’s algorithm, graph coloring, and so on in logic programming, sub-graph isomorphism has not been addressed adequately. Representation and large memory needs seem to be the primary bottlenecks. Traditionally, graphs in logic programming have been represented using lists in two major forms: as a pair of vertices and edges, and as a pair of vertices and their neighbors, in addition to the naive representation of edges as just a set of binary predicates. However, none of these representations make it easy to process isomorphic graph queries. Our attempt to address sub-graph isomorphism in logic programming appears to be the very first since no reference could be found in electronic literature databases that address the issue of sub-graph isomorphism in logic.

Although unification provided a great opportunity for the mapping function  $\mu$  needed, we believe the reason for not having a successful logic programming solution to sub-graph isomorphism is that the three traditional graph representations did not lend themselves to a procedure capable of isolating the structural similarities of the pattern or query graph, and the target graph. In all three representations, their simplicity did not capture enough information to reason about their structures or to identify the structural pieces needed to reconstruct the target sub-graph. In this paper, we present three novel ideas: represent each vertex as the smallest structural unit of the graph it is a member of, a procedure requiring a necessary and sufficient condition to reconstruct any target graph from these unit structures, and finally, a procedure to generate a query from the query graph such that the unification mechanism acts directly as the mapping function  $\mu$ .

### 1.1 Related Research

The use of graphs and graph searching remains an active area of research in artificial intelligence, and in many application areas as discussed earlier despite being a difficult problem to address. However, most known graph morphism problems are at least NP-Complete, Graph homomorphism is the most general graph morphism problem and is NP-Complete. The graph isomorphism problem on the other hand is not known to be NP-Complete, or in P. Although there is no polynomial time algorithm, most of the instances can be solved efficiently with state-of-the-art software for graph iso-morphism such as Nauty [8].

Subgraph isomorphism is NP-Complete but is tractable for some special classes of graphs, such as trees [14], planar graphs [4], and bounded valence graphs [7]. Two algorithms based on backtracking are well known for sub-graph isomorphism – Ullmann’s algorithm [13] and Vfib [3, 14] – and outperform almost all other known algorithms in the literature. In terms of space and time requirements, Vfib appears to be more attractive than Ullmann’s algorithm. However, despite the existence of these competing algorithms, there seems to be an absence of logic programming approaches to sub-graph isomorphism problem<sup>2</sup>, and hence the primary aim of this paper.

---

<sup>2</sup> In fact, no logic programming implementation of any subgraph isomorphism procedure could be found in the literature. It is said that attempts were made to implement



---

In relation to the existing works in the literature, the contributions of this paper can be summarized as follows. We propose a new knowledge representation method to model graphs as a set of node descriptions that essentially is a set of neighbors and the number of minimum structures (triangles) the node is involved in. We then propose the notion of structural unification as an extension of traditional unification in logic programming that makes it possible to compute isomorphic subgraphs as a conjunctive query, which was not possible before. It turns out that this approach is also quite fast and beats Vfib (and thus implicitly, Ullmann's algorithm) by a number of order of magnitudes in benchmark data sets [1] as discussed in section 4.2.

## 2 A Conceptual Graph Data Model

The conceptual model proposed exploits newly discovered general properties most graphs in Biology, Science, Social Science and Business networks share. For example, patterns in a graph are not entirely random, and most graphs form sub-networks of clusters and hubs. In our data model, we use structural properties of nodes as opposed to the adjacency of nodes alone, and thus, represent each node in terms of its neighbors, and the minimum closed structures they form. We treat a triangle as the minimum closed structure, and an edge as a minimum open structure. The idea is that once a node is described in terms of its neighbors and closed structures, it is possible to reconstruct the graph by tracing back the connectivity of the nodes regardless of the interaction graph size, type and form. As we will show, this representation also greatly facilitates the exploration of the graph in a piecewise manner to find isomorphic patterns by novel application of the structural constraints. As we will see in the next few sections, this piecewise representation of the vertices in terms of its closed and open structures makes it possible to capture each vertex separately, yet contain all the structural information we need to use it exactly like a block of a jigsaw puzzle. To proceed, we need a few definitions to keep our discussion focused and to the point.

First, we need to formally define the notion of *pure* graphs and databases in which we do not allow any dangling edge or node. Since we are also interested in finding isomorphic sub-graphs, we then define a sub-graph, based on which we define sub-graph isomorphism in our framework.

**Definition 21 (Pure Graphs)** Let  $N$  be a set of labeled vertices, and **edge** be a distinguished predicate. Then **edge** is a relation or graph over  $N$  such that **edge**  $\subseteq N \times N$ . **edge** is *pure* if whenever  $\langle n, n' \rangle \in \text{edge} \Rightarrow \langle n', n \rangle \in \text{edge}$ , and  $\forall n, n' (\langle n, m \rangle, \langle n', m' \rangle \in \text{edge} \Rightarrow \langle n, n' \rangle \in \text{edge}^t)$  where  $\text{edge}^t$  is the transitive closure of **edge**.

---

Vfib in Prolog at first (ref: As per communication with Professor Larry Holder, CS Department, Washington State University). But the implementation was either too slow, memory intensive or difficult, and finally the version in distribution was implemented in C/C++. The Vfib authors did not respond to an attempt by this author to obtain the said Prolog code (if exists).

**Definition 22 (Sub-Graphs)** Let *edge* and *query* be two pure graphs over a set of vertices  $N$ , and let  $N_e \subseteq N$  and  $N_q \subseteq N$  respectively denote their set of vertices. We say *query* is a sub-graph of *edge*, i.e.,  $\text{query} \subseteq \text{edge}$ , iff  $N_q \subseteq N_e$  and  $\forall n_1, n_2 \in N_q, \langle n_1, n_2 \rangle \in \text{edge} \Leftrightarrow \langle n_1, n_2 \rangle \in \text{query}$ . *edge* and *query* are equal, i.e.,  $\text{edge} = \text{query}$ , if  $N_e = N_q$ , and  $\text{query} \subseteq \text{edge}$ .

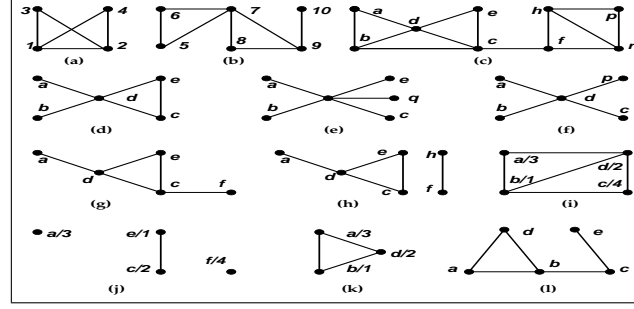


Fig. 1. Example Graphs.

Consider undirected pattern graphs, or a query graph,  $Q_1$  and  $Q_2$ , and a target graph, or a database graph,  $D$  as shown in figures 1(a) through (c) respectively. All these graphs are pure, because all the nodes are connected, i.e., there is a path from any node in the graph to all other nodes. Now according to definition 22, the graph in figure  $D_1$  in 1(d) is a subgraph of  $D$ , but none of the graphs  $D_2$  (figure 1(e)) or  $D_3$  (figure 1(f)) are subgraphs of  $D$ . Because all the nodes and edges in  $D_1$  are a pairwise subset of  $D$ , and both  $D_2$  and  $D_3$  have an edge or a node (respectively the edge  $(d, q)$  and node  $q$ , and the edge  $(d, p)$ ) that  $D$  does not have.

## 2.1 Node Properties

Mathematically, graphs are denoted as  $G = \langle N, E \rangle$  where  $N$  is the set of vertices, and  $E$  is the set of edges. When  $E$  is symmetric, the graph is considered unordered, it is ordered otherwise. For every node  $n \in N$  in any graph  $G = \langle N, E \rangle$ , the fan out  $\phi$  of  $n$  is defined as the cardinality of the set  $E^n \subseteq E$ , i.e.,  $\phi^n = |E^n|$ , such that  $\forall e (e = \langle n_1, n_2 \rangle \in E \wedge ((n_1 = n) \vee (n_2 = n)) \Leftrightarrow e \in E^n)$ . For every node  $n \in N$ , and all edges  $e \in E$  of the form  $\langle n, n_i \rangle$  the set  $\varrho^n = \cup_{i=1}^{\phi^n} \{n_i\}$  is called the neighborhood of  $n$ , and  $\forall e, e \in E$  such that  $e = \langle n_1, n_2 \rangle, \exists e_1, e_2, (e_1 = \langle n, n_1 \rangle, e_2 = \langle n, n_2 \rangle, n \neq n_1 \neq n_2)$  is called the boundary set of  $n$ , denoted  $\beta^n$ . The cardinality of the boundary set is called the triangle count, or t-count, of  $n$ , denoted  $\tau^n$ , i.e.,  $\tau^n = |\beta^n|$ .

In figure 1(c), for node  $d$ ,  $\phi^d = 4, \varrho^d = \{a, b, c, e\}, \beta^d = \{(a, b), (b, c), (c, e)\}$  and  $\tau^d = 3$ . Similarly for node  $f$ ,  $\phi^f = 3, \varrho^f = \{c, h, r\}, \beta^f = \{(h, r)\}$  and  $\tau^f = 1$ .

---

## 2.2 Sub-Graph Isomorphism

Following the classical definitions of graph and sub-graph isomorphisms, we adapt our definition for these two terms in our model as follows. Two graphs, say  $Q$  and  $G$ , are isomorphic if there is a one-to-one correspondence, or mapping, between their vertices, and there is an edge between two vertices of one graph if, and only if, there is an edge between the two corresponding vertices in the other graph. On the other hand, a graph  $Q$  is sub-graph isomorphic to another graph  $G$ , denoted  $Q \trianglelefteq G$ , if there is a one-to-one mapping between the vertices of  $Q$  and a subset of vertices of  $G$ , and there is an edge between two vertices of  $Q$  if, and only if, there is an edge between the two corresponding vertices in graph  $G$ .

According to this definition, the graph in figure 1(a) (shown with substitutions in figure 1(i)) is subgraph isomorphic to graph  $D$ , i.e.,  $Q \trianglelefteq D$  holds for the substitution  $S_1 = \{a/3, b/1, d/2, c/4\}$ . There, however, are several other such mappings that make it so, but for figures 1(j) and (k),  $Q \trianglelefteq D$  does not hold because for substitutions  $S_2 = \{(e/1), (c/2), (a/3), (f/4)\}$  and  $S_3 = \{(b/1), (d/2), (a/3)\}$  do not result in a similar structure that is a pure graph.

## 3 Vertex Structures and Vertex Reduced Databases

The computational approach to sub-graph isomorphism requires representation of graphs in a textual form that is amenable to clever manipulation. In our proposed representation, a graph is viewed as a set of node or vertex descriptions in which their relationships with other vertices are localized and decoupled from the entire graph in a more elaborate way than Ullmann's adjacency matrices [13]. Consequently, each vertex becomes a self sufficient sub-structure of a large graph, similar to a piece of a huge complex jigsaw puzzle. The idea is that if we pick the right assembling sequence from this collection of vertices, we should be able to put together the whole original graph, or a specific part of it. This is based on an assumption that for a given set of vertices and their associated sub-structures (nodes and edges), there is only one possible graph that can be built and it is always the intended (sub)graph. Intuitively, given a graph  $G = \langle N = \{a, b, c, d, e\}, E = \{(a, b), (b, c), (d, a), (b, d), (c, e)\} \rangle$ , for any given set of edges  $S \subseteq E$ , there is only one graph we can build which is always a subgraph of  $G$  (figure 1(l)).

It is important to understand the whole search and assembling process on intuitive grounds before we delve into a formal treatment. Let us represent the graph  $G$  in the following format.

$$\begin{aligned} &\langle a, 2, \{b, d\}, 1, \{(b, d)\} \rangle, \langle b, 3, \{a, c, d\}, 1, \{(a, d)\} \rangle, \langle c, 2, \{b, e\}, 0, \{\} \rangle, \\ &\langle d, 2, \{a, b\}, 1, \{(a, b)\} \rangle, \langle e, 1, \{c\}, 0, \{\} \rangle \end{aligned}$$

In the above representation, the expression  $\langle a, 2, \{b, d\}, 1, \{(b, d)\} \rangle$  captures everything structural we know about node  $a$  as discussed in section 2.1. As such, the description of  $a$  is the tuple  $\langle a, 2, \{b, d\}, 1, \{(b, d)\} \rangle$  where  $n = a$ ,  $\phi^a = 2$ ,  $\rho^a = \{b, d\}$ ,  $\tau^a = 1$  and  $\beta^a = \{(b, d)\}$ . Thus, for a given graph  $G = \langle N, E \rangle$ , the

---

description of a node  $n \in N$ , denoted  $\delta^n$ , in general is a quintuple of the form  $\langle n, \phi^n, \varrho^n, \tau^n, \beta^n \rangle$ . The set of all node descriptors,  $\delta^n$ , for all  $n \in G$ , is called a vertex reduced database, denoted  $D^\varphi$ , where  $\varphi$  is the transformation function that reduces  $D$  to  $D^\varphi$ .

Suppose now we are looking to find a structure  $Q = \langle \{1, 2, 3\}, \{(1, 2), (2, 3), (3, 1)\} \rangle$ . Then if we represent the query graph also in the following form

$$\langle X, 2, \{Y, Z\}, 1, \{(Y, Z)\} \rangle, \langle Y, 2, \{X, Z\}, 1, \{(X, Z)\} \rangle, \langle Z, 2, \{X, Y\}, 1, \{(X, Y)\} \rangle$$

by replacing node names with unique variable names, it is now possible to match the node description in  $Q$  with description for  $G$ , in a way similar to unification in logic programming. Notice that several optimizations are possible. For example, it turns out that the three descriptions of  $Q$  are logically and structurally identical, and if one of these descriptions unify with any description of  $G$ , the rest will too. So, we can only retain one. Finally, since all the  $Q$  terms are looking for a node that has two neighbors and one boundary set, they will never unify with node descriptions such as  $e$  (has  $\phi^e = 1$ , and  $\tau^e = 0$ ), or  $c$  (although  $\phi^c = 2$ , but  $\tau^c = 0$ ). It should be clear now, that finding any of the nodes  $a, b$  or  $d$  structurally similar to any of the query node descriptions, say for  $X$ , will actually find the isomorphic sub-graph of  $Q$  in  $G$  above once we recall the mapping  $X = 1, Y = 2$ , and  $Z = 1$ . The algorithm below makes sure that we only retain the necessary and sufficient set of node descriptions in our vertex reduced query graphs for processing.

#### Procedure Query\_Graph

**Input:** Query Graph  $Q$ .

**Output:** Vertex Reduced Query Graph  $\rho(Q)$ .

**begin**

- 1 Obtain vertex reduced graph  $V = \varphi(Q)$
- 2 using definition for vertex structure.
- 3 Re-label all nodes in all descriptors  $\delta^n \in V$
- 4 with unique variable names.
- 5 Sort the node descriptors  $\delta^X \in V$  in descending order
- 6 of  $\phi^X$  and then of  $\tau^X$  to obtain  $V_s$ .
- 7 For every descriptor  $\delta^X \in V_s$  in sorted order
- 8 of the form  $\langle X, \phi^X, \varrho^X, \tau^X, \beta^X \rangle$  such that
- 9  $X$  is not marked dead
- 10 Mark  $X$  dead.
- 11 For every node  $Y_i \in \varrho^X$  not already marked dead
- 12 with descriptor  $\delta^{Y_i}$
- 13 Remove all dead nodes from  $\varrho^{Y_i}$  to obtain  $\nu^{Y_i}$ .
- 14 If  $\nu^{Y_i} \subseteq \varrho^X$
- 15 Mark  $Y_i$  dead.
- 16 Remove  $\delta^{Y_i}$  from  $V_s$ .
- 17 Return  $V_s$ .

**end**

Since graphs are fundamentally structures involving relationships among vertices, to be able to compute isomorphic sub-graphs, we define the notion of

---

structural similarity. This definition will help us identify potential candidates for assembling, and help us put together an assembling sequence that is efficient and focused, as opposed to random or naive. Since a structural unit may be similar to a larger unit because it shares structural properties (a sub-structure), we define sub-vertex structure as follows.

**Definition 31 (Sub-vertex Structures)** Let  $\delta^n = \langle n, \phi^n, \varrho^n, \tau^n, \beta^n \rangle$  be a node descriptor. Then the node descriptor  $\langle n, \phi_s^n, \varrho_s^n, \tau_s^n, \beta_s^n \rangle$  derived from  $\delta^n$  by choosing  $\varrho_s^n \subseteq \varrho^n$ ,  $\beta_s^n \subseteq \beta^n$ ,  $\phi_s^n \leq \phi^n$ , and  $\tau_s^n \leq \tau^n$  such that  $|\varrho_s^n| = \phi_s^n$ , and  $|\beta_s^n| = \tau_s^n$ , is a sub-structure of  $\delta^n$ , denoted  $\delta_s^n \preceq \delta^n$ .

The structural similarity of two vertex structures can be determined using an extended definition of unification in logic programming. The goal of structural unification is to find a complete set of substitutions that make the two node descriptors such that both are ground, and the relationship  $\preceq$  holds between them. Since in our model, all data node descriptors are ground and all query node descriptors have place holders or variables for node names, we just need to focus on computing the query node descriptors as sub-vertices of data node descriptors in the database. Consequently, we try to find an instantiation of the query node descriptors for which they will become a sub-vertex of some data node descriptors.

That goal, unfortunately, raises some challenges. For example, for a query node descriptor  $\delta^X$ , any data node descriptor  $\delta^n$  will offer multiple possible substitutions when  $\phi^X < \phi^n$ . Higher the difference, the larger is the set of possible substitutions. The same comments apply to  $\tau^X$ . Consequently, the most general unifier in the case of structural unification is a distinct set of possible substitutions. These substitutions will produce a set of ground data node descriptors, one for each substitution. In this algorithm, we do not present the actual matching process of how the substitution  $\theta$  is computed. Interested readers may refer to [9] for an in depth discussion on most general unifiers, algorithms to compute most general unifiers, and related issues.

#### Procedure Structural\_Unification

**Input:**  $\varphi(G)$ , and  $\varphi(Q)$ .

**Output:**  $\Xi_G$ .

**begin**

```

1   Set  $\Xi_G = \emptyset$ .
2   For every descriptor  $\delta^{n_q} \in \varphi(Q)$ 
3     For every descriptor  $\delta^{n_g} \in \varphi(G)$ 
4       if  $\phi^{n_g} \geq \phi^{n_q}$  and  $\tau^{n_g} \geq \tau^{n_q}$ 
5         For every  $\phi^{n_g} C_{\phi^{n_q}}$  subsets  $s$  of  $\phi^{n_g}$ 
6           Let  $\theta = mgu(s \cup \{n_g\}, \varrho^{n_q} \cup \{n_q\})$ .
7           If  $(\beta^{n_q})[\theta] \subseteq \beta^{n_g}$ 
8              $\Xi_G = \Xi_G \cup \delta^{n_q}[\theta]$ .
9   Return  $\Xi_G$ .
```

**end**

---

It is always the case that if the conditions in line 4 are met, there will always be a successful unification because  $\delta^{n_q}$  is totally non-ground, and  $\delta^{n_g}$  is always ground. But, having a substitution  $\theta$  does not always guarantee structural similarity as boundary instances  $\beta^{n_q}[\theta]$  under substitutions may not match because of the differences in topology. Hence, the additional test in line 7 is required. Furthermore, the For qualifier  $\phi^{n_g} C_{\phi^{n_q}}$  subsets  $s$  of  $\phi^{n_g}$  in line 5 picks exactly the number of neighbors in  $\varrho^{n_g}$  such that the cardinality matches with  $\varrho^{n_q}$ . Consequently, larger the difference  $\phi^{n_g} - \phi^{n_q}$ , the larger is the size of the possible sets of substitutions. Since all substitutions are unique, all topologies selected are also unique. For example, the boundary (1, 2) and (2, 1) are identical, and hence both will never be selected. The same observation applies for neighborhood selection, i.e., neighbors {1, 2} and {2, 1} are identical.

## 4 An Interpreter for Sub-Graph Isomorphism

In this section, we develop a logical interpreter  $I$ , that given a graph  $G$  and a query graph  $Q$ , finds a mapping  $\mu$  from the vertices of  $Q$  to the vertices of  $G$  such that  $\mu(Q)$  is a sub-graph of  $G$  as defined in definition 22. We introduce the interpreter  $I$  in reference to the logic program in section 4.1 below. The interpreter is based on the idea of program transformation where a logic program  $P = \langle G, Q \rangle$  is rewritten as  $P^T = \langle \varphi(G), \rho(Q), A \rangle$ , and  $\rho(Q)$  is computed against  $\varphi(G) \cup A$  using a traditional logic engine such as Prolog or XSB. Here  $\varphi$  is the graph reduction function discussed in section 3, and  $\rho$  and  $A$  are respectively the query graph reduction function discussed in algorithm **Query\_Graph** and a set of axioms that captures the spirit of the structural unification introduced in algorithm **Structural\_Unification** discussed in the same section. Our example database below was computed using an online Prolog interpreter JIProlog version v.3.0.3-1 at <http://www.ugosweb.com/>.

### 4.1 Example Logic Program

```
1.  :- dynamic node/5, arc/2.

% Input Graph

2.  edge(a,d).   3.  edge(a,b).   4.  edge(b,d).   5.  edge(b,c).   6.  edge(c,d).
7.  edge(c,e).   8.  edge(c,f).   9.  edge(d,e).  10. edge(f,h).  11. edge(f,r).
12. edge(h,p).  13. edge(h,r).  14. edge(p,r).

15. arc(X,Y) :- edge(X,Y).
16. arc(X,Y) :- edge(Y,X).
```

#### % Graph Transformation Engine

---

<sup>3</sup> In this expression,  $\{(1, 2)\} \subseteq \{(2, 1), (2, 3)\}$  will test positively in our setup. In other words,  $(1, 2) \equiv (2, 1)$ , i.e., the order is immaterial. Also, notice that for any given  $\theta$ ,  $\delta^X[\theta]$  is always ground, and all  $\delta^n$  are ground by definition.

---

```

17. setof(X,Goal,Bag) :- post_it(X,Goal), gather([],Bag).
18. post_it(X,Goal) :- call(Goal), asserta(data999(X)), fail.
19. post_it(_,_).

20. gather(B,Bag) :- data999(X), retract(data999(X)), gather([X|B],Bag),
    distinct(Bag), !.
21. gather(S,S).

22. dist(X,[]).
23. dist(X,L) :- not member(X,L).
24. distinct([]).
25. distinct([X|L]) :- dist(X,L), distinct(L).

26. neigh(X,C,L) :- setof(Y,arc(X,Y),S), vrm_dupl(S, L), length(L,C).

27. bound(X,e(N1,N2)) :- neigh(X,Nc,Nb), Nc>1, member(N1,Nb),
    member(N2,Nb), not N1=N2, arc(N1,N2).

28. node(X,Nc,Nb,Ts,Bs) :- setof(e(N1,N2), bound(X,e(N1,N2)),TBs),
    neigh(X,Nc,Nb), rm_dupl(TBs,Bs), length(Bs,Ts).

29. rm_dupl([],[]).
30. rm_dupl([X|T],L2) :- symmem(X,T), rm_dupl(T,L2).
31. rm_dupl([X|T],[X|T1]) :- \+ symmem(X,T), rm_dupl(T,T1).

32. symmem(X,S) :- member(X,S).
33. symmem(e(X,Y),S) :- member(e(Y,X),S).

34. vrm_dupl([],[]).
35. vrm_dupl([X|T],L2) :- member(X,T), vrm_dupl(T,L2).
36. vrm_dupl([X|T],[X|T1]) :- \+ member(X,T), vrm_dupl(T,T1).

37. vertices(L) :- setof(X,arc(X,Y),S), vrm_dupl(S,L), !.

38. recurse([H|R]) :- recurse(R), node(H,Nc,Nb,Ts,Bs),
    write(node(H,Nc,Nb,Ts,Bs)), write('\n').
39. recurse([]).

40. transform :- vertices(R), recurse(R).

% Transformed Graph Representation in Minimum
% Structural Units

41. JIP:-transform.
Yes

42. node(d,4,[c,b,a,e],3,[e(b,c),e(a,b),e(e,c)]).
43. node(b,3,[a,c,d],2,[e(d,a),e(d,c)]).
44. node(c,4,[b,f,e,d],2,[e(d,b),e(d,e)]).

```

---

---

```

45. node(e,2,[d,c],1,[e(c,d)]).
46. node(f,3,[c,r,h],1,[e(h,r)]).
47. node(h,3,[f,r,p],2,[e(r,f),e(p,r)]).
48. node(r,3,[p,h,f],2,[e(h,p),e(f,h)]).
49. node(p,2,[h,r],1,[e(r,h)]).
50. node(a,2,[b,d],1,[e(d,b)]).

% Structural Unifier: Generic Interpreter for
% Sub-Graph Isomorphism

51. qry(V,Qneighcnt,Qneigh,Qtset,Qbounset):-
    node(V,Neighcnt,Neigh,Tset,Bounset), Neighcnt >= Qneighcnt,
    Tset >= Qtset, subset(Qneigh,Neigh), subset(Qbounset,Bounset).

52. subset([],_).
53. subset([X|R],S) :- symmem(X,S), subset(R,S).

% Input Query Graph

54. query(5,6). 55. query(5,7). 56. query(6,7). 57. query(7,8).
58. query(7,9). 59. query(8,9). 60. query(9,10).

% Transformed and Optimized Sub-graph
% Isomorphic Query

61. JIP:-qry(C,4,[A,B,E,F],2,[e(A,B),e(E,F)]),
    qry(F,3,[H,C,E],1,[e(C,E)]),
    distinct([A,B,C,E,F,H]).
Yes

62. C = d 63. A = a 64. B = b 65. E = e 66. F = c 67. H = f

```

## 4.2 Structure of the Interpreter $I$

The interpreter essentially consists of a set of axioms, and an external rewriting function to convert a query graph to a conjunction of subgoals, evaluation of which will provide the vertex mappings expected. Formally, the interpreter  $I$  is a tuple of the form  $\langle A_\varphi, \rho, A_\psi \rangle$  such that  $A_\varphi$  are the axioms that convert  $G$  to  $\varphi(G)$ ,  $\rho$  is the external function<sup>4</sup> for the transformation of query graph  $Q$  to  $\rho(\varphi(Q))$ , and the set of axioms  $A_\psi$  is the unifier that implements structural unification using classical unifier in the host system.

**Graph Transformation Function  $\varphi$**  The set of 24 rules in 17 through 40 implement the transformation function  $\varphi$  as axioms  $A_\varphi$ . The functionality of

---

<sup>4</sup> For the sake of simplicity and brevity we chose to use an external function for the current presentation. It, however, is readily possible to develop a meta-interpreter to transform the query graph  $Q$  to the conjunctive goals as presented, and is left as an exercise.



---

these rules can be described as follows. Please recall that a vertex is represented as a tuple of the form  $\langle n, \phi^n, \varrho^n, \tau^n, \beta^n \rangle$ . The predicate **neigh** in rule 26 of  $A_\varphi$  computes  $\phi^n$  and  $\varrho^n$  for a given vertex  $n$ . It does so by collecting every edge from the **arc** rules in 15 and 16 (which actually guarantee that the graph  $G$  represented in **edge** is symmetric) using Prolog's **setof** or **findall** like feature. We added our own version of **setof** rules because many Prolog interpreters do not include this feature, including the JIProlog we used. It then sanitizes the collected list of neighbors of  $n$  by removing duplicates and computes the cardinality of the set (of neighbors). Notice that for the **setof** rule to work, we need to make the argument  $X$  ground by supplying the name of the vertex. Rule 37 collects all the vertex names in a similar fashion into the list  $L$ . Rule 40, in turn, supplies these vertices to rule 38 (as well as to rules 28 and 26) as adornments. Rule 27 computes the set of all boundaries for a given vertex, and using 26 and 27, rule 28 computes the ultimate vertex representation for all the vertices in a manner similar to rule 26. Finally, rule 40 fires the transformation process by first generating all the vertices in list, and then for each member of the list, firing rule 28. Lines 42 through 50 show the transformed graph  $G$ . This is the representation we proposed that we use for sub-graph isomorphism queries. Notice that, rule 38 needs to be edited to replace the **write** subgoals with **assert** before these transformed vertex representation can be used for computation, or these predicates will need to be added manually to the program.

**Query Graph Reduction using  $\rho$**  The query graph  $Q$  is shown in lines 54 through 60 corresponding to the graph  $Q_1$  in figure 1(b). To develop the query in rule 61, we use the external function  $\rho$ , and first reduce  $Q$  to its vertex reduced form using the axioms  $A_\varphi$ . Then we use the algorithm **Query\_Graph** in section 3 to obtain only the sufficient and necessary set of vertex reduced general query graph (a graph is called general when all its node names are replaced with variable names). Since we require that these variables in all the vertex structures bind to distinct nodes, we form a conjunctive query using all the vertex structures returned by algorithm **Query\_Graph** and make sure that they bind to unique nodes by adding another subgoal **distinct**( $[L]$ ) (used in query 61, and implemented in rules 22 through 25), where  $L$  is the list of all variable names in all the vertex structures in  $\rho(\varphi(Q))$ . Finally, we let unification of the host language find the mapping.

**Structural Unification Axiom  $A_\psi$**  The structural unification now actually needs to account for matching  ${}^nC_k$  candidates in a given set of neighbors or boundary set in the target vertex with the pattern vertex. For that to be successful, rules 51, and 52 and 53 are enough to capture the spirit of algorithm **Structural\_Unification**. To be structurally unifiable, we need to query all **nodes** facts using **qry** rule that succeeds only when the required subset of the neighbors and boundary sets unify with the corresponding pattern neighbors and boundary set. Unification takes care of the rest. Notice here the application of **subset** rules 52 and 53 that succeeds for symmetric edges as discussed.

---

**Execution and the Bindings** The results for sub-graph isomorphic query for graph  $Q_1$  against the database  $D$  is given in lines 62 through 67 from an actual run in JIProlog. In this example, the mappings are then  $\{5 \rightarrow A/a, 6 \rightarrow B/b, 7 \rightarrow C/d, 8 \rightarrow E/e, 9 \rightarrow F/c, 10 \rightarrow H/f\}$ <sup>5</sup>.

**Experimental Evaluations** The main objective for the experimental evaluation carried out as part of this study was to establish a relative performance comparison with Vfib, arguably the best known subgraph isomorphism algorithm in the literature to date. The Dell Desktop we used to run these two procedures have a Pentium Dual Core processor with 4GB memory and 6MB cache. We have used the identical benchmark data sets as the developers of Vfib did in their paper [3] and have supplied online for the comparison. The graphs in figure 2 show how our algorithm significantly outperforms Vfib. In these figures, we have used two different kinds of graphs – called the random graphs and mesh graphs. A random graph has no basic protocol of constructing except that there is a parameter called  $\eta$  that reflects the probability that two nodes will have an edge between them. The mesh graphs on the other hand are constructed differently to assure a more tighter connection. All mesh graphs are first created as a ladder graph, and then a probability  $\rho$  is used to add more edges between nodes. The result is a more tightly connected and dense graph.

In general, more edges a node has, the more it costs to compute its isomorphic counterparts. So, we can expect higher computational costs, and thus performance difference between these two algorithms, in case of mesh graphs, as shown in figures 2(a) through 2(e). In these graphs, we keep the number of data nodes fixed and vary query nodes. It can be observed that as the number of query nodes, or the mesh density increases, our algorithm outperforms Vfib more and more, which is expected. Figure 2(f) through 2(j) show similar graphs, but this time we keep the query nodes fixed, and vary the number of data nodes in the target graphs. Here too, the algorithms behave similarly<sup>6</sup>.

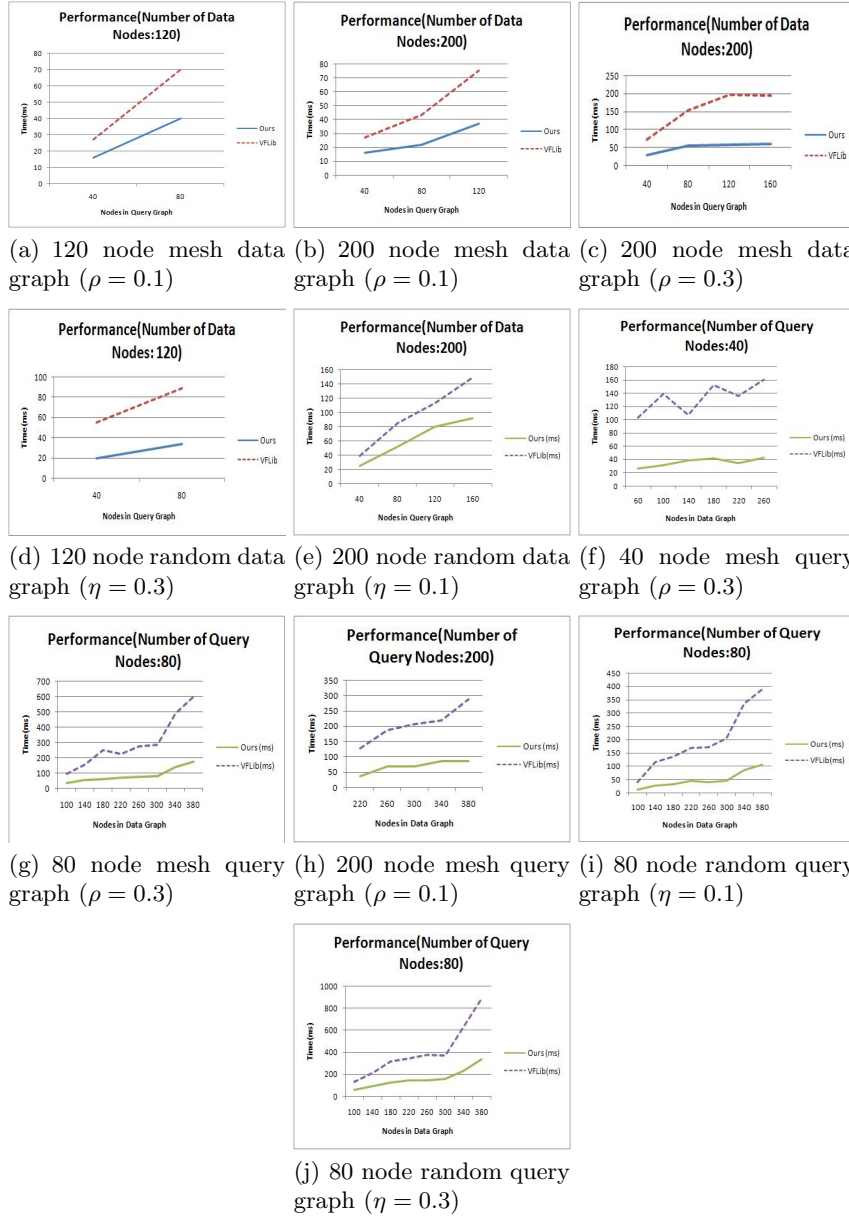
## 5 Conclusion

It should be evident that the core of our framework is in the way the nodes in a graph are represented, and how the concept of structural unification makes it possible to find and “fit” the pieces together in a structurally consistent way. The number of rules needed for graph reduction is only 27, and for structural unification only 2 (**qry** and **subset**). It is also interesting to note that subgoal evaluation needed is linearly proportional to the number of reduced vertices

---

<sup>5</sup> In this representation,  $9 \rightarrow F/c$  meant that node 9 in query graph  $Q_1$  was first replaced by variable  $F$  by the function  $\rho$  after reduction by  $\varphi$ , and then was unified with the node  $c$  of the database  $D$  by  $A_\psi$ .

<sup>6</sup> However, we did not have the opportunity to test the algorithms for high number of nodes in query and target graphs at this time because the data generation tool at [1] did not compile. Once we are able to fix this bug, we plan to conduct a more elaborate comparison in the future.



**Fig. 2.** Performance comparison of our procedure with Vflib (lower curve in each graph reflects our procedure). (a), (b) and (c) - on mesh graphs, (d) and (e) - on random graphs. In each of these graphs, performance is plotted as the number of query graph nodes are varied. Figures (f) through (j) show the performance when the number of data graph nodes are varied keeping query graph nodes fixed.

---

of query graph  $\rho(\varphi(Q))$ , and the solution space depends upon the number of solutions possible. The conditions such as `Neighcnt >= Qneighcnt`, `Tset >= Qtset` in  $A_\psi$  guarantee that only the potential candidates are tried, not all possible vertices. More optimization in the direction of query optimization in deductive databases using query rewriting and indexing is possible and remains our future research goals.

## References

1. *Graph Benchmark Database*. <http://amalfi.dis.unina.it/graph/>.
2. L. Chen and S. Bhowmick. In the search of nectars from evolutionary trees. In *DASFAA*, pages 714–729, 2009.
3. L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition*, pages 149–159, Cuen, 2001.
4. J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. In *ACM Symposium on Theory of Computing*, pages 172–184, 1974.
5. Y. Jia, J. Hoberock, M. Garland, and J. C. Hart. On the visualization of social and other scale-free networks. *IEEE Trans. Vis. Comput. Graph.*, 14(6):1285–1292, 2008.
6. M. Koyuturk, Y. Kim, S. Subramaniam, W. Szpankowski, and A. Grama. Detecting conserved interaction patterns in biological networks. *JCB*, 3(7):12991322, 2006.
7. E. M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. In *IEEE FOCS*, pages 42–49, 1980.
8. B. D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
9. E. Rich and K. Knight, editors. *Artificial Intelligence*. McGraw Hill Higher Education; 2nd edition, 1991.
10. S. Saito, S. Aburatani, and K. Horimoto. Network evaluation from the consistency of the graph structure with the measured data. *BMC Systems Biology*, 2(84), October 2008.
11. J. Scott, T. Ideker, R. M. Karp, and R. Sharan. Efficient algorithms for detecting signaling pathways in protein interaction networks. *J Comput Biol*, 13(2):133–144, March 2006.
12. N. Shrivastava, A. Majumder, and R. Rastogi. Mining (social) network graphs to detect random link attacks. In *ICDE*, pages 486–495, 2008.
13. J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of ACM*, 23(1):31–42, 1976.
14. G. Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag, Berlin, 2002.
15. H. Wang, J. Li, J. Luo, and H. Gao. Hash-based subgraph query processing method for graph-structured xml documents. *PVLDB*, 1(1):478–489, 2008.
16. H. Wang, J. Li, W. Wang, and X. Lin. Coding-based join algorithms for structural queries on graph-structured xml document. *World Wide Web*, 11(4):485–510, 2008.

---

# NL Database Dialogue Question-Answering as a Constraint Satisfaction Problem

Irene Pimenta Rodrigues, Ligia Ferreira, Luis Quintano

ipr@di.uevora.pt, lsf@di.uevora.pt, ljcq@di.uevora.pt

Departamento de Informática  
Universidade de Évora, Évora, Portugal

**Abstract.** This paper presents a system for database question-answering that is able to cooperatively answer the user. The system may provide justifications to answers that are not what the user intended and will also introduce clarification dialogues to disambiguate the user questions when it is relevant to do so, for instance when we have more than one interpretation for a user question.

Our prototype is implemented using declarative tools using GNU Prolog/CX and the ISCO framework which provides access to relational databases as prolog predicates.

In our systems sentences pragmatic are represented as a constraint satisfaction problem where the discourse entities are quantified finite domain variables constraint to the database relations.

This sentence representation enables the reasoning necessary for answering questions using generalized quantifiers such as: All, How Many, at least, what-more, etc. That allow the user to build factoids and list questions. We are not yet able to deal with definition, Why or How questions.

## 1 Introduction

This paper presents a system for database question answering that is able to cooperatively answer the user. The system may justify some answers which might otherwise appear as unexpected to the user. The system also inserts clarification dialogues to disambiguate the user's questions whenever relevant, namely when there are different possible answers to a user question. The types of dialogue we support in a QA system are *Clarification* and *Justification*:

**Clarification** - applies when we have more than one answer for a question. For example the query:

*Does John teach Databases?*

Requires clarification if, for instance, there are 3 teachers called *John* in our data or 4 courses with the word "Databases" in their name.

- If the answer is the same for all teachers called *John* and courses with "Databases" in their name, there is no need to clarify. It can be argued that, if the system requires the user to clarify the teacher and course in that situation, is not being cooperative.

- 
- If, for some teachers called *John* and Database courses, the answer is *yes* and for others the answer is *no* then the system must ask the user to clarify which teacher and course that are meant. To be deemed *cooperative*, the system must choose the best question, i.e. the one that leads the user to an answer in the least number of steps.

But to be cooperative, the system must also take the user knowledge into account. For example, if the user had already said that *John* name is called *John Smith*, presenting the all 3 teachers' full name may not be a cooperative attitude, it may be best to disambiguate on another propriety of *John*, such as the Department he works in.

**Justification** - This sort of dialogue is relevant when the answer is not what the user expects. Consider the question:

*Are all students enrolled?*

If the answer is **No** because only some of the students are enrolled, the system could relax the query by changing the discourse entities quantifiers, i.e. it could rewrite  $All(X)$  into something like  $HowMany(X)$ .

Our prototype is implemented in GNU Prolog/CX [8,2] using the ISCO persistence framework [3] which provides access to relational databases as prolog predicates.

In our system, the pragmatics of a sentence is represented as a Constraint Satisfaction Problem (CSP) where the discourse entities are quantified finite domain variables, constrained to the values which occur in the database relations. This builds on our previous work, reported in [18,19], extending it as follows: (1) questions may now be represented with generalized quantifiers and (2) interpretation is now formalized as a CSP. This sentence representation enables the reasoning necessary to answer questions using generalized quantifiers such as: *All*, *HowMany*, *AtLeast*, etc. which allow the user to build factoids and list questions. We are not yet able to deal with definitions, *Why* or *How* questions.

Current question answering systems may be divided into two classes (see, for example, [21,6,10,15]):

- Databases – or restricted domain. Most of these systems, (e.g. Masque/SQL [5], Precise [16], Wom [11]), and [6] directly generates SQL queries from the user question, to access a relational database.
- Repository of documents in a corpus or even in the web – Most current QA systems in open domain documents compute their answers by [13,12,6,21]:
  1. analysing the question in order to build a representation of the question; inferring the type of question: *factoid*, *list*, *definition*.
  2. identifying the documents fragments needed to extract the information that may match the answer representation;
  3. ranking the document fragments to choose the best answer that may either be exact or approximate.

QA systems on open domain such as Start [13] or on restricted domains such as Precise [16] or Masque/SQL [4] do not consider clarification dialogues to

disambiguate user questions. In Precise the importance of a clarification dialog is discussed but the feature is not implemented.

Other QA system such as those described in [9,7,14,17] try to disambiguate through a clarification dialogue but this is done regardless of it being relevant or not.

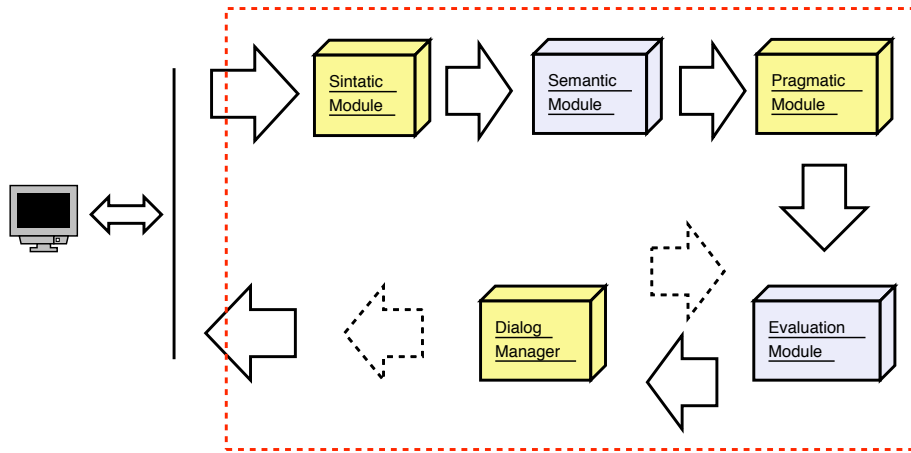
The Wow system [11], is a multilingual QA (Hungarian and English) which translates written genitive relations to SQL as per the V(ISA) algorithm: disambiguation is only necessary if more than one SQL query is generated. The clarification dialogue will decide which SQL query should be used to compute the answer.

Likewise, HITIQA [20] also uses clarification dialogues to disambiguate in order to compute the answer to the question. HITIQA is a frame-based QA system that computes answers in open domains. This system deals with definitions and *why* and *how* questions.

In the next Section, we describe the architecture of our proposal. In section 3 we discuss answer computation as well as the clarification and the justification processes. In section 4 we briefly assess the system and finally in section 5 we summarize our main achievements and discuss future work.

## 2 Dialogue Question Answering System Architecture

The system is composed of five distinct modules connected through well specified APIs, sketched in figure 1. A brief discussion of the various components follows:



**Fig. 1.** System Architecture

---

**The syntactic parser** is chunk-based [1], imposing no strong restrictions on the query sentence structure and leaving all possible interpretation analysis and filtering to the semantic/pragmatic parser.<sup>1</sup>

**Semantic interpretation** is built using First-Order Predicate Logic extended with generalized quantifiers. We take special care with the discourse entities in order to have the appropriate quantifier introduced by the determinant interpretation.

**Pragmatic interpretation** is responsible of transforming the predicates in the semantic representation into ISCO queries.

**the Evaluation** transforms the pragmatic representation into a constraint set of variables annotated with the generalized quantifiers, i.e. a set of discourse entities constrained by predicates extension.

**the Dialogue Manager** deals with the set of discourse entities and is able to compute the question answer, do decide when to relax the question in order to justify the answer and to decide when to clarify a question and how to clarify.

## 2.1 Example

Consider the question *What teachers teach all students?*

**Syntactic analysis** transforms the question into a set of syntactic structures.

A sentence may have more then one syntactic structure.

**Semantic Interpretation** transforms the set of syntactic structures into a set of Discourse Structures with the determiners linked to the discourse entities. The semantics of a question is then represented by the quadruple  $(D_U, D_Q, P_P, P_M)$ , where:

$D_U$  is a list of Discourse entities that must be unique e.g. entities referred by a name such as *John* or *Mary* or by an anaphoric term such as *the teacher*, *he*, etc.

$D_Q$  is a list of Discourse entities with a quantifier, these quantifiers are introduced by the interpretation of noun determiners.

$P_P$  is a logic term representing the question presupposition.

$P_M$  is the sentence main predication.

In the example at hand, we end up with:

$([], [\text{what-Y}, \text{all-X}], (\text{student(X)}, \text{teacher(Y)}), \text{teaches(Y,X)})$

**Pragmatic Interpretation** rewrites the predicates in the semantic representations as ISCO goals, i.e. in our example the **teaches** relation applies to different types of arguments and must be rewritten:

$([], [\text{what-Y}, \text{all-X}], (\text{student(id=X)}, \text{teacher(id=Y)}),$   
 $(\text{teaches(teacher=Y, course=Z)},$   
 $\text{enrolled(student=X, course=Z)}))$

---

<sup>1</sup> The number of possible interpretations is directly influenced by the number of attached propositional phrases (PPs).



---

**Pragmatic Evaluation** transforms the above representation into a constraint satisfaction problem, as  $([], [\text{what-}Y, \text{all-}X], [Y, X], [Y1, X1])$ , where:  $(X, Y)$  are bound by an extensional constraint onto the tuples resulting from `student(id=X), teacher(id=Y)`.  $(Y1, X1)$  are constrained to the extension of `student(id=X1), teacher(id=Y1), teaches(teacher=Y1, course=Z), enroll(student=X1, course=Z)`.

**The Dialogue Manager** receives a set of representations with all variables properly constrained and will decide whether to:

- Answer the question directly.
- Answer the question and justify the answer by answering the relaxed question.
- Initiate a dialogue with user in order to clarify the question.

## 2.2 Porting of our Dialogue QA system

The domain-dependent component of our QA system is adaptable to different domains. In order to do so:

- We need the schema of the target database.
- Each database individual must have a *unique identifier* allowing us to infer its class from the identifier alone. For instance, in the academic services domain, suppose id 123 is used in table `person` in the record (123, John Smith). It cannot be used by any other individual of any type. Therefore, when we see id 123, we know that it is a `person` with the name “John Smith.”

## 3 Interpreting natural language questions

Consider the question *What teachers teach all students?* and a possible representation in first order logic:

$$\{y | \forall x [Student(x) \rightarrow Teaches(y, x)]\}$$

The representation proposed by A. Badia [6] in his *Query Language with Generalized Quantifiers* would be:

$$\{y | all\{x | Student(x)\}, \{x | Teaches(y, x)\}\}$$

Lastly, the SQL translation of the same query:

```
SELECT pid FROM Teaches T
WHERE NOT EXISTS (SELECT sid FROM Student
                  WHERE sid NOT IN
                  (SELECT sid FROM Teaches WHERE pid = T.pid))
```

Even if this query captures the correct answer, we think that our approach is preferable because ours can be built and computed automatically, while this one was built manually.

---

## Dealing with Ambiguous Questions

Our representation for this question is (U, DrefQuant, Lref, Lref1) (see section 2.1). Once we have this, the system is able to answer the user whenever there is a unique solution or to start a dialogue in order to ask for a clarification when there is more than one solution.

**if U is empty** there are no ambiguities in this representation.

**if U is non-empty** it means that there are ambiguities in the question representation, so we must compute all possible answers. The dialogue manager will interact with the user roughly along these lines:

```
dialogueManager(U, Refs, Lref, Lref1, Ans).  
    % find all answers (AnsL)  
    % factorize answers (AnsL into AnsL1)  
    clarify(U, Refs, Lref, Lref1, AnsL, M)).
```

The *factorize* predicate groups similar answers in order to count them.

This above step computes all possible answers, when there are ambiguities such as the names that may identify more than one individual, e.g. the name John may be that of a student that is not aware that there are more than one teacher named John. Another source of ambiguity is that sentence syntactic analysis and pragmatic interpretation that may give rise to more than one sentence representation.

AnsL is the set of all question answers. If all the answers in AnsL are equal, then the system may answer the question, otherwise it must initiate a clarification dialogue with the user.

## Answering a Non-Ambiguous Question

To do so is to interpret the question representation, taking into account the discourse referent quantifiers (DRQ). To compute an answer we define a predicate `answer(DRQ, Lref, Lref1, Ans)`, which we now proceed to illustrate:

1. Interpreting *Who, What, When, ...*:

```
answer([WH-X|R], Lref, Lref1, Ans):-  
    memberIg(X, Lref, Lref1, X1), X #= X1,  
    findall(X, ( fd_labelingff(X),  
                answerYes(R,Lref,Lref1,_) ), Ans).
```

The first line requires that the finite domain variable quantified with WH must be the same in the presupposition and in the main predication, giving the right meaning to the quantifier WH ( $X=X1$ ). The last goal collects all values in the domain of X that satisfy the rest of quantified variables.<sup>2</sup>

---

<sup>2</sup> The predicate `answerYes` is the same as `answer` except that it fails when the answer is No.

---

2. Interpreting *How Many*:

```
answer([howmany-X|R], Lref, Lref1, N):-  
    ...  
    sort(..., L), length(L, N).
```

This definition is identical to the previous one, except that in the end we bind the result with a count of the distinct values of X.

3. Interpreting *All*:

```
answer([all-X|R], Lref, Lref1, yes):-  
    memberIg(X,Lref,Lref1,X1),  
    \+ (fd_labelingff(X), \+ X#=X1),  
    answerYes(R,Lref,Lref1,_).
```

The interpretation of *all X* will give rise to an *Yes* or *No*. The condition to be satisfied is that all the Xs in presupposition are in the X1 of the main predication and the rest of the quantified variables succeed.

4. Interpreting *Exists*:

```
answer([ex-X|R], Lref, Lref1, yes):-  
    memberIg(X,Lref,Lref1,X1),  
    X#=X1, fd_labeling(X),  
    answerN(R,Lref,Lref1,_).
```

To interpret *exists X* it is enough that one value of X in the presupposition be equal to a value of X1 in the main predication.

5. When nothing succeeds the answer is No.

```
answer(_Refs,_Lref,_Lref1,no).
```

### Computing the Justification

When the Dialogue Manager detects that the answer to the question is *No* it will resort to a set of rules for relaxing the query and to compute the question justification.

### Computing the clarification question

When a question has more then one different answer, the dialogue manager looks for a “good” question to ask the user, in order to obtain more information about the Discourse entities. In our framework this can be done in a quite straightforward way:

```
clarify(U, Refs, Lref, Lref1, AnsL, NumAns, Ans) :-  
    choose(AnsL, Cquestion, N),  
    askUser(Cquestion, U),  
    dialogueManager(U, Refs, Lref, Lref1, Ans).
```

---

To choose a question means picking a property of a Discourse referent in  $U$  whose domain size is greater than 1, that will reduce the number of answers. After choosing the property, the system asks the user about the value of the property for that discourse entity. The answer will then constrain the entity, so the dialogue manager will go on with questioning the user, only with a more constrained search space.

The procedure for choosing an entity property requires access to the database schema and is described in [19,18]. We are using a very simple method which gives surprisingly good results. If the user cannot answer the question this procedure will, in backtracking, give the next best choice.

## 4 Evaluation of the Question Answering Dialogue System

In order to evaluate our system we take into account:

**Correctness of the computed answer** – we distinguish three situations:

1. *The system directly answers the question.* We can use the *recall* and *precision* measures to evaluate the system accuracy. In a question-answering system we should enhance the *precision* measure since the user is not interested in having wrong answers. We have tested our system with a set of 200 questions and obtained a 90% precision (% of correct answers). The cases where we obtained wrong answers were due to the construction of a bad question representation. We obtained a 80% recall (% of questions that were answered) mainly due to failures in the pragmatic interpretation, there were words (verbs, nouns, etc) for which we could not assign a database relation.
2. *The systems must relax the question in order to be cooperative.* It answers the question and tries to justify it. For this case the above measures (recall and precision) are not enough. We must take into account the user satisfaction. We still do not have a proper evaluation of this aspect of our system.
3. *The system must clarify the user question.* When a question is to be clarified, the system must pose a pertinent question to the user in order to disambiguate the user question. The problem is that for some user questions, one system question is not enough.

**Time taken to compute an answer** – with our database, the answers never take longer than 10 seconds,<sup>3</sup> when the discourse entities have a large domain. The heavier tasks may take from 0.1 ms to up to a few seconds, depending on the data size necessary for representing the question (not the data size in the database). The steps are:

- Syntactic and semantic analyses: these are very fast, under 10 ms, and are essentially independent from the question.
- Pragmatic analysis: takes up to 1 second depending on the question.

---

<sup>3</sup> All times presented in this section are measured on a Linux VMware virtual machine, running on an Intel Core2 2GHz processor, which also hosts the relational database.

- 
- Data retrieval: this is done using ISCO with a relational database back-end.
  - Constraining of the discourse variables to the extension of the retrieved relations. This is done with the GNU Prolog predicate `fd_relation`.
  - Evaluation of the generalized quantifiers: *All*, *How many*, *more than*, *at least*, etc. The interpretation of some generalized quantifiers has non polynomial time complexity.

From our experimental assessment with this database, the times are acceptable for an interactive question-answering system. The quality of the interaction clearly exceeds that of comparable systems,<sup>4</sup> mostly as a consequence of our reasoning over the entire set of solutions which allows us to make pertinent decisions when asking the user to disambiguate.

Consider the question:<sup>5</sup>

U: All teachers teach at least one course?  
 C: No, only 1311 of the 1502 teachers.

The database has 1502 teachers, 1837 courses and 11541 tuples in relation teaches for the academic year 2008/2009; this database has the actual data from the academic services of Universidade de Évora.

The time taken for the interpretation of the question representation breaks down as follows: 1532 ms for the presupposition, 2396 ms for the main predication, 951 ms for the generalized quantifiers question representation interpretation and 1357 ms for relaxing the question (All is transformed into How many) and the interpretation of the new question representation. The systems takes a total of 6 seconds to answer the user cooperatively.

U: How many teachers from the mathematics department  
 - teach exactly one course?  
 C: 15

For this query, the times taken are 35 ms for the presupposition, 142 ms for the main predication, 37 ms for the generalized quantifiers question representation interpretation. This shows that a question where the universe is restricted is much faster: constraining teachers to the math department (56 people) dramatically reduces the computation time.

U: What teacher teaches more courses?  
 C: José ..... Silva

In this case, we have 1332 ms for the presupposition, 3396 ms for the main predication and 2351 ms for the generalized quantifiers question representation

---

<sup>4</sup> Recall that Badia's system [6] has no working implementation.

<sup>5</sup> User input is annotated with U: while system output is labeled C: and continuations are marked with a "-".

---

interpretation. This question is answered in 7 seconds, an acceptable time for an interactive user.

Consider the following interaction, which includes a clarification dialogue:<sup>6</sup>

```
U: Does Vitor teach Databases?
C: Teacher has Department?
- 1. Chemistry Department
- 2. Biophysical and Landscape Planning Department
- 3. Pedagogy and Education Department
- 4. Informatics Department
U: 4
C: Course has Program?
- 1. Informatics Engineering
- 2. Information Science and Documentation
- 3. Chemistry
- 4. Philosophy
U: 4
C: yes
```

This dialogue could have had different flows depending on the user's answers. If we precompute all possible answers, the average number of questions will be 1 because there is only one Vitor that teaches Databases in the Informatics Department. The remaining alternatives are still presented because the answer may be a "No".

## 5 Conclusions and Future Work

We presented a Dialogue Question Answering System for databases that is able to cooperatively dialogue with a user. The system may justify some answers that are not what the user intended and it may introduce clarification dialogues to disambiguate the user's questions when it is relevant to do so.

We are able to automatically generate a question representation using generalized quantifiers and to correctly compute the answer; the prototype was implemented within a declarative framework using GNU Prolog/CX and ISCO to access databases as Prolog predicates.

We represent the pragmatics of a sentence as a constraint satisfaction problem, where the discourse entities are quantified finite domain variables constrained to the extent of database queries. This sentence modeling and representation enables the reasoning required for answering questions using generalized quantifiers such as: *All*, *How Many*, *at least*, etc. This allows the user to build factoids and list questions.

Compared with our previous work, presented in [19,18], the inclusion of Generalized Quantifiers allowed us to achieve much better results w.r.t. having correct interpretations for short and pertinent dialogues.

---

<sup>6</sup> Note that there are 5 teachers named Vitor and 7 courses with "Databases" in their name.

---

We are not yet able to deal with definition, *Why* or *How* questions. We are presently working:

- To improve the quality of our tools for syntactic, semantic and pragmatic interpretation, aiming at enhancing recall and precision.
- To improve performance: we are pursuing different approaches, namely distribution of the query answering.
- To build and use an *Interpretation Context*. We intend to explore the use of contextual logic programming to represent and use the discourse context, namely to interpret a question's tense and aspect.

## References

1. S. P. Abney. Parsing by chunks. In Robert C. Berwick, Steven P. Abney, and Carol Tenny, editors, *Principle-Based Parsing: Computation and Psycholinguistics*, pages 257–278. Kluwer, Dordrecht, 1991. 2
2. Salvador Abreu and Daniel Diaz. Objective: In minimum context. In Catuscia Palamidessi, editor, *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer, 2003. 1
3. Salvador Abreu and Vitor Nogueira. Using a Logic Programming Language with Persistence and Contexts. In Osamu Takata, Masanobu Umeda, Isao Nagasawa, Naoyuki Tamura, Armin Wolf, and Gunnar Schrader, editors, *Declarative Programming for Knowledge Management, 16th International Conference on Applications of Declarative Programming and Knowledge Management, INAP 2005, Fukuoka, Japan, October 22-24, 2005. Revised Selected Papers.*, volume 4369 of *Lecture Notes in Computer Science*, pages 38–47. Springer, 2006. 1
4. I. Androutsopoulos, G. Ritchie, and P. Thanisch. Masque/sql: An efficient and portable natural language query interface for relational databases. In *Proc. of the Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems IEA/AIE-93*, pages 327–330, Edinburgh, Scotland, 1993. Gordon and Breach. 1
5. Ion Androutsopoulos and Graeme Ritchie. Database interfaces. In R. Dale, H. Moisl, and H. Somers, editors, *Handbook of Natural Language Processing*, pages 209–240. Marcel Dekker Inc., 2000. 1
6. Antonio Badia. Question answering and database querying: Bridging the gap with generalized quantification. *J. Applied Logic*, 5(1):3–19, 2007. 1, 3, 4
7. Marco De Boni. An analysis of clarification dialogue for question answering, 2003. 1
8. D. Diaz. <http://www.gnu.org/software/prolog>, 1999. 1
9. J. Ginzburg. Clarifying utterances, 1998. 1
10. Haiqing Hu, Fuji Ren, Shingo Kuroiwa, and Shuwu Zhang. A question answering system on special domain and the implementation of speech interface. In Alexander F. Gelbukh, editor, *CICLing*, volume 3878 of *Lecture Notes in Computer Science*, pages 458–469. Springer, 2006. 1
11. Zsolt Tivadar Kardkovács. On the transformation of sentences with genitive relations to sql queries. In Andrés Montoyo, Rafael Muñoz, and Elisabeth Métais, editors, *NLDB*, volume 3513 of *Lecture Notes in Computer Science*, pages 10–20. Springer, 2005. 1, 1

- 
12. B. Katz, S. Felshin, D. Yuret, A. Ibrahim, J. Lin, G. Marton, A. McFarland, and B. Temelkuran. Omnibase: Uniform access to heterogeneous data for question answering, 2002. 1
  13. Boris Katz and Jimmy J. Lin. Start and beyond, 2002. 1, 1
  14. Michael F. McTear. Spoken dialogue technology: enabling the conversational user interface. *ACM Comput. Surv.*, 34(1):90–169, 2002. 1
  15. Diego Mollá and José Luis Vicedo. Question answering in restricted domains: An overview. *Comput. Linguist.*, 33(1):41–61, 2007. 1
  16. H. Kautz O. Etzioni and A. Popescu. Towards a theory of natural language interfaces to databases. In *Intelligent User Interfaces (IUI)*, 2003. 1, 1
  17. Matthew Purver, Jonathan Ginzburg, and Patrick Healey. On the means for clarification in dialogue. In R. Smith and J. van Kuppevelt, editors, *Current and New Directions in Discourse and Dialogue*, volume 22 of *Text, Speech and Language Technology*, pages 235–255. Kluwer Academic Publishers, 2003. 1
  18. Luis Quintano and Irene Pimenta Rodrigues. Using a logic programming framework to control database query dialogues in natural language. In Sandro Etalle and Mirosław Truszczyński, editors, *ICLP*, volume 4079 of *Lecture Notes in Computer Science*, pages 406–420. Springer, 2006. 1, 3, 5
  19. Luis Quintano and Irene Pimenta Rodrigues. Question/answering clarification dialogues. In Alexander F. Gelbukh and Eduardo F. Morales, editors, *MICAI*, volume 5317 of *Lecture Notes in Computer Science*, pages 155–164. Springer, 2008. 1, 3, 5
  20. Sharon Small, Nobuyuki Shimizu, Tomek Strzalkowski, and Ting Liu. Hitqa: A data driven approach to interactive question answering: A preliminary report. In Mark T. Maybury, editor, *New Directions in Question Answering*, pages 94–104. AAAI Press, 2003. 1
  21. Ellen M. Voorhees and Lori P. Buckland, editors. *Proceedings of The Seventeenth Text REtrieval Conference, TREC 2008, Gaithersburg, Maryland, USA, November 18-21, 2008*, volume Special Publication 500-277. National Institute of Standards and Technology (NIST), 2008. 1



---

# Extending XQuery for Semantic Web Reasoning<sup>\*</sup>

Jesús M. Almendros-Jiménez

University of Almería, Spain. Email: [jalmen@ual.es](mailto:jalmen@ual.es)

**Abstract.** In this paper we investigate an extension of the XQuery language for querying and reasoning with OWL-style ontologies. The proposed extension incorporates new primitives (i.e. boolean operators) in XQuery for the querying and reasoning with OWL-style triples in such a way that XQuery can be used as query language for the Semantic Web. In addition, we propose a Prolog-based implementation of the extension.

## 1 Introduction

XQuery [36, 11] is a typed functional language devoted to express queries against XML documents. It contains *XPath 2.0* [35] as a sublanguage. *XPath 2.0* supports navigation, selection and extraction of fragments from XML documents. XQuery also includes expressions to construct new XML documents and to join multiple documents.

*Web Ontology Language (OWL)* [37] is a proposal of the *W3C consortium*<sup>1</sup> for ontology modeling. OWL is an ontology language based on the so-called *Description Logic (DL)* [10]. OWL is syntactically layered on the *Resource Description Framework (RDF)* [34], whose underlying model is based on triples. The *RDF Schema (RDFS)* [33] is also an ontology language, enriching RDF with meta-data, however, OWL offers more complex relationships than RDF(S). OWL includes, among others, means to infer that items with various properties are members of a particular class, means to define complex vocabularies: equivalence and inclusion between entities, symmetric, inverse and transitive properties, cardinality restrictions on properties, etc.

In this paper we present an extension of XQuery for querying and reasoning with OWL-style ontologies. Such extension incorporates to XQuery mechanisms for the traversal and reasoning with OWL statements. The main features of our proposal can be summarized as follows:

- XQuery has been developed for querying XML documents, however, Web data can be also represented by means of RDF(S) and OWL. Therefore, XQuery should support the simultaneous querying of XML data by both its structure and by its associated meta-data given in form of OWL-style ontologies. The proposed extension allows to query XML/RDF(S)/OWL documents and to obtain as output the same kind of documents.

---

<sup>\*</sup> This work has been partially supported by the Spanish MICINN under grant TIN2008-06622-C03-03.

<sup>1</sup> <http://www.w3.org>.

- 
- In his current form, XQuery can be used for querying RDF(S) and OWL assuming a XML representation of RDF(S) and OWL. The extension proposed in this paper is independent of the XML encoding of ontologies, working directly on the conceptual RDF(S) (and OWL) data model: triples.
  - RDF(S) and OWL querying should be combined with reasoning. The proposed extension is able to use semantic information inferred from RDF(S) and OWL resources.

In addition, we will propose an implementation of the extension in Prolog. Now, we will review the existing proposals of query languages for the Semantic Web and we will present the advantages of our proposal.

A great effort has been made for defining query languages for RDF(S)/OWL documents (see [6, 18] for surveys about this topic). The proposals mainly fall on extensions of *SQL*-style syntax for handling the *triple-based RDF structure*. In this line the most representative languages are SquishQL [25], SPARQL [14] and RQL [20]. However, they are designed for RDF(S) and OWL querying in the following sense. The syntax of the previous languages resemble SQL extensions for expressing queries in which the database contains RDF triples. The answers to such queries resemble to SQL answers (i.e. tables). In our proposal, the extension of XQuery can express answers as XML documents, and, in particular, as RDF documents, and therefore XML/RDF(S)/OWL documents work as input and as output of the XQuery extension. It does not mean that the existing proposals of RDF(S)/OWL query languages cannot produce XML documents as output but they require the use of *ad-hoc* languages for formatting the output. This is the case, for instance, of SPARQL which incorporates to the queries the so-called *SPARQL Results Document* which defines the XML format of the queries. In our case, we take advantage of the XQuery mechanisms for obtaining XML documents as output.

There exist in the literature some proposals of extensions of *XPath*, *XSLT* and *XQuery* languages for the handling of RDF(S) and OWL. *XPath*, *XSLT* were also designed for XML, and therefore the proposals are *XML-based* approaches. In this line the most representative languages are XQuery for RDF (*‘the Syntactic Web Approach’*) [26], RDF Twig [38], RDFPath [30], RDFT [12] and XsRQL [21]. Such languages assume the serialization (i.e. encoding, representation) of RDF(S) and OWL in XML. However, such serialization is not standard. Our proposal aims to amalgamate the SPARQL (SquishQL and RQL) design with the XML-based approaches in the following sense. In our proposed extension of XQuery we do not assume a fixed serialization of RDF(S) and OWL in XML, rather than, we extend the XQuery syntax in order to admit the traversal of RDF triples similarly to SPARQL-style query languages. However, we retain the XQuery capability to generate XML documents as output. In addition, our proposal extends the syntax of XQuery but XQuery can be still used for expressing queries against XML documents. Therefore, with our extension XQuery can be used for expressing queries which combine input resources which can have XML/RDF(S)/OWL format, any of them. As far as we know, our proposal is the first one in which heterogeneous resources can be combined in a query.

---

Finally, there are some proposals of logic-based query languages for the Semantic Web. This is the case of TRIPLE [28], N3QL [7] and XCerpt [27, 15]. They have their own syntax similar to *deductive logic languages*, and handle RDF(S)/OWL (in the case of TRIPLE and N3QL) and XML/RDF (in the case of XCerpt). Our proposal rather than defining a new query language (i.e. syntax and semantics), aims to define an extension of an standarized query language like XQuery. However, our work is close to logic-based query languages in the following sense. We will propose a logic-based implementation of XQuery using Prolog as host language.

The question now is, why Prolog? The existing XQuery implementations either use functional programming (with *Objective Caml* as host language) or Relational Database Management Systems (RDBMS). In the first case, the *Galax* implementation [24] encodes XQuery into *Objective Caml*, in particular, encodes XPath. Since XQuery is a functional language (with some extensions) the main encoding is related with the type system for allowing XML documents and XPath expressions to occur in a functional expression. With this aim an specific type system for handling XML tags, the hierarchical structure of XML, and sequences of XML items is required. In addition, XPath expressions can implemented from this representation. In the second case, XQuery has been implemented by using a RDBMS. It evolves in most of cases the encoding of XML documents by means of relational tables and the encoding of XPath and XQuery. The most relevant contribution in this research line is *MonetDB/XQuery* [8]. It consists of the *Pathfinder* XQuery compiler [9] on top of the *MonetDB* RDBMS, although *Pathfinder* can be deployed on top of any RDBMS. MonetDB/XQuery encodes the XML tree structure in a relational table following a pre/post order traversal of the tree (with some variant). XPath can be implemented from such table-based representation. XQuery can be implemented by encoding *flwor* expressions into the *relational algebra*, extended with the so-called *loop-lifted staircase join*.

The motivation for using a logic-based language for implementing XQuery is that our extension of XQuery has to handle OWL-style ontologies. However, RDF(S) and OWL should be handled not only as a database of triples, but reasoning with RDF(S) and OWL should be incorporated. The underlying model of RDF(S) and OWL is suitable for reasoning about data and metadata. For instance, class and property (i.e. concepts and roles) hierarchies can define complex models in which a given individual can belong to more than one complex class. Inferencing and reasoning is then required. Logic programming (and in particular, Prolog) is a suitable framework for inferencing and reasoning. It has been already noted by some authors. OWL has been combined and extended with logic rules in some works. The closest to our approach aims to study the intersection of OWL and logic programming, in other words, which fragment of OWL can be expressed in logic programming. In this research line, some authors [16, 32] have defined the so-called *Description Logic Programming*, which is the intersection of logic programming and description logic. Such intersection can be defined by encoding OWL into logic programming. With this aim, firstly, the corresponding fragment of OWL is represented by means of descrip-

---

tion logic, after such fragment of the description logic can be encoded into a fragment of *First Order Logic* (FOL); finally, the fragment of FOL can be encoded into logic programming. Several fragments of OWL/DL can be encoded into logic programming, in particular, Volz [32] has encoded OWL subsets into *Datalog*, *Datalog(=)*, *Datalog(=,IC)* and *Prolog(=,IC)*; where “=” means “with equality”, and “IC” means “with Integrity constraints”. Some recent proposals have encoded description logic fragments into *disjunctive Datalog* [19], into *Datalog(IC,≠,not)* (for OWL-Flight) [13], where “not” means “with negation”, and into *Prolog* [23]. The most relevant implementations of OWL reasoners fully based on logic programming are KAON2 [19, 22] and DLog [23]. Some other approaches are based on tableaux procedures (for instance, *Racer* [17], FaCT++ [31], Pellet [29], among others).

Using Prolog as host language for a XQuery implementation we will have a fully logic based implementation in which accommodate the Semantic Web extension. In previous works [4, 5] we have proposed a Prolog based implementation of the XQuery (and XPath) languages. In such proposal, *XML* documents are translated into a Prolog program by means of facts and rules. Then, XQuery query is executed by automatically encoding the query by means of Prolog rules, and an specific goal is used for obtaining the answer. The rules of the encoding of the query specializes the Prolog program representing the input XML document. From the Prolog computed answers we are able to rebuild the output XML document. Now, both XQuery constructs and RDF(S)/OWL reasoning can be expressed by means of rules and therefore a Prolog based implementation of the extension of XQuery can be easily defined.

With respect to the implementation of RDF(S) and OWL reasoning in Prolog, we have followed the quoted works about the intersection of logic programming and description logic. However, we restrict our framework to the case of a simple kind of OWL ontology, which can be encoded in *Datalog* [32], and therefore possible to encode into Prolog. Therefore the handling of OWL in our framework is restricted to the same conditions as in [32] for the encoding into *Datalog* programs, and therefore decidability and complexity aspects of our approach are based on it. Since the main aim of our approach is to exhibit the combination of querying of XML, RDF and OWL, the restriction to a simple kind of ontology makes our work easier. However, it does not affect substantially to the relevance of the approach once interesting examples can be handled in our query language. We believe that more complex ontologies which can be encoded into extensions of *Datalog* could be also integrated in our approach following the same ideas presented here. Such extensions are considered as future work.

We would like to remark that our work also continues a previous work about XQuery. In [1], we have studied how to define an extension of XQuery for querying RDF documents. Similarly to the current proposal, such extension allows to query XML and RDF resources with XQuery, and queries can take as input XML and RDF documents, producing also as output both kind of documents. The proposed RDF extension of XQuery uses the **for** construction of XQuery for traversing RDF triples. In addition, our RDF extension of XQuery is equipped with

**Fig. 1. TBox and ABox Formulas** (see [32])

$C \sqsubseteq D$	( <code>rdfs:subClassOf</code> )	$E \equiv F$	( <code>owl:equivalentClass</code> )
$P \sqsubseteq Q$	( <code>rdfs:subPropertyOf</code> )	$P \equiv Q$	( <code>owl:equivalentProperty</code> )
$P \equiv Q^-$	( <code>owl:inverseOf</code> )	$P \equiv P^-$	( <code>owl:symmetricProperty</code> )
$P^+ \sqsubseteq P$	( <code>owl:TransitiveProperty</code> )	$\top \sqsubseteq \forall P^-.D$	( <code>rdfs:domain</code> )
$\top \sqsubseteq \forall P.D$	( <code>rdfs:range</code> )	$P(a, b)$	(property fillers)
$D(a)$	(individual assertions)		

built-in boolean operators for RDF/RDFS properties like *rdf:type*, *rdfs:domain*, *rdfs:range*, *rdfs:subClassOf*, *rdfs:subPropertyOf*, and so on. Such built-in boolean operators can be used for reasoning about RDF, that is, for using the RDFS entailment relationship in the queries. In addition, we have studied a Prolog based implementation of this extension.

Finally, we have tested the proposal of this paper by means of a prototype implemented in SWI-Prolog. The prototype consists of a XQuery implementation and a RDF(S)/OWL reasoning module. The XQuery implementation is described in [2] and the RDF(S)/OWL reasoning module is described in [3]. The proposal of this paper amalgamates both implementations in a XQuery/RDF(S)/OWL querying/reasoning tool. The *XQuery* implementation can be downloaded from <http://indalog.ual.es/XQuery> and the RDF(S)/OWL module from <http://indalog.ual.es/0Prolog>.

The structure of the paper is as follows. Section 2 will present the kind of ontologies we consider in our framework. Section 3 will describe the extension of XQuery for RDF(S)/OWL. Section 4 will show the implementation in Prolog and, finally, Section 5 will conclude and present future work.

## 2 Ontology Representation

In this section we will show which kind of ontologies will be handled in our framework. We will restrict ourselves to the case of a subset of DL expressible in Datalog [32], and therefore in Prolog. Such kind of DL ontologies contains a **TBox** and a **ABox** of axioms whose syntax is shown in Figure 1. In Figure 1,  $E, F$  are class descriptions of type  $\mathcal{E}$  (see Figure 2),  $C$  is a class description of left-hand side type (type  $\mathcal{L}$ , see Figure 2), and  $D$  is a class description of right-hand side type (type  $\mathcal{R}$ , see Figure 2),  $P, Q$  are property names and  $a, b$  are individual names.

Basically, the proposed subset of DL restricts the form of class descriptions in right and left hand sides of subclass and class equivalence axioms, and in individual assertions. Such restriction is required to be encoded by means of Datalog rules. Roughly speaking, the universal quantification is only allowed in the right hand side of DL formulas, which corresponds in the encoding to the occurrences of the same quantifier in the left hand side (i.e. head) of rules. The same kind of reasoning can be used for explaining why existential quantifiers can only occur in left hand sides of DL formulas. Union formulas are required to appear in left hand sides which corresponds with the definition of two or more rules in the encoding. The details about the encoding can be found in Section 4.1. Let us see now an example of a such DL ontology (see Figure 3).

The ontology of Figure 3 describes, in the **TBox**, meta-data in which the elements of *Person* are elements of *Man* or elements of *Woman* (cases (1) and

**Fig. 2.** Allowed Types (see [32])

type $\mathcal{E}$		type $\mathcal{L}$		type $\mathcal{R}$	
$A$	atomic class	$A$	atomic class	$A$	atomic class
$E_1 \sqcap E_2$	$\text{owl:intersectionOf}$	$C_1 \sqcap C_2$	$\text{owl:intersectionOf}$	$D_1 \sqcap D_2$	$\text{owl:intersectionOf}$
$\exists P.\{o\}$	$\text{owl:hasValue}$	$\exists P.\{o\}$	$\text{owl:hasValue}$	$\exists P.\{o\}$	$\text{owl:hasValue}$
		$C_1 \sqcup C_2$	$\text{owl:unionOf}$	$\forall P.D$	$\text{owl:allValuesFrom}$
		$\exists P.C$	$\text{owl:someValuesFrom}$		

**Fig. 3.** An Example of Ontology

TBox	
(1) $Man \sqsubseteq Person$	(2) $Woman \sqsubseteq Person$
(3) $Person \sqcap \exists author\_of.Manuscript \sqsubseteq Writer$	(4) $Paper \sqcup Book \sqsubseteq Manuscript$
(5) $Book \sqcap \exists topic.\{“XML”\} \sqsubseteq XMLbook$	(6) $Manuscript \sqcap \exists reviewed\_by.Person \sqsubseteq Reviewed$
(7) $Manuscript \sqsubseteq \forall rating.Score$	(8) $Manuscript \sqsubseteq \forall topic.Topic$
(9) $author\_of \equiv writes$	(10) $average\_rating \sqsubseteq rating$
(11) $authored\_by \equiv author\_of^-$	(12) $\top \sqsubseteq \forall author\_of.Manuscript$
(13) $\top \sqsubseteq \forall author\_of^-.Person$	(14) $\top \sqsubseteq \forall reviewed\_by.Person$
(15) $\top \sqsubseteq \forall reviewed\_by^-.Manuscript$	
ABox	
(1) $Man(“Abiteboul”)$	(3) $Man(“Suciu”)$
(2) $Man(“Buneman”)$	(5) $Book(“XML in Scotland”)$
(4) $Book(“Data on the Web”)$	(7) $Person(“Anonymous”)$
(6) $Paper(“Growing XQuery”)$	(9) $authored\_by(“Data on the Web”, “Buneman”)$
(8) $author\_of(“Abiteboul”, “Data on the Web”)$	(11) $author\_of(“Buneman”, “XML in Scotland”)$
(10) $author\_of(“Suciu”, “Data on the Web”)$	(13) $reviewed\_by(“Data on the Web”, “Anonymous”)$
(12) $writes(“Simeon”, “Growing XQuery”)$	(15) $average\_rating(“Data on the Web”, “good”)$
(14) $reviewed\_by(“Growing”, “Almendros”)$	(17) $average\_rating(“Growing XQuery”, “good”)$
(16) $rating(“XML in Scotland”, “excellent”)$	(19) $topic(“Data on the Web”, “Web”)$
(18) $topic(“Data on the Web”, “XML”)$	
(20) $topic(“XML in Scotland”, “XML”)$	

(2)); and the elements of *Paper* and *Book* are elements of *Manuscript* (case (4)). In addition, a *Writer* is a *Person* who is the *author\_of* a *Manuscript* (case (3)), and the class *Reviewed* contains the elements of *Manuscript* reviewed\_by a *Person* (case (6)). Moreover, the *XMLBook* class contains the elements of *Manuscript* which have as *topic* the value “XML” (case (5)). The classes *Score* and *Topic* contain, respectively, the values of the properties *rating* and *topic* associated to *Manuscript* (cases (7) and (8)). The property *average\_rating* is a subproperty of *rating* (case (10)). The property *writes* is equivalent to *author\_of* (case (9)), and *authored\_by* is the inverse property of *author\_of* (case (11)). Finally, the property *author\_of*, and conversely, *reviewed\_by*, has as domain a *Person* and as range a *Manuscript* (cases (12)-(15)).

The **ABox** describes data about two elements of *Book*: “Data on the Web” and “XML in Scotland” and a *Paper*: “Growing XQuery”. It describes the *author\_of* and *authored\_by* relationships for the elements of *Book* and the *writes* relation for the elements of *Paper*. In addition, the elements of *Book* and *Paper* have been reviewed and rated, and they are described by means of a topic.

### 3 Extended XQuery

Now, we would like to show how to query an OWL-style ontology by means of XQuery. The grammar of the extended XQuery for querying XML, RDF(S) and OWL is described in Figure 4, where “name : resource” assigns name spaces to URL resources; “value” can be URLs / URIs, strings, numbers or XML trees; “tag” elements are XML labels; “att” elements are attribute names; “doc” elements are URLs; and finally, *Op* elements can be selected from the usual binary

**Fig. 4.** Extension of the core of XQuery

---

```

xquery:= namespace name : resource in xquery | flwr | value
        | dexpr | < tag att = vexpr, ..., att = vexpr > '{ xquery, ..., xquery }' < /tag > .
dexpr:= document(doc) '/' expr.          rdfdoc := rdfdocument(doc).
owldoc := owldocument(doc).             tripledoc:= rdfdoc | owldoc.
flwr:= for $var in vexpr [where constraint] return xqvar
        | for ($var,$var,$var) in tripledoc [where constraint] return xqvar
        | let $var := vexpr [where constraint] return xqvar.
xqvar:= vexpr | < tag att = vexpr, ..., att = vexpr > '{ xqvar, ..., xqvar }' < /tag >
        | flwr | value.
vexpr:= $var | $var '/' expr | dexpr | value.      expr:= text() | @att | tag | tag[expr] | '/' expr.
constraint := Op(vexpr, ..., vexpr) | constraint 'or' constraint | constraint 'and' constraint.

```

---

operators:  $\leq$ ,  $\geq$ ,  $<$ ,  $>$ ,  $=$ ,  $\neq$ , and from *OWL/RDF(S)* built-in boolean operators. Basically, the XQuery language has been extended as follows:

- The **namespace** statement has been added allowing the declaration of URIs.
- A new kind of **for** expression has been added for traversing triples from a RDF document whose location is specified by means of the **rdfdocument** primitive; analogously, a new kind of **for** expression has been added for traversing triples from an OWL document whose location is specified by means of the **owldocument** primitive.
- In addition, the **where** construction includes boolean conditions of the form  $Op(vexpr, \dots, vexpr)$  which can be used for checking RDF(S)/OWL properties. The boolean operator  $Op$  can be one of *rdf:type*, *rdfs:subClassOf*, *owl:equivalentClass*, etc.

The above XQuery is a typed language in which there are two kinds of variables: those variables used in XPath expressions, and those used in RDF(S)/OWL triples. However they can be compared by means of boolean expressions, and they can be used together for the construction of the answer. We have considered a subset of the XQuery language in which some other built-in constructions for XPath can be added, and also it can be enriched with other XQuery constructions, following the W3C recommendations [36]. However, with this small extension of the core of XQuery we are able to express interesting queries against XML, RDF and OWL documents.

Now, we would like to show an example of query in order to give us an idea about how the proposed extension of XQuery is suitable for OWL querying and reasoning. The query we like to show is “Retrieve the authors of manuscripts”. It can be expressed in our proposed extension of XQuery as follows:

---

```

< list > {
for ($Author,$Property,$Manuscript) in owldocument("ex.owl") return
for ($Manuscript,$Property2,$Type) in owldocument("ex.owl")
where rdfs:subPropertyOf($Property,author_of)
and $Property2=rdf:typeOf and rdfs:subClassOf($Type,manuscript) return
< author>{ $Author } </ author >
} </ list >

```

---

In the example we can see the following elements:

- OWL triples are traversed by means of the new **for** expression of XQuery. Each triple is described by means of three variables (prefixed with '\$' as usual

- 
- in XQuery). Given that we have to query two properties, that is, *rdf:typeOf* and *author\_of*, we have to combine two **for** expressions.
- The **where** expression has to check whether the first property, that is, *\$Property*, has to be a subproperty of *author\_of*, and the second property has to be *rdf:typeOf*. Let us remark that we could write *\$Property=author\_of* instead of *rdfs:subPropertyOf(\$Property,author\_of)* but in such a case only triples “*author\_of*” would be considered, and not subproperties of “*author\_of*”.
  - The type of the manuscript should be “*Manuscript*”, and it is checked by means of *rdfs:subClassOf(\$Type,manuscript)*.
  - Finally, the output of the query is shown by means of XML in which each element (i.e. the author) is labeled by means of “*author*”.

In this case the answer would be:

---

```
<list>
<author>Abiteboul</author>
<author>Suciu</author>
<author>Buneman</author>
<author>Buneman</author>
<author>Simeon</author>
</list>
```

---

Let us remark that our proposed query language is able to reason with OWL, that is, it uses that *Book* and *Paper* are subclasses of *Manuscript* and the relationship *author\_of* is equivalent to *writes*.

## 4 Prolog Implementation

In this section, we will propose the implementation of the extension of XQuery in Prolog. Firstly, we will describe the encoding of the subset of OWL into Prolog. Secondly, we will show the encoding of XQuery in Prolog. The encoding of the extension of XQuery will use the encoding of XPath studied in [5], and the encoding of XQuery studied in [4].

### 4.1 Ontology Encoding

The encoding of the ontologies defined in Section 2 consists of Prolog facts and rules. Facts are used for representing a given ontology instance. Rules are used for representing the ontology reasoning.

**1. Ontology Instance Encoding:** The encoding introduces Prolog facts about a predicate called *triple*. There is one fact for each element of the ontology instance. The encoding of DL formulas is as follows. Class and property names are represented by means of Prolog atoms. Quantified formulas are represented by means of Prolog terms, that is,  $\forall P.C$ ,  $\exists P.C$  and  $\exists P.\{o\}$  are represented by means of Prolog term *forall*(*p*, *c*), *exists*(*p*, *c*) and *hasvalue*(*p*, *o*), respectively. Unions (i.e.  $C \sqcup D$ ) and intersections (i.e.  $C \sqcap D$ ) are also represented as *union*(*c*, *d*) and *inter*(*c*, *d*), respectively. Inverse and transitivity properties (i.e.  $P^-$  and  $P^+$ ) are represented as Prolog terms: *inv*(*P*) and *trans*(*P*). Finally OWL relationships: *rdfs:subClassOf*, *rdf:type*, etc are represented as atoms in Prolog. In the running example, we will have the facts of Figure 5.

**2. Encoding for OWL Reasoning:** Now, the second element of the encoding consists of Prolog rules defining how to reason about OWL properties. Such



**Fig. 5.** Representation of Ontology instances

```
triple(man, rdfs:subClassOf, person).
triple(woman, rdfs:subClassOf, person).
triple(inter(person, exists(author_of, manuscript)), rdfs:subClassOf, writer).
triple(union(paper, book), rdfs:subClassOf, manuscript).
triple(inter(book, exists(topic, "XML")), rdfs:subClassOf, xmlbook).
triple(inter(manuscript, exists(reviewed_by, person)), rdfs:subClassOf, reviewed).
triple(manuscript, rdfs:subClassOf, forall(rating, score)).
triple(manuscript, rdfs:subClassOf, forall(topic, topic)).
triple(author_of, owl:equivalentProperty, writes).
triple(authored_by, owl:equivalentProperty, inv(author_of)).
triple(average_rating, rdfs:subPropertyOf, rating).
triple(thing, rdfs:subClassOf, forall(author_of, manuscript)).
triple(thing, rdfs:subClassOf, forall(inv(author_of), person)).
triple(thing, rdfs:subClassOf, forall(reviewed_by, person)).
triple(thing, rdfs:subClassOf, forall(inv(reviewed_by), manuscript)).
triple("Abiteboul", rdf:type, man).
triple("Buneman", rdf:type, man).
triple("Suciu", rdf:type, man).
triple("Data on the Web", rdf:type, book).
triple("XML in Scotland", rdf:type, book).
triple("Growing XQuery", rdf:type, paper).
triple("Anonymous", rdf:type, person).
triple("Abiteboul", author_of, "Data on the Web").
triple("Data on the Web", authored_by, "Buneman").
...
```

**Fig. 6.** Triple-based encoding of DL in FOL

$fol^t(C \sqsubseteq D) = \forall x. fol_x^t(C) \rightarrow fol_x^t(D)$ $fol^t(E \equiv F) = \forall x. fol_x^t(E) \leftrightarrow fol_x^t(F)$ $fol^t(P \sqsubseteq Q) = \forall x, y. triple(x, p, y) \rightarrow triple(x, q, y)$ $fol^t(P \equiv Q) = \forall x, y. triple(x, p, y) \leftrightarrow triple(x, q, y)$ $fol^t(P \sqsubseteq Q^-) = \forall x, y. triple(x, p, y) \leftrightarrow triple(y, q, x)$ $fol^t(P \equiv P^-) = \forall x, y. triple(x, p, y) \leftrightarrow triple(y, p, x)$ $fol^t(P^+ \sqsubseteq P) = \forall x, y, z. triple(x, p, y) \wedge triple(y, p, z) \rightarrow triple(x, p, z)$ $fol^t(\sqcap \sqsubseteq \forall P.C) = \forall x. fol_x^t(\forall P.C)$ $fol^t(\sqcap \sqsubseteq \forall P^-.C) = \forall x. fol_x^t(\forall P^-.C)$	$fol_x^t(A) = triple(x, rdf : type, A)$ $fol_x^t(C \sqcap D) = fol_x^t(C) \wedge fol_x^t(D)$ $fol_x^t(C \sqcup D) = fol_x^t(C) \vee fol_x^t(D)$ $fol_x^t(\exists P.C) = \exists y. triple(x, p, y) \wedge fol_y^t(C)$ $fol_x^t(\forall P.C) = \forall y. triple(x, p, y) \rightarrow fol_y^t(C)$ $fol_x^t(\forall P^-.C) = \forall y. triple(y, p, x) \rightarrow fol_y^t(C)$ $fol_x^t(\exists P.\{o\}) = \exists y. triple(x, p, y) \wedge y = o$
---	--

rules express the semantic information deduced from a given ontology instance. Such rules infer new relationships between the data in the form of triples. Therefore new Prolog terms make true the predicate *triple*. For instance, new triples for *rdf:type* are defined from the *rdfs:subClassOf* and previously inferred *rdf:type* relationships. In order to define the encoding, we have to follow the encoding of DL into FOL, which is described in Figure 6. Such encoding is based on a representation by means of triples of the DL formulas. In such encoding, class and property names are considered as constants in FOL. The encoding of OWL reasoning by means of rules is shown in Figure 7. The rules from **Eq1** to **Eq3** handle inference about the reflexive, symmetric and transitive relationship of equivalence. The rules from **Sub1** to **Sub14** handle inference about subclasses. Cases from **Sub3** to **Sub7** define new subclass relationships from the already defined subclass relationships and union and intersection operators. Cases from **Sub8** to **Sub13** define new subclass relationships for complex formulas. The rules **Type1** to **Type8** infer type relationships using subclass and equivalence relationships. The most relevant are the cases from **Type5** to **Type7** defining the meaning of complex formulas with regard to individuals. Finally, the rules **Prop1** to **Prop10** infer relationships about roles. The most relevant are the

**Fig. 7.** Rules for RDF(S)/OWL Reasoning

Rule Name	Prolog Rules
(Eq1)	$\text{triple}(E, \text{owl} : \text{equivalentClass}, E) : \neg \text{class}(E).$
(Eq2)	$\text{triple}(E, \text{owl} : \text{equivalentClass}, G) : \neg \text{triple}(E, \text{owl} : \text{equivalentClass}, F),$ $\text{triple}(F, \text{owl} : \text{equivalentClass}, G).$
(Eq3)	$\text{triple}(E, \text{owl} : \text{equivalentClass}, F) : \neg \text{triple}(F, \text{owl} : \text{equivalentClass}, E).$
(Sub1)	$\text{triple}(E, \text{rdfs} : \text{subClassOf}, F) : \neg \text{triple}(E, \text{owl} : \text{equivalentClass}, F).$
(Sub2)	$\text{triple}(C, \text{rdfs} : \text{subClassOf}, E) : \neg \text{triple}(C, \text{rdfs} : \text{subClassOf}, D),$ $\text{triple}(D, \text{rdfs} : \text{subClassOf}, E).$
(Sub3-I)	$\text{triple}(D, \text{rdfs} : \text{subClassOf}, E) : \neg \text{triple}(\text{union}(C, D), \text{rdfs} : \text{subClassOf}, E).$
(Sub3-II)	$\text{triple}(C, \text{rdfs} : \text{subClassOf}, E) : \neg \text{triple}(\text{union}(C, D), \text{rdfs} : \text{subClassOf}, E).$
(Sub4-I)	$\text{triple}(E, \text{rdfs} : \text{subClassOf}, C) : \neg \text{triple}(E, \text{rdfs} : \text{subClassOf}, \text{inter}(C, D)).$
(Sub4-II)	$\text{triple}(E, \text{rdfs} : \text{subClassOf}, D) : \neg \text{triple}(E, \text{rdfs} : \text{subClassOf}, \text{inter}(C, D)).$
(Sub5)	$\text{triple}(\text{inter}(E, C_2), \text{rdfs} : \text{subClassOf}, D) : \neg$ $\text{triple}(\text{inter}(C_1, C_2), \text{rdfs} : \text{subClassOf}, D),$ $\text{triple}(E, \text{rdfs} : \text{subClassOf}, C_1).$
(Sub6)	$\text{triple}(\text{union}(E, C_2), \text{rdfs} : \text{subClassOf}, D) : \neg$ $\text{triple}(\text{union}(C_1, C_2), \text{rdfs} : \text{subClassOf}, D),$ $\text{triple}(E, \text{rdfs} : \text{subClassOf}, C_1).$
(Sub7)	$\text{triple}(C, \text{rdfs} : \text{subClassOf}, \text{inter}(E, D_2)) : \neg$ $\text{triple}(C, \text{rdfs} : \text{subClassOf}, \text{inter}(D_1, D_2)),$ $\text{triple}(D_1, \text{rdfs} : \text{subClassOf}, E).$
(Sub8)	$\text{triple}(\text{hasvalue}(Q, O), \text{rdfs} : \text{subClassOf}, D) : \neg \text{triple}(Q, \text{owl} : \text{subPropertyOf}, P),$ $\text{triple}(\text{hasvalue}(P, O), \text{rdfs} : \text{subClassOf}, D).$
(Sub9)	$\text{triple}(\text{exists}(Q, C), \text{rdfs} : \text{subClassOf}, D) : \neg \text{triple}(Q, \text{owl} : \text{subPropertyOf}, P),$ $\text{triple}(\text{exists}(P, C), \text{rdfs} : \text{subClassOf}, D).$
(Sub10)	$\text{triple}(\text{exists}(P, E), \text{rdfs} : \text{subClassOf}, D) : \neg \text{triple}(E, \text{rdfs} : \text{subClassOf}, C),$ $\text{triple}(\text{exists}(P, C), \text{rdfs} : \text{subClassOf}, D).$
(Sub11)	$\text{triple}(C, \text{rdfs} : \text{subClassOf}, \text{hasvalue}(Q, O)) : \neg \text{triple}(P, \text{owl} : \text{subPropertyOf}, Q),$ $\text{triple}(C, \text{rdfs} : \text{subClassOf}, \text{hasvalue}(P, O)).$
(Sub12)	$\text{triple}(C, \text{rdfs} : \text{subClassOf}, \text{forall}(Q, D)) : \neg \text{triple}(Q, \text{owl} : \text{subPropertyOf}, P),$ $\text{triple}(C, \text{rdfs} : \text{subClassOf}, \text{forall}(P, D)).$
(Sub13)	$\text{triple}(C, \text{rdfs} : \text{subClassOf}, \text{forall}(P, E)) : \neg \text{triple}(D, \text{rdfs} : \text{subClassOf}, E),$ $\text{triple}(C, \text{rdfs} : \text{subClassOf}, \text{forall}(P, D)).$
(Sub14)	$\text{triple}(C, \text{owl} : \text{subClassOf}, \text{owl} : \text{thing}) : \neg \text{class}(C).$
(Type1)	$\text{triple}(A, \text{rdf} : \text{type}, D) : \neg \text{triple}(C, \text{rdfs} : \text{subClassOf}, D), \text{triple}(A, \text{rdf} : \text{type}, C).$
(Type2)	$\text{triple}(A, \text{rdf} : \text{type}, E) : \neg \text{triple}(\text{inter}(C, D), \text{rdfs} : \text{subClassOf}, E),$ $\text{triple}(A, \text{rdf} : \text{type}, C), \text{triple}(A, \text{rdf} : \text{type}, D).$
(Type3)	$\text{triple}(A, \text{rdf} : \text{type}, E) : \neg \text{triple}(\text{union}(C, D), \text{owl} : \text{subClassOf}, E),$ $\text{triple}(A, \text{rdf} : \text{type}, C).$
(Type4-I)	$\text{triple}(A, \text{rdf} : \text{type}, C) : \neg \text{triple}(A, \text{rdf} : \text{type}, \text{inter}(C, D)).$
(Type4-II)	$\text{triple}(A, \text{rdf} : \text{type}, D) : \neg \text{triple}(A, \text{rdf} : \text{type}, \text{inter}(C, D)).$
(Type5)	$\text{triple}(A, \text{rdf} : \text{type}, \text{exists}(P, C)) : \neg \text{triple}(B, \text{rdf} : \text{type}, C), \text{triple}(A, P, B).$
(Type6)	$\text{triple}(A, \text{rdf} : \text{type}, \text{hasvalue}(P, O)) : \neg \text{triple}(A, P, O), \text{individual}(O).$
(Type7)	$\text{triple}(B, \text{rdf} : \text{type}, D) : \neg \text{triple}(A, P, B), \text{triple}(A, \text{rdf} : \text{type}, \text{forall}(P, D)).$
(Type8)	$\text{triple}(A, \text{rdf} : \text{type}, \text{owl} : \text{thing}) : \neg \text{individual}(A).$
(Prop1)	$\text{triple}(P, \text{owl} : \text{equivalentProperty}, P) : \neg \text{property}(P).$
(Prop2)	$\text{triple}(P, \text{owl} : \text{equivalentProperty}, R) : \neg \text{triple}(P, \text{owl} : \text{equivalentProperty}, Q),$ $\text{triple}(Q, \text{owl} : \text{equivalentProperty}, R).$
(Prop3)	$\text{triple}(P, \text{owl} : \text{equivalentProperty}, Q) : \neg \text{triple}(Q, \text{owl} : \text{equivalentProperty}, P).$
(Prop4)	$\text{triple}(P, \text{rdfs} : \text{subPropertyOf}, Q) : \neg \text{triple}(P, \text{owl} : \text{equivalentProperty}, Q).$
(Prop5)	$\text{triple}(A, Q, B) : \neg \text{triple}(P, \text{rdfs} : \text{subPropertyOf}, Q), \text{triple}(A, P, B).$
(Prop6)	$\text{triple}(B, Q, A) : \neg \text{triple}(P, \text{rdfs} : \text{subPropertyOf}, \text{inv}(Q)), \text{triple}(A, P, B).$
(Prop7)	$\text{triple}(A, P, C) : \neg \text{triple}(\text{trans}(P), \text{rdfs} : \text{subPropertyOf}, P),$ $\text{triple}(A, P, B), \text{triple}(B, P, C).$
(Prop7)	$\text{triple}(A, P, O) : \neg \text{triple}(A, \text{rdf} : \text{type}, \text{hasvalue}(P, O)).$
(Prop8)	$\text{triple}(P, \text{rdfs} : \text{subPropertyOf}, \text{inv}(R)) : \neg \text{triple}(P, \text{rdfs} : \text{subPropertyOf}, \text{inv}(Q)),$ $\text{triple}(Q, \text{rdfs} : \text{subPropertyOf}, R).$
(Prop9)	$\text{triple}(\text{inv}(R), \text{rdfs} : \text{subPropertyOf}, P) : \neg \text{triple}(\text{inv}(Q), \text{rdfs} : \text{subPropertyOf}, P),$ $\text{triple}(R, \text{rdfs} : \text{subPropertyOf}, Q).$
(Prop10)	$\text{triple}(\text{trans}(R), \text{rdfs} : \text{subPropertyOf}, P) : \neg \text{triple}(\text{trans}(Q), \text{rdfs} : \text{subPropertyOf}, P),$ $\text{triple}(R, \text{rdfs} : \text{subPropertyOf}, Q).$

cases **Prop6** about the inverse of a property and the case **Prop7** about a transitive property. The rules of Figure 7 are the basis of the RDF(S)/OWL reason-

**Fig. 8.** A Subset of the Core XQuery

---

```

xquery := namespace name : resource in xquery | flwr | value
        | < tag att = vexpr, ..., att = vexpr > '{ xquery, ..., xquery }' < /tag >.
owldoc := owldocument(doc).          vexpr := $var | value.
flwr := for ($var, $var, $var) in owldoc [where constraint] return xqvar.
xqvar := vexpr | < tag att = vexpr, ..., att = vexpr > '{ xqvar, ..., xqvar }' < /tag >
        | flwr | value.
constraint := Op(vexpr, ..., vexpr) | constraint 'or' constraint | constraint 'and' constraint.

```

---

ning module which has been implemented in SWI-Prolog and can be downloaded from <http://indalog.ual.es/0Prolog>.

The proposed encoding allows to use Prolog as inference engine for OWL. For instance, the triple *triple*("Data on the Web", *rdf:type*, *reviewed*) is deduced from the following facts:

---

```

triple("Data on the Web",rdf:type,book).          triple(book,rdfs:subClassOf,manuscript).
triple("Data on the Web",reviewed_by,"Anonymous"). triple("Anonymous",rdf:type,person).
triple(inter(manuscript,exists(reviewed_by,person)),rdfs:subClassOf,reviewed).

```

---

and the rules (**Type1**), (**Type2**) and (**Type5**).

Now, we would like to show how the extended XQuery language can be encoded in Prolog, in such a way that the queries formulated in our proposed query language can be executed under Prolog. In order to make the paper self-contained we will restrict to a fragment of the extended XQuery for querying and reasoning with OWL triples and for generating XML documents as output. The encoding of the full version of extension of XQuery has to combine in a uniform way the encoding of [4] for XML querying and the encoding of [1] for RDF querying and reasoning. Now, the fragment to be encoded will consist in the grammar of the Figure 8, where *Op* can be one of the OWL built-in boolean operators. The reader can check that such fragment has been used in the example of the previous section. Now, a crucial point is the encoding of XML documents in Prolog. Such encoding will allow the encoding of the output of the query.

## 4.2 Encoding of XML Documents

In this section we will show an example of encoding of XML documents. A formal and complete definition can be found in [5]. Let us consider the following document called "books.xml":

---

```

<books>
<book year="2003">
  <author>Abiteboul</author>
  <author>Buneman</author>
  <author>Suciu</author>
  <title>Data on the Web</title>
  <review>A <em>fine</em> book.</review>
</book>
<book year="2002">
  <author>Buneman</author>
  <title>XML in Scotland</title>
  <review><em>The <em>best</em> ever!</em></review>
</book> </books>

```

---

Now, it can be represented by means of a Prolog program as in Figure 9. In our XML encoding we can distinguish the so-called *schema rules* which define the structure of the XML document, and *facts* which store the leaves of the XML tree (and therefore the values of the XML document). Each tag is translated into a predicate name: *books*, *book*, etc. Each predicate has four arguments. The first one, used for representing the XML document structure, is encapsulated into a function symbol with the same name as the tag adding the suffix *type*. Therefore,

**Fig. 9.** Encoding of XML documents

Rules (Schema):	Facts (Document):
<pre> books(bookstype(Book, []), NBooks, 1, Doc) :-   book(Book, [NBook NBooks], 2, Doc). book(booktype(Author, Title, Review,   [year=Year]), NBook, 2, Doc) :-   author(Author, [NAu NBook], 3, Doc),   title(Title, [NTitle NBook], 3, Doc),   review(Review, [NRe NBook], 3, Doc),   year(Year, NBook, 3, Doc). review(reviewtype(Un, Em, []), NReview, 3, Doc) :-   unlabeled(Un, [NUn NReview], 4, Doc),   em(Em, [NEm NReview], 4, Doc). review(reviewtype(Em, []), NReview, 3, Doc) :-   em(Em, [NEm NReview], 5, Doc). em(emtype(Unlabeled, Em, []), NEms, 5, Doc) :-   unlabeled(Unlabeled, [NUn NEms], 6, Doc),   em(Em, [NEm NEms], 6, Doc). </pre>	<pre> year('2003', [1, 1], 3, "books.xml"). author('Abiteboul', [1, 1, 1], 3, "books.xml"). author('Buneman', [2, 1, 1], 3, "books.xml"). author('Suciu', [3, 1, 1], 3, "books.xml"). title('Data on the Web', [4, 1, 1], 3, "books.xml"). unlabeled('A', [1, 5, 1, 1], 4, "books.xml"). em('fine', [2, 5, 1, 1], 4, "books.xml"). unlabeled('book.', [3, 5, 1, 1], 4, "books.xml"). year('2002', [2, 1], 3, "books.xml"). author('Buneman', [1, 2, 1], 3, "books.xml"). title('XML in Scotland', [2, 2, 1], 3, "books.xml"). unlabeled('The', [1, 1, 3, 2, 1], 6, "books.xml"). em('best', [2, 1, 3, 2, 1], 6, "books.xml"). unlabeled('ever!', [3, 1, 3, 2, 1], 6, "books.xml"). </pre>

we have *bookstype*, *booktype*, etc. The second argument is used for numbering each node (a list of natural numbers identifying each node); the third argument of the predicates is used for numbering each type (a natural number identifying each type); and the last argument represents the document name. The key element of our encoding is to be able to recover the original XML document from the set of rules and facts. The encoding has the following peculiarities. In order to specify the order of an XML document in a fact based representation, each fact is numbered (from left to right and by levels in the XML tree). In addition, the hierarchical structure of the XML records is described by means of the identifier of each fact: the length of the numbers of the children is larger than the number of the parent. The type number makes possible to map schema rules with facts.

### 4.3 Encoding of XQuery

Now, we will show how to encode the proposed extension of XQuery. We can summarize the encoding as follows. The encoding combines the schema rules of the output document with rules which call the triple predicates. A new predicate is defined called *join*. The join predicate calls to triple predicate in order to make the join of the triples. Now, let us see an example of query in our proposal and the corresponding encoding. Let us suppose the query “Retrieve the authors of manuscripts” which is defined in XQuery as follows:

```

< list > {
  for ($Author,$Property,$Manuscript) in owldocument("ex.owl") return
  for ($Manuscript,$Property2,$Type) in owldocument("ex.owl")
  where rdfs:subPropertyOf($Property,author_of)
  and $Property2=rdf:typeOf and rdfs:subClassOf($Type,manuscript) return
  <author>{ $Author } </author>
} </ list >

```

Now, the encoding is as follows:

```

(1) list(listtype(Author,[]), NList, 1, Doc) :- author(Author, [NAuthor|NList], 2, Doc).
(2) author(authortype(Author,[]), NAuthor, 2, "result.xml") :- join(Author, NAuthor).
(3) join(Author, [NAuthor, [1]]) :- triple(Author, Property, Manuscript, NAuthor, "ex.owl"),
  triple(Manuscript, Property2, Type, -, "ex.owl"),
  triple(Property, rdfs:subPropertyOf, author_of, -, "ex.owl"),
  eq(Property2, rdf:typeOf),
  triple(Type, rdfs:subClassOf, manuscript, -, "ex.owl").

```

---

The encoding takes into account the following elements.

- The **return** expression generates an XML document, and therefore the encoding includes the schema rules of the output document. In the previous example, this is the case of rule (1), describing that the *list* label includes elements labeled as *author*.
- The rules (2) and (3) are the main rules of the encoding, in which the elements of the output document are computed by means of the so-called *join* predicate.
- The *join* predicate is the responsible of the encoding of the **for** and **where** constructs. Each **for** associated to an OWL triple is encoded by means of a call to the *triple* predicate with variables.
- Now, the **where** expression is encoded as follows. In the case of binary operators like “=”, “>”, “>=”, etc, they are encoded by means of Prolog predicates, *eq*, *ge*, *geq*, etc. In the case of built-in boolean operators of the kind *rdf:typeOf*, *rdfs:subClassOf*, etc, a call to the *triple* predicate is achieved.
- Finally, we have to incorporate to the *triple* predicate two new arguments in facts and rules. The first new argument is a list of natural numbers identifying the triple. Each element of the **TBox** and the **ABox** is identified by means of [1], [2], etc. The triples inferred from the **TBox** and the **ABox** can be identified by appending the identifiers of the triples used for the inference. Therefore, in general, each triple can be identified as a list of natural numbers (for instance, [1, 4, 5]). Triple identifiers are required for representing the order of the elements of the output XML document, according to our encoding. The second new argument is the name of the document which stores the triple.

Now, the Prolog goal *?-author(Author,Node,Type,Doc)* has the following answers:

---

```
Author=authortype("Abiteboul",[]),Node=[...],Type=2,Doc="result.xml"
Author=authortype("Suciu",[]),Node=[...],Type=2,Doc="result.xml"
Author=authortype("Buneman",[]),Node=[...],Type=2,Doc="result.xml"
Author=authortype("Buneman",[]),Node=[...],Type=2,Doc="result.xml"
Author=authortype("Simeon",[]),Node=[...],Type=2,Doc="result.xml"
```

---

From the schema rule (rule (1)), and these computed answers, we can rebuild the output XML document:

---

```
<list>
<author>Abiteboul</author>
<author>Suciu</author>
<author>Buneman</author>
<author>Buneman</author>
<author>Simeon</author>
</list>
```

---

Finally, we would like to show the query “Retrieve the equivalent properties of the ontology”, in which we can extract from the input ontology some properties and to represent the output of the query as an ontology:

---

```
< owl:Ontology > {
for ($Object1,$Property,$Object2) in owldocument ("ex.owl")
where $Property=owl:equivalentProperty return
<owl:ObjectProperty rdf:about=$Object1/>
```

---

---

```

<owl:equivalentProperty rdf:resource=$Object2 />
</owl:ObjectProperty>
} </ owl:Ontology >

```

---

Now, the goal is ? – *owlObjectProperty*(*OwlObjectProperty*, *Node*, *Type*, *Doc*), and it is encoded as follows:

---

```

owlOntology(owlOntologytype(OwlObjectProp,[]),Nowl,1,Doc):-
    owlObjectProperty(OwlObjectProp,[Nowlp|Nowl],2,Doc).
owlObjectProperty(owlObjectPropertytype(equivalentPropertytype(“”,[rdfresource=Object2]),
    [rdfabout=Object1]),Nowlp,2,“result.xml”):-
    join(Object1,Object2,Nowlp).
join(OwlOntology,[NTriple,[1]]):-
    triple(Object1,Property,Object2,NTriple,“ex.owl”),
    eq(Property,owl:equivalentProperty).

```

---

## 5 Conclusions and Future Work

In this paper we have studied an extension of XQuery for the querying and reasoning with OWL style ontologies. Such extension combines RDF(S)/OWL and XML documents as input/output documents. By means of built-in boolean operators XQuery can be equipped with inference mechanism for OWL properties. We have also studied how to implement/encode such language in Prolog. We have developed an implementation of the XQuery language and a RDF(S) / OWL reasoning which can be downloaded from our Web site: <http://indalog.ual.es/XQuery> and <http://indalog.ual.es/0Prolog>. In order to avoid the looping of the rules (for instance, the rule (**Sub2**) loops in a Prolog interpreter) a bottom-up interpreter in Prolog has been implemented. In addition, the rules of Figure 7 satisfies a nice property. Applying the rules in a bottom-up fashion they compute a finite set of OWL relationships, that is, a finite set of triples. Therefore, the reasoning in the selected fragment of OWL is finite. It does not hold in general in description logic. Therefore, the implemented RDF(S)/OWL reasoner actually can be used for the pre-processing OWL documents in order to be used in the solving of XQuery queries. It improves the efficiency of the proposed language. The implementation of the XQuery is described in a recent paper [2], and the implementation of the RDF(S)/OWL reasoner is described in [1]. As future work, we would like to extend our work with the handling of more complex ontologies in the line of [13, 19, 32]. In addition, we are now developing an implementation of the proposed extension of XQuery, but using the available extension mechanisms of XQuery. We believe that it can be significant for the widespread acceptance of the approach.

## References

1. J. M. Almendros-Jiménez. An RDF Query Language based on Logic Programming. *Electronic Notes in Theoretical Computer Science*, 200(3), 2008.
2. J. M. Almendros-Jiménez. An Encoding of XQuery in Prolog. In *Proceedings of the Sixth International XML Database Symposium XSym’09*, LNCS 5679, pages 145–155. Springer, 2009.
3. J. M. Almendros-Jiménez. A Query Language for OWL based on Logic Programming. In *5th Int’l Workshop on Automated Specification and Verification of Web Systems, WWv’09*, pages 69–84, 2009.

- 
4. J. M. Almendros-Jiménez, A. Becerra-Terón, and F. J. Enciso-Baños. Integrating XQuery and Logic Programming. In *Proceedings of INAP-WLP'07*, pages 117–135, Heidelberg, Germany, 2009. Springer LNAI, 5437.
  5. J. M. Almendros-Jiménez, A. Becerra-Terón, and Francisco J. Enciso-Baños. Querying XML documents in logic programming. *Journal of Theory and Practice of Logic Programming*, 8(3):323–361, 2008.
  6. James Bailey, François Bry, Tim Furche, and Sebastian Schaffert. Web and Semantic Web Query Languages: A Survey. In *Proc. of Reasoning Web, First International Summer School*, pages 35–133, Heidelberg, Germany, 2005. Springer LNCS 3564.
  7. Tim Berners-Lee. N3QL-RDF Data Query Language. Technical report, Online only, 2004.
  8. P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 479–490. ACM New York, NY, USA, 2006.
  9. Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Pathfinder: XQuery - The Relational Way. In *Proc. of the International Conference on Very Large Databases*, pages 1322–1325, New York, USA, 2005. ACM Press.
  10. Alex Borgida. On the relative expressiveness of Description Logics and Predicate Logics. *Artificial Intelligence*, 82(1-2):353–367, 1996.
  11. D. Chamberlin, Denise Draper, Mary Fernández, Michael Kay, Jonathan Robie, Michael Rys, Jerome Simeon, Jim Tivy, and Philip Wadler. *XQuery from the Experts*. Addison Wesley, Boston, USA, 2004.
  12. I. Davis. RDF Template Language 1.0. Technical report, Online only, September 2003.
  13. Jos de Bruijn, Rubén Lara, Axel Polleres, and Dieter Fensel. OWL DL vs. OWL Flight: conceptual modeling and reasoning for the semantic Web. In *WWW '05: Proceedings of the 14th International Conference on World Wide Web*, pages 623–632, New York, NY, USA, 2005. ACM Press.
  14. Cristian Pérez de Laborda and Stefan Conrad. Bringing Relational Data into the Semantic Web using SPARQL and Relational OWL. In *Procs. of ICDEW'06*, page 55, Los Alamitos, CA, USA, 2006. IEEE Computer Society.
  15. Tim Furche, François Bry, and Oliver Bolzer. Marriages of Convenience: Triples and Graphs, RDF and XML in Web Querying. In *Proceedings of Third Workshop on Principles and Practice of Semantic Web Reasoning*, pages 72–84, Heidelberg, Germany, 2005. REWERSE, Springer LNCS 3703.
  16. Benjamin N. Grosz, Ian Horrocks, Raphael Volz, and Stefan Decker. Description Logic Programs: Combining Logic Programs with Description Logic. In *Proc. of the International Conference on World Wide Web*, pages 48–57, USA, 2003. ACM Press.
  17. Volker Haarslev and Ralf Möller. Racer system description. In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, pages 701–706, London, UK, 2001. Springer-Verlag.
  18. Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A Comparison of RDF query languages. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *Proceedings of the Third International Semantic Web Conference*, pages 502–517, Heidelberg, Germany, November 2004. Springer LNCS 3298.
  19. Ullrich Hustadt, Boris Motik, and Ulrike Sattler. Reasoning in Description Logics by a Reduction to Disjunctive Datalog. *J. Autom. Reasoning*, 39(3):351–384, 2007.

- 
20. Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: a declarative query language for RDF. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 592–603, New York, NY, USA, 2002. ACM Press.
  21. H. Katz. XsRQL: an XQuery-style Query Language for RDF. Technical report, Online only, 2004.
  22. Markus Krötzsch, Sebastian Rudolph, and Pascal Hitzler. Description Logic Rules. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI-08)*. IOS Press, 2008.
  23. G. Lukacsy, P. Szeredi, and B. Kadar. Prolog Based Description Logic Reasoning. In *Proceedings of the 24th International Conference on Logic Programming*, pages 455–469. Springer, 2008.
  24. A. Marian and J. Simeon. Projecting XML Documents. In *Proc. of International Conference on Very Large Databases*, pages 213–224, Burlington, USA, 2003. Morgan Kaufmann.
  25. Libby Miller, Andy Seaborne, and Alberto Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 423–435, Heidelberg, Germany, 2002. Springer.
  26. Jonathan Robie, Lars Marius Garshol, Steve Newcomb, Michel Biezunski, Matthew Fuchs, Libby Miller, Dan Brickley, Vassilis Christophides, and Gregorius Karvounarakis. The Syntactic Web: Syntax and Semantics on the Web. *Markup Languages: Theory & Practice*, 4(3):411–440, 2002.
  27. S. Schaffert and F. Bry. A Gentle Introduction to Xcerpt, a Rule-based Query and Transformation Language for XML. In *Proc. of International Workshop on Rule Markup Languages for Business Rules on the Semantic Web*, page 22 pages, Aachen, Germany, 2002. CEUR Workshop Proceedings 60.
  28. Michael Sintek and Stefan Decker. TRIPLE - A Query, Inference, and Transformation Language for the Semantic Web. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 364–378, Heidelberg, Germany, 2002. Springer.
  29. Evren Sirin, Bijan Parsia, Bernardo C. Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53, June 2007.
  30. Adam Souzis. RxPath: a mapping of RDF to the XPath Data Model. In *Extreme Markup Languages*, 2006.
  31. D. Tsarkov and I. Horrocks. FaCT++ Description Logic Reasoner: System Description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, pages 292–297, Heidelberg, Germany, 2006. Springer LNAI 4130.
  32. Raphael Volz. *Web Ontology Reasoning with Logic Databases*. PhD thesis, Universität Fridericiana zu Karlsruhe, 2004.
  33. W3C. RDF Vocabulary Description Language 1.0: RDF Schema. Technical report, www.w3.org, 2004.
  34. W3C. Resource Description Framework (RDF). Technical report, www.w3.org, 2004.
  35. W3C. XML Path Language (XPath) 2.0. Technical report, www.w3.org, 2007.
  36. W3C. XML Query Working Group and XSL Working Group, XQuery 1.0: An XML Query Language. Technical report, www.w3.org, 2007.
  37. W3C. OWL 2 Web Ontology Language. Technical report, www.w3.org, 2008.
  38. Norman Walsh. RDF Twig: Accessing RDF Graphs in XSLT. In *Proceedings of Extreme Markup Languages*, 2003.
-



---

# JSquash: Source Code Analysis of Embedded Database Applications for Determining SQL Statements

Dietmar Seipel<sup>1</sup>, Andreas M. Boehm<sup>1</sup>, and Markus Fröhlich<sup>1</sup>

University of Würzburg, Department of Computer Science  
Am Hubland, D-97074 Würzburg, Germany  
seipel@informatik.uni-wuerzburg.de, markusfroehlich1@gmx.de, ab@andiboehm.de

**Abstract.** In this paper, we analyse Java source code of embedded database applications by means of static code analysis. If the underlying database schema is changed due to refactoring or database tuning, then the SQL statements in the embedding Java program need to be adapted correspondingly. This should be done mostly automatically, since changing software manually is error-prone and time consuming.

For determining the SQL statements that access the database, we can either look at the database logfile, an audit file, or at the Java source code itself. Here, we show how to statically determine even the strings for dynamically built SQL statements directly from the Java source code. We do this without using a debugger or a virtual machine technique; instead, we trace the values of variables that contribute to a query string backwards to predict the values as precisely as possible.

We use PROLOG's declarative features and its backtracking mechanism for code analysis, refactoring, and tuning.

## 1 Introduction

During the software life-cycle, enterprise-class databases undergo a lot of changes in order to keep up with the ever-changing requirements. The growing space requirements and complexity of productive databases make the task of maintaining a good performance of the database query execution more and more complicated. The performance is highly dependent on the database schema design [14], and additionally, a complex database schema is more prone to design errors.

Increasing the performance and the manageability of a database usually requires *analysing* and *restructuring* the database schema and therefore affects the application code indirectly. The application code highly depends on the database schema, because the database queries are embedded in the source code. Usually, they are contained in string variables or, more dynamically, statements are generated step by step by using control structures and string concatenations.

Nearly any type of database schema modification implies the adaption of the queries embedded in the application code. Sometimes, the logic of the construction of the query strings has to be changed, too. Some efforts have been made to prescind the applications code from the data persistence in the database [11]. The employed methods require a configuration that tells the database service the mapping between database tables and

---

application objects. Moreover, the relationships between the tables must also be included in that configuration. Therefore, such approaches are also affected by database changes and need to be adjusted accordingly.

Recently, we have developed a standardized XML representation of SQL statements and database schemas named SQUASHML, that was introduced within the PROLOG based toolset Squash for *refactoring* and *tuning* relational database applications [1]. Squash is the first tool that supports the analysis and the refactoring of database applications with coordinated simultaneous modification of the SQL code and the database schema definition. Squash also detects inconsistencies and common flaws in SQL statements. It can determine an optimized configuration of indexes as well as of tables, and propose modifications of the database schema that result in an efficiency gain. The changes are applied automatically to both the schema and the SQL statements.

In this paper, we develop an extension named JSquash of Squash that analyses a given Java source code and presents all expressions in the code that influence the construction of the embedded SQL statements. Our approach estimates or even determines the values of variables in the source code of the databases application by static code analysis; this is used to predict the embedded database queries that can be generated by the application.

Related research on *source code analysis* has mainly focussed on static analysis providing information about security issues (secure code) [8,3], model extraction [10], code smells [7], obvious errors [15,13], code metrics [6], and flow analysis [5]. Recently published results were mainly about supporting the complex task of understanding the run-time behaviour of legacy and non-legacy software systems [18,6]. An extensive amount of research has been dedicated to system analysis by extracting information generated during the run-time of a software system [6]. These approaches require the code being instrumented prior to execution by various techniques, such as wrapping, logging or extended virtual machines. The output of such an analysis is an execution trace containing information about the run-time behaviour of the system, such as call hierarchies of methods and object creation. Such traces of productive software systems are often very voluminous, and usually the values of interesting variables are missing.

The rest of the paper is organized as follows: Section 2 summarizes the basic concepts of managing source code with PROLOG in the JSquash repository. Section 3 describes some basic methods of static code analysis supported by JSquash, such as the calculation of metrics and the detection of code smells. Section 4 presents the strategies of JSquash used for recursively evaluating the control flow of the source code in order to determine values of variables at run-time for predicting embedded SQL statements. Section 5 shows how JSquash can visualise the creation of embedded SQL statements using HTML 4, CSS, and JavaScript technology, and how Squash can visualise join conditions of complex SELECT statements. Finally, Section 6 summarizes our work.

## 2 Management of Java Source Code with PROLOG

In JSquash, Java code is first parsed and transformed into an XML representation called JAML [9]. Subsequently, a source code *repository* is built from the XML representa-

---

tion, which suitably represents the constructs of the analysed programming language. It is essential to choose an adequate format for the repository, such that the analysis can handle the data efficiently. The analysis is implemented using the declarative logic programming system SWI-PROLOG [4, 19].

## 2.1 Representation of Java Source Code in XML

The XML representation JAML completely resembles the Java source code; it even conserves the layout of the source code. JAML enables standardised access to the source code. Generally, XML is a markup language for representing hierarchically structured data, which is often used for the exchange of data between different applications; as a particular XML language, JAML is a very good candidate for an intermediate representation during the creation of the repository out of the Java source code. It comes with a plugin that can be installed as an online update in the integrated development environment Eclipse, that performs the transformation from Java to XML on-the-fly while programming. For every type of expressions, i.e., variables, statements and control structures, there is a special XML element in JAML, which holds the hierarchical design of the program as well as detailed information about the represented Java component.

According to the W3C XML 1.1 recommendation, the terseness in XML markup is of minimal importance, and JAML increases the volume by a factor of about 50. To meet our requirements, the contained information has to be condensed; e.g., the single indenting white spaces are of no interest.

For example, the Java variable declaration `int a = 5`, that additionally includes an initialisation, is presented in JAML, as shown in the listing below; for clarity, the representation has been strongly simplified. The Java declaration is represented by a `variable-declaration-statement` element in JAML. The included `type` element sets the data type of the variable `a` to integer. The assignment is represented by the `variable-declaration` subelement: the identifier of the variable is given by the `identifier` element, and the literal expression is given by an `expression` element.

```
<variable-declaration-statement>
  <type kind="primitive-type">
    <primitive-type> <int>int</int> </primitive-type>
  </type>
  <whitespace/>
  <variable-declaration-list>
    <variable-declaration>
      <identifier>a</identifier> <whitespace/>
      <initializer>
        <equal>=</equal> <whitespace/>
        <expression>
          <literal-expression type-ref="int">
            <literal>
              <number-literal>5</number-literal>
            </literal>
          </literal-expression>
        </expression>
      </initializer>
    </variable-declaration>
  </variable-declaration-list>
</variable-declaration-statement>
```

---

```

        </literal>
    </literal-expression>
</expression>
</initializer>
</variable-declaration>
</variable-declaration-list>
</variable-declaration-statement>

```

For deriving the JSquash repository, the JAML data are represented in field notation and processed using the XML query, transformation, and update language FNQuery [16, 17]. The query part of the language resembles an extension of the well-known XML query language XQuery [2]; but FNQuery is implemented in and fully interleaved with PROLOG. The usual axes of XPath are provided for selection and modification of XML documents. Moreover, FNQuery embodies transformation features, which go beyond XSLT, and also update features.

## 2.2 The JSquash Repository

The JSquash repository stores the relevant elements of the Java code, which are extracted from the JAML representation, in the form of PROLOG facts. These facts represent information that at least consists of the type and a description of the location within the source code including the locations of the surrounding code blocks. Additionally, necessary parameters can be added, depending on the type.

The construct of a *path* reflects the hierarchy of the code nesting. A path starts with the file number of the source code; the further elements of the path are derived from the position attributes (*pos*) of the JAML representation.

The JSquash repository supports access to variables, objects, classes as well as method calls. E.g., a variable declaration is stored in the repository using a fact

```

jsquash_repository(
    Path:'variable-declaration', Type, Id, P:T).

```

where *Path* is the path of the statement, *Type* is the type and *Id* is the name of the variable, and *P:T* is an optional reference to the in-place initialisation.

For example, the representation of the following fragment of a Java source code, which starts at position 101 in the Java source file, in the JSquash repository will be explained in more detail.

```

1: int a = 5;
2: int b = 10;
3: int c = a + b;
4: int d = c;

```

Due to surrounding blocks and preceding statements, all paths have the common prefix "0, 49, 3, 49, 94", which we abbreviate by "...". Since *a* starts at position 105 and *b* starts at position 109, the declaration `int a = 5` in line 1 is represented by the following two facts:

---

```

jsquash_repository(
  [..., 105]:'variable-declaration', int, a,
  [..., 105, 109]:'literal-expression').
jsquash_repository(
  [..., 105, 109]:'literal-expression', int, 5).

```

The type `variable-declaration` requires two parameters: the type of the variable and its name. In this example, the in-place initialisation with the value 5 is represented in the repository by the path of that expression as a reference, here `[..., 105, 109]`, together with the fact of the referenced expression. Such expressions are represented in JSquash by the type `literal-expression`.

The source code of line 2 is represented by two similar facts, where "5" is replaced by "10", "105" is replaced by "119", and "109" is replaced by "123".

Line 3 is more complex, because it contains the sum of `a` and `b` in the initialisation, a binary expression that comes in the repository as the type `binary-expression`.

```

jsquash_repository(
  [..., 134]:'variable-declaration', int, c,
  [..., 134, 138]:'binary-expression').
jsquash_repository(
  [..., 134, 138]:'binary-expression',
  [..., 134, 138, 138]:'variable-access-expression', +,
  [..., 134, 138, 142]:'variable-access-expression').
jsquash_repository(
  [..., 134, 138, 138]:'variable-access-expression', a).
jsquash_repository(
  [..., 134, 138, 142]:'variable-access-expression', b).

```

The reference of the declaration fact points to the description of the binary expression, which holds two references, one for the left and one for the right expression; in our case, both are accesses to variables.

Line 4 contains an access to a variable instead of a literal expression. This is represented in the repository by the type `variable-access-expression`, which works like `literal-expression`, but it has no qualifier for local variables – as in our example.

```

jsquash_repository(
  [..., 152]:'variable-declaration', int, d,
  [..., 152, 156]:'variable-access-expression').
jsquash_repository(
  [..., 152, 156]:'variable-access-expression', c).

```

The notation described above has been used for convenience to improve the readability of rules referring to the repository. For *efficiency* reasons, we store different facts with the predicate symbols `jsquash_repository_/3,4,5` in the repository; then we can make use of the index on the first argument to search for facts of a given type. The predicates from above are derived using the following simple rules:

---

```

jsquash_repository(P1:T1, Type, Id, P2:T2) :-
    jsquash_repository_(T1, Type, Id, P2:T2, P1).
jsquash_repository(P1:T1, P2:T2, Op, P3:T3) :-
    jsquash_repository_(T1, P2:T2, Op, P3:T3, P1).
jsquash_repository(P:T, Id) :-
    jsquash_repository_(T, Id, P).

```

Using the design of the repository described here, JSquash is able to work efficiently with application code of any complexity.

### 3 Static Code Analysis

Source code analysis comprises the manual, tooled or automated verification of source code regarding errors, coding conventions, programming style, test coverage, etc.

The tool JSquash supports some standard methods of static code analysis, such as the calculation of metrics and the detection of code smells. In the following, we will show some examples.

#### 3.1 Local and Global Variables

The following rule determines the classes and identifiers of all local (if `Type` is given by 'variable-declaration') or global (if `Type` is 'field-declaration') variables within the source code, respectively:

```

variables_in_system(Type, Class, Id) :-
    jsquash_repository([N]:'java-class', _, Class),
    jsquash_repository([N|_]:Type, _, Id, _).

```

Note, that the repository facts for Java classes have a path consisting of only one number. Every variable in a Java class must have this number as the first element of its path.

#### 3.2 Detection of Flaws and Code Smells

JSquash supports the detection of some types of code flaws, i.e., code sections that could be sources of potential errors.

The following rule determines the method signatures of all methods having more than one `return` statement:

```

methods_with_several_returns(
    Class, Method, Signature) :-
    jsquash_repository([N]:'java-class', _, Class),
    jsquash_repository([N|Ns]:'method-declaration',
        Method, Signature, _),
    findall( Path,

```

---

```

    ( jsquash_repository(
        Path:'return-expression', _, _),
      append([N|Ns], _, Path) ),
    Paths ),
    length(Paths, Length),
    Length > 1.

```

The method declarations within a Java class have the position of the class as the first element in their path. Similarly, the return expressions of a method have the path of the method as a prefix of their path.

The following rule detects all `if` conditions that always evaluate to `false`. It determines the sections of all such `if` statements within the analysed code. If all possible evaluations of an `if` condition are `false`, then the `if` statement is unnecessary.

```

unnecessary_if_statements(If_Path) :-
    jsquash_repository(
        If_Path:'if-statement', P:T, _, _),
    init_run_time(R),
    set_run_time(R, R2, [
        searched_expression@path=P,
        searched_expression@type=T]),
    forall( eval(R2:P:T, Value),
        Value = [boolean, false|_] ).

```

After initialising the run-time, which will be explained in the following section, with `init_run_time/1`, the path and the type of the searched expression are stored in the run-time using `set_run_time/3`.

All of these features make use of the basic code analysis that is supported and integrated in JSquash. This comprises the detection of the following facts: which code sections influence the values of variables, which methods are calling other methods (call dependencies), and which objects are created at which time by which other objects. The predicate `eval/2` for backward tracing the flow of control leading to the values of variables will be explained in the following section.

## 4 Backward Tracing of the Control Flow

While looking for actual values of variables, JSquash recursively calls the predicate `eval/2` until the evaluation reaches assignments having concrete literal expressions on their right hand side. As literal expressions need not to be evaluated, their values can immediately be used as arguments in complex expressions. After the recursion is rolled out completely, the calculated values of the partial expressions are returned until the value of the examined expression is determined.

For example, during the search for the current value of `d` in line 4 of the code fragment from Section 2.2, first the variable `c` in the right hand side of the assignment has to be evaluated. Therefore, JSquash detects the assignment in line 3, where `c` can be

---

```

public class Sender {

    public void sendMessage() {

        String tCondColumns = { "a", "b", "c" };
        String tCondValues = { "1", "2", "3" };
        String tOrderByColumns = { "c", "d" };

        String tSQL = "SELECT * FROM training";
        tSQL += " WHERE ";
        tSQL += tCondColumns[0] + " = " + tCondValues[0] + " AND ";
        tSQL += tCondColumns[1] + " = " + tCondValues[1] + " AND ";
        tSQL += tCondColumns[2] + " = " + tCondValues[2];

        tSQL += " ORDER BY ";
        tSQL += tOrderByColumns[0] + ", ";
        tSQL += tOrderByColumns[1];
        tSQL += " ASCENDING";

        Connection con = DBTools.getConnection();

        try {
            PreparedStatement st = con.prepareStatement(tSQL);
            st.execute();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

**Fig. 1.** A code example that uses arrays of strings to construct an SQL statement.

evaluated by determining the current values of **a** in line 1 and **b** in line 2 and by returning their sum. Another example using arrays of strings to construct an SQL statement is shown in Figure 1.

The current state of the analysed program plays an important role for `eval/2`, because it determines the control flow of the applications's calculations. The program state is the set of all variables and their current values that are needed for evaluation of the control structures involved in the calculation of the value of the examined variable.

#### 4.1 Overview of the Evaluation Strategy

Evaluating a specific variable means accessing it at a specific point of run-time. As variables always receive their values strictly via assignment expressions, the analysis component has to find the last assignment before the examined access to the variable.

Therefore, the possible assignments have to be examined. To do so, a simulation of the control flow has been implemented in JSquash that considers control flow statements such as loops and `if-then-else` constructs. This machine recognizes, if a program block directly or indirectly influences the value of the examined variable.

The search for the current value of a variable has been implemented in the predicate `eval/2`. Given the reference to a variable, it returns the corresponding value, depending on the current state. On backtracking, it determines all further value assignments for the given variable. E.g., `if` statements may lead to alternative value assignments,



---

if the condition cannot be evaluated and the `then` or `else` part assigns a value to the examined variable. But, often there is only one possible value assignment.

`eval/2` is always called with the first argument of the form  $R:P:T$ . The *run-time* object  $R$  holds fundamental information for determining the values of variables. It is represented as an XML object in field notation, which holds the following information:

- the current position of the analysis in the source code,
- the analysed control structure,
- the state of the variables defining the current state and control flow,
- the currently examined variable, and
- the currently examined assignment.

The run-time is needed as a supporting data structure for analysing the lifetime behavior of loops and other constructs that have nested blocks, including recursive method calls. In these cases, it is used for holding the distinct states of the blocks passed during the analysis. The value history of the run-time is used extensively to reflect the construction and dependencies of each single value of a variable.

$P:T$  *references* the currently searched expression. Based on the type of the referenced expressions, JSquash can decide which rule of `eval/2` has to be used for determining the actual value of the given expression. In the case of variable accesses, the `eval/2` rule – which is shown in Section 4.2 – determines the current value of the referenced variable at the time of the access.

For each type  $T$  of *control structure*, a rule has been developed that simulates its behaviour and functionality. These rules implement evaluation strategies that yield the current value for all type primitives. The handling of local and global variables (class fields) is implemented separately, since the evaluation strategies differ. In the following, we will show some examples; more complicated cases, such as the handling of Java loops, cannot be shown due to the inherent complexity of their evaluation.

## 4.2 Variable Access Expressions

While processing the Java code fragment of Section 2.2, JSquash has at first to resolve the identifier `d`. The JSquash repository fact

```
jsquash_repository(  
    [..., 152, 156]:'variable-access-expression', c).
```

shows that the variable access expression at `[..., 152, 156]` refers to the variable `c`. Based on the repository fact

```
jsquash_repository(  
    [..., 134]:'variable-declaration', int, c,  
    [..., 134, 138]:'binary-expression').
```

the predicate `next/2` finds out that the most recent assignment defining `c` was the binary expression at `[..., 134, 138]`, cf. line 3 of the Java code fragment:

---

```

eval(R:P:T, Value) :-
    T = 'variable-access-expression',
    jsquash_repository(P:T, Id),
    set_run_time(R, R2, [
        @searched_id=Id,
        @search_mode=variable,
        @scope=T ]),
    next(R2:P:T, R3:P3:T3),
    eval(R3:P3:T3, V),
    handle_postfix(P3:T3, V, Value).

```

Similarly, the repository fact

```

jsquash_repository(
    [..., 105]:'variable-declaration', int, a,
    [..., 105, 109]:'literal-expression').

```

is used later during the computation to find out that the variable `a` is declared using a literal expression.

For finding the most recent assignment to the examined variables, the predicate `next/2` has to traverse the JSquash repository facts in inverse code order. This can be supported by further PROLOG facts in the repository, which link a statement to its preceding statement in the code order.

### 4.3 Binary Expressions

The repository fact

```

jsquash_repository([..., 134, 138]:'binary-expression',
    [..., 134, 138, 138]:'variable-access-expression', +,
    [..., 134, 138, 142]:'variable-access-expression').

```

shows that the binary expression for `c` refers to two variable access expressions. After evaluating them, the resulting values are combined using the binary operator (in our case `+`) indicated by the fact from the repository:

```

eval(R:P:T, Value) :-
    T = 'binary-expression',
    jsquash_repository(P:T, P1:T1, Op, P2:T2),
    eval(R:P1:T1, V1),
    eval(R:P2:T2, V2),
    apply(Op, [V1, V2, Value]).

```

If we evaluate the following code fragment, then both expressions within the binary expression in line 2 are evaluated w.r.t. the same runtime `R`, but with different references `P1:T1` and `P2:T2`, respectively:

```

1: int a = 5;
2: int c = a++ + a;

```

---

The call `eval(R:P1:T1, V1)` evaluates the left expression `a++` to 5; only afterwards, the value of `a` is incremented to the new value 6. The call `eval(R:P2:T2, V2)` for the right expression `a` re-evaluates the left expression, since the new value of `a` is relevant. Thus, the right expression `a` correctly evaluates to 6, and finally, `c` evaluates to 11, i.e., `5 + 6`.

#### 4.4 Literal Expressions

The repository fact

```
jsquash_repository(  
    [..., 105, 109]:'literal-expression', int, 5).
```

shows that the value of the literal expression at `[..., 105, 109]` is 5. This type of expression is evaluated using the following rule:

```
eval(R:P:T, Value) :-  
    T = 'literal-expression',  
    jsquash_repository(P:T, _, Value).
```

### 5 Visualisation of Embedded SQL Statements

The tool JSquash can detect and analyse SQL statements embedded in the Java source code of database applications.

#### 5.1 SQL Statements Embedded in the Source Code

For presenting the results of the analysis to the user, JSquash includes a generator component, that produces an HTML document containing the complete information about the detected SQL statements, including the full SQL code and the source code that contributes to each SQL statement. This HTML document comprises a fully functional *graphical user interface* (GUI) that can be opened and used with any Web browser. The GUI is implemented in HTML 4 using cascading style sheets (CSS) and JavaScript; the JavaScript helps building a dynamic GUI.

The expressions that contribute to the following generated SQL statement have been detected by JSquash and are automatically highlighted, see Figure 1. JSquash was also able to build the full SQL statement by only analyzing the source code:

```
SELECT * FROM training  
WHERE a = 1 AND b = 2 AND c = 3  
ORDER BY c, d ASCENDING
```

This statement – which is the second SQL statement in the GUI of Figure 2 – is visualised in Figure 3. The left side in of the GUI shown in Figure 3 displays all the class files of the source code that contribute to the detected SQL statements. Clicking on the

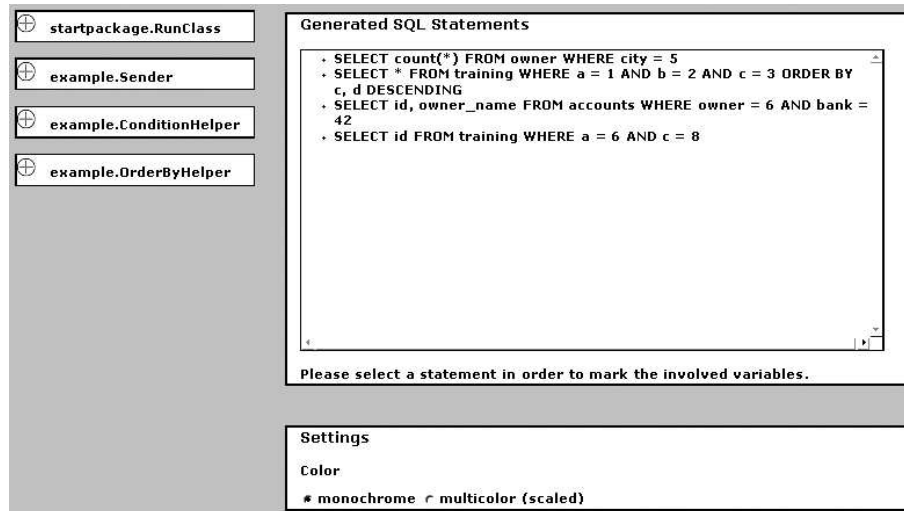


Fig. 2. The GUI of JSquash. No SQL statement is selected.

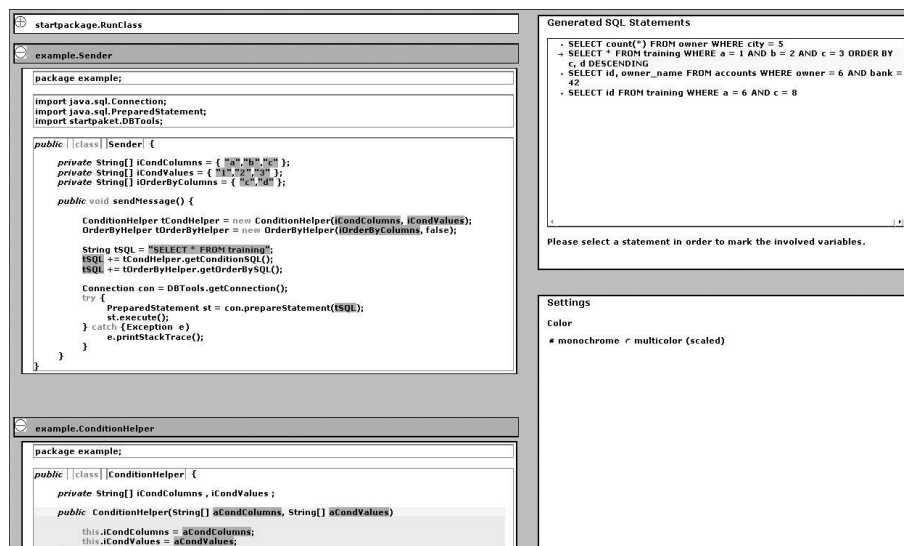


Fig. 3. All contributing values and variables of the selected second SQL statement are marked.

buttons at the left side of the class name opens (+) or closes (–) the source code, respectively. At the upper right side, all detected SQL statements are shown. Below is the block of settings, that allow for changing the highlighting colors.

If an SQL statement of the list is selected, then the corresponding code sections and expressions are automatically highlighted in the listings at the left side, cf. Figure 3. Thus, the user can easily analyse all code sections that contribute to the selected SQL

---

statement. This feature is implemented in JavaScript and CSS, making extensive use of the path information from the repository.

## 5.2 Representation and Visualisation of SQL Statements

Recently, we have developed the tool Squash for analysing, tuning and refactoring relational database applications [1]. It uses an extensible and flexible XML representation for SQL database schema definitions and queries called SQUASHML, that is designed for representing schema objects, as well as database queries and data modification statements.

The core of SQUASHML follows the SQL standard, but it also allows for system-specific constructs of different SQL dialects; for example, some definitions and storage parameters from the Oracle database management system have been integrated as optional XML elements.

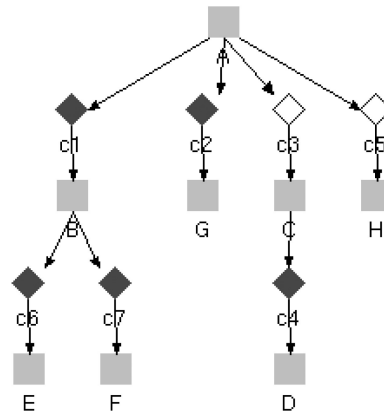
The SQUASHML format allows for easily processing and evaluating the database schema information. Currently, supported schema objects include table and index definitions. Other existing XML representations of databases, such as SQL/XML, usually focus on representing the database contents, i.e., the table contents, and not the schema definition itself [12]. SQUASHML was developed specifically to map the database schema and queries, without the current contents of the database.

The SQL statements detected by JSquash are transformed to SQUASHML, and then the tool Squash can be used for the visualisation of the single statements. E.g., the following SELECT statement from a biological application joins 9 tables; the table names have been replaced by aliases A, ..., I:

```
SELECT * FROM A, B, C, D, E, F, G, H
WHERE A.ID_DICT_PEPTIDE IN (
  SELECT ID_PEPTIDE FROM I
  WHERE I.ID_SEARCH = 2025
  GROUP BY ID_PEPTIDE
  HAVING COUNT(ID_SEARCH_PEPTIDE) >=1 )
AND A.ID_SEARCH = 2025
AND c1 AND c2 AND A.FLAG_DELETED = 0
AND c3 AND c6 (+) AND c7 (+) AND c4 AND c5
AND E.LEN >= 6 AND A.NORMALIZED_SCORE >= 1.5
ORDER BY ...
```

The following 7 join conditions are used:

```
c1: A.ID_SEARCH_PEPTIDE = B.ID_SEARCH_PEPTIDE
c2: A.ID_SPECTRUM = G.ID_SPECTRUM
c3: A.ID_SEARCH_PEPTIDE = C.ID_PEPTIDE
c4: C.ID_SEQUENCE = D.ID_SEQUENCE
c5: A.ID_SEARCH = H.ID_SEARCH
c6: B.ID_PEPTIDE = E.ID_DICT_PEPTIDE
c7: B.ID_PEPTIDE_MOD = F.ID_DICT_PEPTIDE
```



**Fig. 4.** Join Conditions in a Query

This query is parsed into the following SQUASHML element; we leave out some opening and closing tags, respectively:

```
<select>
  <subquery id="subquery_1">
    <select_list> <expr> <simple_expr>
      <object table_view="A" column="ID_SEARCH"/> ...
    <from> <table_reference> ...
      <simple_query_table_expression
        object="A" schema="USER"/> ...
    <where> ...
    <order_by> ...
  </subquery>
</select>
```

The conditions in the WHERE part (e.g., the join condition c1) look like follows:

```
<condition>
  <simple_comparison_condition operator="=">
    <left_expr> <expr> <simple_expr>
      <object table_view="A" column="ID_SEARCH_PEPTIDE"/> ...
    <right_expr>...
      <object table_view="B" column="ID_SEARCH_PEPTIDE"/> ...
  </simple_comparison_condition>
</condition>
```

Squash provides a number of different visualization methods for the database schema and the queries. Complex select statements tend to include many tables in join operations. Therefore, Squash uses a graph representation for query visualization, cf. Figure 4. If a SELECT statement contains nested subqueries (like the statement shown above), then these queries can be included in the graphical output if desired.

---

## 6 Conclusions

We have shown, how to derive the content of application variables in Java programs using means of static code analysis. Our tool JSquash, which is implemented in PROLOG, predicts the values of variables as precisely as possible; obviously, some values cannot be discovered at compile-time, e.g., if a value was obtained through I/O operations.

Now, we are able to analyse *embedded SQL statements* of a given database application, either by analysing audit files of the database connection using the basic tool Squash [1], or by static source code analysis with the extended tool JSquash. The statements derived from the source code of the database application can be imported into Squash, which can then generate database modifications for improving the performance of the application.

Future work will be on developing methods that preserve the linkage between the detected single SQL statement fragments and their positions as well as each of their effects in the completed statement. This extension to SQUASHML will then allow for injecting the changes proposed by Squash into the original source code of the application, and it will help conducting the appropriate changes there.

Moreover, we will try to apply similar techniques of static code analysis to PROLOG programs with embedded SQL statements as well.

## References

1. BOEHM, A. M., SEIPEL, D., SICKMANN, A., WETZKA, M.: *Squash: A Tool for Analyzing, Tuning and Refactoring Relational Database Applications*. Proc. 17th International Conference on Declarative Programming and Knowledge Management, INAP 2007, pp. 113–124
2. CHAMBERLIN, D.: *XQuery: a Query Language for XML*. Proc. ACM International Conference on Management of Data, SIGMOD 2003. ACM Press, 2003, pp. 682–682
3. CHESS, B., MCGRAW, G.: *Static Analysis for Security*. IEEE Security & Privacy 2(6). 2004, pp. 76–79
4. CLOCKSIN, W. F.; MELLISH, C. S.: *Programming in PROLOG*. 5th Edition, Springer, 2003
5. CORBETT, J. C.; DWYER, M. B.; HATCLIFF, J.; LAUBACH, S.; PASAREANU, C. S.; ZHENG, R. H.: *Bandera: Extracting Finite State Models From Java Source Code*. Proc. International Conference on Software Engineering, ICSE 2000, pp. 439–448
6. DUCASSE, S., LANZA, M., BERTULI, R.: *High-Level Polymetric Views of Condensed Run-Time Information*. Proc. 8th European Conference on Software Maintenance and Reengineering, CSMR 2004, pp. 309–318
7. VAN EMDEN, E.; MOONEN, L.: *Java Quality Assurance by Detecting Code Smells*. Proc. 9th Working Conference on Reverse Engineering, WCRE 2002. IEEE Computer Society, pp. 97–108
8. EVANS, D., LAROCHELLE, D.: *Improving Security Using Extensible Lightweight Static Analysis*. IEEE Software 19(1). 2002, pp. 42–51
9. FISCHER, D.; LUSIARDI, J.: *JAML: XML Representation of Java Source Code*. Technical Report, University of Würzburg, Department of Computer Science. 2008
10. HOLZMANN, G. J.; SMITH, M. H.: *Extracting Verification Models by Extracting Verification Models*. Proc. Joint International Conference on Formal Description Techniques, FORTE 1999, and Protocol Specification, Testing, and Verification, PSTV 1999, Kluwer, pp. 481–497

- 
11. JBOSS; RED HAT: *Hibernate*. <https://www.hibernate.org/>
  12. INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: *ISO/IEC 9075-14:2003 Information Technology – Database Languages – SQL – Part 14: XML Related Specifications (SQL/XML)*. 2003
  13. MARINESCU, R.: *Detection Strategies: Metrics-Based Rules for Detecting Design Flaws*. Proc. 20th IEEE International Conference on Software Maintenance, ICSM 2004, pp. 350–359
  14. RAMAKRISHNAN, R.; GEHRKE, J.: *Database Management Systems*. 3rd Edition, McGraw–Hill, 2003
  15. REN, X.; SHAH, F.; TIP, F.; RYDER, B. G.; CHESLEY, O.: *Chianti: A Tool for Change Impact Analysis of Java Programs*. ACM SIGPLAN Notices 39(10). 2004, pp. 432–448
  16. SEIPEL, D.: *Processing XML Documents in PROLOG*. Proc. 17th Workshop on Logic Programmierung, WLP 2002
  17. SEIPEL, D.; BAUMEISTER, J.; HOPFNER, M.: *Declarative Querying and Visualizing Knowledge Bases in XML*. Proc. 15th International Conference on Declarative Programming and Knowledge Management, INAP 2004, pp. 140–151
  18. SYSTÄ, T.; YU, P.; MÜLLER, H.: *Analyzing Java Software by Combining Metrics and Program Visualization*. Proc. 4th European Conference on Software Maintenance and Reengineering, CSMR 2000, IEEE Computer Society, pp. 199–208
  19. WIELEMAKER, J.: *An Overview of the SWI-PROLOG Programming Environment*. Proc. 13th International Workshop on Logic Programming Environments, WLPE 2003, pp. 1–16
  20. WIELEMAKER, J.: SWI-PROLOG. Version: 2007. <http://www.swi-prolog.org/>



---

# Reference Model and Perspective Schemata Inference for Enterprise Data Integration

Valéria Magalhães Pequeno and João Carlos Moura Pires

CENTRIA, Departamento de Informática,  
Faculdade de Ciências e Tecnologia, FCT, Universidade Nova de Lisboa  
2829-516, Caparica, Portugal  
`vmp@fct.unl.pt`, `jmp@di.fct.unl.pt`

**Abstract.** We propose a declarative approach based on the creation of a reference model and perspective schemata to deal with the problem of integrating data from multiple, possibly heterogeneous and distributed, databases. The former provides a common semantic, while the latter connects schemata. This paper focuses on deduction of new perspective schemata using a proposed inference mechanism. A proof-of-concept prototype, based on Logic Programming, is slightly presented.

## 1 Introduction

One of the leading issues in database research is to develop flexible mechanisms for providing integrated access to multiple, distributed, heterogeneous databases and other information sources. A wide range of techniques has been developed to address this problem, including approaches based on creation of Data Warehouses (DWs), and Federated Database Systems (FDBSs). DWs are highly specialised database systems which contain the unified history of an enterprise at a suitable level of detail for decision support. All data is integrated into, normally, a single repository, with a generalised and global schema. A FDBS enables a unified virtual view of one or more autonomous sources of information to hide data heterogeneity from applications and users. Closely coupled FDBSs, those that occur in DWs, provides a global schema expressed in a common, “canonical” data model. Unlike a DW, a FDBS leaves data at the source.

One of the main drawbacks of these approaches is the difficulty in developing a single (global or common) database schema that captures all the nuances of diverse data types, and expresses a unified view of the enterprise. The designer usually deal with incompatible data models, characterised by subtle differences in structure and semantic. He/she should define mappings between the global and source information schemata. These problems are hardest to deal with because of the rapid growth of the data volume and the data model complexity (both in sources and in global schema), which implies the rise of the difficulty of managing and understanding these models [1].

In order to deal with these problems, it is proposed to take a declarative approach, based on the creation of a reference model and, perspective schemata.

---

A Reference Model (conceptual model, business data model, or enterprise data model) is an abstract framework that represents the information used in an enterprise from a business viewpoint. It provides a common semantic that can be used to guide the development of other models and help with data consistency [1]. Its benefits include: a) reduction of the development risk by ensuring that all implemented systems correctly reflect the business environment [1]; b) helping with the project scope definition once the designer can use the reference model to identify the information that will be addressed by the systems; c) serving as basis for multiple products such as application systems, DWs, and FDBSs [2, 1], being a more stable basis for identifying information requirements to the DW systems than user query requirements, which are unpredictable and subject to frequent change [3].

A perspective schema describes a data model, part or whole (*the target*), in terms of other data models (*the base*). It represents the mapping between the base schemata (e.g., the information sources) and the target schema (e.g., the reference model). In the proposed approach, the relationship between the base and the target schemata is made explicitly and declaratively through correspondence assertions. By using the perspective schemata the designer has an explicit and formal representation, with well defined semantic, which allows to: evince diverse points of views of the (same or different) source information; b) deal with semantic heterogeneity in a declarative way; and c) reuse (a same perspective schema can be used simultaneously in several (application, DW, FDBS) systems).

An advantage of the proposed approach is that by using the reference model the designer does not need to map each models. This effort is theoretically reduced since schemata (source or global) must only align with the reference model, rather than with each participating schema. Thus, the designer of the DW system (or FDBS) only needs to describe the mapping between the DW and the reference model, and he/she cannot be not involved with the mapping between the different sources. The designer can describe the global system without concerns about where the sources are or how they are stored. Furthermore, the mapping between the global schema and its sources are automatically generated by an inference mechanism. This paper focuses on the deduction of new perspective schemata using a proposed inference mechanism.

The remainder of this paper is laid out as follows. Section 2 concisely mentions representative works in the data integration area. Section 3 presents an overview of the reference model-based framework proposed in [4]. Section 4 briefly describes the language to define the perspective schemata. Section 5 details the process to infer new perspective schemata. The paper ends with Section 6, which points out new features of the approach presented here and for ongoing or planned future work on this topic.

## 2 Related Work

The database community has been engaged for many years with the problem of data integration. Research in this area has developed in several important

---

directions: schema matching and data quality, to cite a couple (see [5] for a survey), which covers different architectures (e.g., FDBSs and DWs), representation of data and involved data models (e.g., relational and non-structured). Recent research in FDBSs has included: behaviour integration [6], integration of non-traditional data (e.g., biomedical [7,8], intelligence data [9], and web source [10]), interactive integration of data [11,12], and federated data warehouse systems [13]. All these approaches use a global schema, but do not deal with a reference model. Similar to the research of the current paper, [10] uses correspondence assertions (in this case, for specifying the semantics of XML-based mediators). However, their CAs only deal with part of the semantic correspondence managed here. Furthermore, they assume that there is a universal key to determine when two distinct objects are the same entity in the real-world, which is often an unreal supposition.

Researches in DWs have focused on technical aspects such as multidimensional data models (e.g., [14–19]) as well as the materialised view definition and maintenance (e.g., [20]). In particular, the most conceptual multidimensional models are extensions to the Entity-Relationship model (e.g., [21–24]) or extensions to UML (e.g., [25–27]).

The work in [28] focuses on an ontology-based approach to determine the mapping between attributes from the source and the DW schemata, as well as to identify the transformations required for correctly moving data from source information to the DW. Their ontology, based on a common vocabulary as well as a set of data annotations (both provided by the designer), allows formal and explicit description of the semantic of the sources and the DW schemata. However, their strategy requires a deep knowledge of all schemata involved in the DW system, which is usually not the usual case. In our research, it is dispensable, since each schema (source or DW) needs to be related only to the reference model. Additionally, we deal with the matching of instances (i.e., the problem to identify the same instances of the real-world that are differently represented in the diverse schemata), and the work in [28] does not.

The closest approach to our research is described in [29]. Similar to our study, their proposal included a reference model (cited as “enterprise model”) designed using an Enriched Entity-Relationship (EER) model. However, unlike our research, all their schemata, including the DW, are formed by relational structures, which are defined as views over the reference model. Their proposal provides the user with various levels of abstraction: conceptual, logical, and physical. In their conceptual level, they introduce the notion of intermodel assertions that precisely capture the structure of an EER schema or allow for the specifying of the relationship between diverse schemata. However, any transformation (e.g., restructuring of schema and values) or mapping of instances is deferred for the logical level, unlike the current work. In addition, they did not deal with complex data, integrity constraints, and path expressions, as our research does.

### 3 The Framework

The proposal presented in [4] offers a way to express the existing data models (source, reference model, and global/integrated schema) and the relationship between them. The approach is based on *Schema language* ( $L_S$ ) and *Perspective schema language* ( $L_{PS}$ ).

The language  $L_S$  is used to describe the actual data models (source, reference model, and global/integrated schema). The formal framework focuses on an object-relational paradigm, which includes definitions adopted by the main concepts of object and relational models as they are widely accepted ([30, 31]).

The language  $L_{PS}$  is used to describe *perspective schemata*. A perspective schema is a special kind of model that describes a data model (part or whole) (*the target*) in terms of other data models (*the base*). In Fig. 1, for instance,  $P_{s'1|RM}$ ,  $P_{s'2|RM}$ ,  $P_{s4|RM}$ , ...,  $P_{sn|RM}$  are perspective schemata that map the reference model (**RM**) in terms of the sources ( $S'_1$ ,  $S'_2$ ,  $S_4$ , ...,  $S_n$ ).  $L_{PS}$  mainly extends  $L_S$  with two components: Correspondence Assertions (CAs) and Matching Functions (MFs). CAs formally specify the relationship between schema components. MFs indicate when two data entities represent the same instance of the real world.  $L_{PS}$  includes data transformations, such as names conversion and data types conversion.

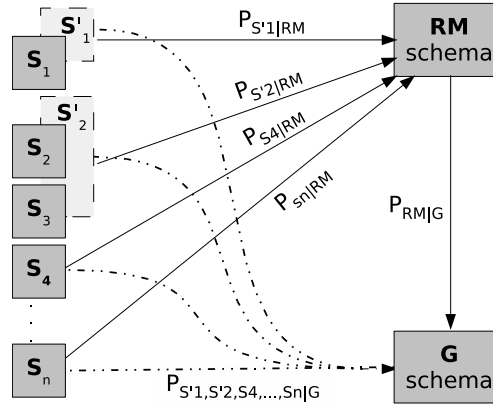


Fig. 1. Proposed architecture

Figure 1 illustrates the basic components of the proposed architecture and their relationships. The schemata **RM**,  $S_1, \dots, S_n$  and **G** are defined using the language  $L_S$  and represent, respectively, the reference model, the sources  $S_1, \dots, S_n$ , and a global schema. The schemata  $S'_1$  and  $S'_2$  are defined using the language  $L_{PS}$ , and represent, respectively, a viewpoint of  $S_1$  and an integrated viewpoint of  $S_2$  and  $S_3$ .  $S'_1$  and  $S'_2$  are special kinds of perspective schemata (called *view schema*), since the target schema is described in the scope of a perspective schema, instead of just referring to an existing model. The relationships

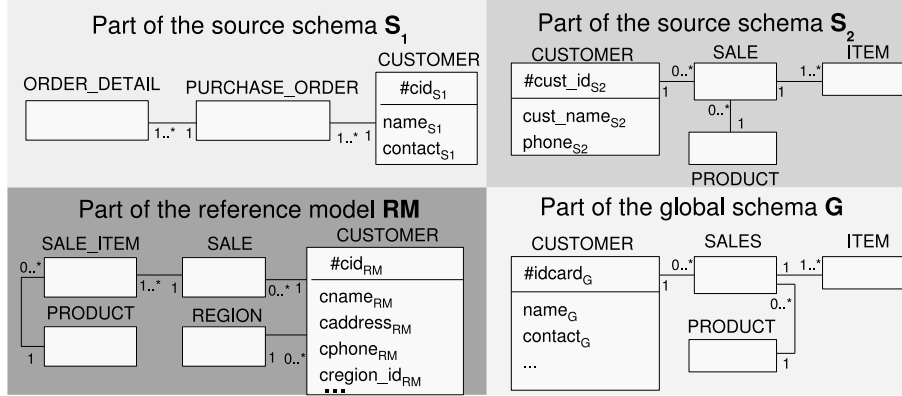


Fig. 2. Motivating example

between the target and the base schemata are shown through the perspective schemata  $P_{s'1|RM}$ ,  $P_{s'2|RM}$ ,  $P_{s4|RM}$ , ...,  $P_{sn|RM}$ ,  $P_{RM|G}$ , and  $P_{s'1,s'2,s4,...,sn|G}$  (denoted by arrows). In the current research,  $P_{s'1,s'2,s4,...,sn|G}$  can be automatically deduced by the proposed inference mechanism. The next Sect. illustrates, through the examples, the language  $L_{PS}$ , and the Sect. 5 presents the proposed inference mechanism. For a more detailed and formal description of  $L_S$  and  $L_{PS}$  languages, the reader is referred to [32, 33, 4].

## 4 Perspective Schema Language

The remainder of the paper, considers a simple sales scenario comprising two data sources  $S_1$  and  $S_2$ , a reference model  $RM$ , and a global schema  $G$ . The schemata are shown in Fig. 2. All properties that are key to a relation (or class) are shown in Fig. 2 using “#” before their names.

The language  $L_{PS}$ , as we mentioned before, is used to define perspective schemata. Usually, a perspective schema is formed by the following components:

1. *Name* is a schema name with the notation:  $P_{S|T}$ , being  $S$  the name of one or more base schemata and  $T$  the name of the target schema. In Fig. 1, for instance,  $P_{s'1|RM}$  is a name of a perspective schema whose base is  $S'_1$  and the target is  $RM$ ;
2. *Require declarations* express the subset of the components of the target schema (classes, relations, keys, and foreign keys) that will be necessary in the perspective schema;
3. *Matching Function signatures* indicate which matching functions must be implemented to determine when two objects/tuples are distinct representations of the same object in the real-world;
4. *Correspondence Assertions* establish the semantic correspondence between schemata's components.

---

The target schema may have much more information than is required to represent in a perspective schema, namely when the target is the Reference Model. Hence, it is necessary to clearly indicate which elements of the target schema are in the scope of the perspective schema. This is done in  $L_{PS}$  using ‘require’ declarations. For instance, consider the perspective schema  $\mathbf{P}_{S_2|RM}$  between the schemata  $\mathbf{RM}$  and  $\mathbf{S}_2$ , both as presented in Fig. 2. For this perspective schema, four relations from  $\mathbf{RM}$  are needed (PRODUCT, CUSTOMER, SALE, and SALE\_ITEM). The ‘require’ declaration to relation CUSTOMER, for example, would be as follows:

$$\text{require}(\text{CUSTOMER}, \{\mathbf{cid}_{RM}, \mathbf{cname}_{RM}, \mathbf{cphone}_{RM}\})$$

Note that, for instance, the properties  $\mathbf{cregion\_id}_{RM}$  and  $\mathbf{caddress}_{RM}$  from  $\mathbf{RM.CUSTOMER}$  are not declared as being required.

#### 4.1 Matching Functions

From a conceptual viewpoint, it is essential to provide a way to identify instances of different models that represent the same entity in the real-world in order to combine them appropriately. This identification (we call it *the instance matching problem*) usually is expensive to compute, due to the complex structure and the character of the data. It is not part of the current research deal with the full instance matching problem. We assume, usually like the DW designers do, that data quality tools were used and that, for instance, duplicates were removed. Even then, in a data integration context, the instance matching problem still persists. The proposal presented in [4] is using MF signatures, which points to situations that should be considered in a data integration context. These signatures define a 1:1 correspondence between the objects/tuples in families of corresponding classes/relations. In particular, the work shown in [4] is based on the following matching function signature:

$$\mathbf{match} : ((\mathbf{S}_1[R_1], \tau_1) \times (\mathbf{S}_2[R_2], \{\tau_2\})) \rightarrow \text{Boolean} , \quad (1)$$

being  $\mathbf{S}_i$  schema names,  $R_i$  class/relation names, and  $\tau_i$  the data type of the instances of  $R_i$ , for  $i \in \{1,2\}$ . When both arguments are instanced, **match** verifies whether two instances are semantically equivalent or not. If only the first argument is instanced (i.e.,  $\mathbf{S}_1.R_1$ ) then it obtains the semantically equivalent  $\mathbf{S}_2.R_2$  instance of the given  $\mathbf{S}_1.R_1$  instance, returning true when it is possible, and false when nothing is found or when there is more than one instance to match.

In some scenarios one-to-many correspondence between instances are common (e.g., when historical data is stored in the DW). In this case, a variant of **match** should be used, which has the following form:

$$\mathbf{match} : ((\mathbf{S}_1[R_1], \tau_1) \times (\mathbf{S}_2[R_2(\mathbf{predicate})], \{\tau_2\})) \rightarrow \text{Boolean} . \quad (2)$$

In (2), **predicate** is a boolean condition that determines the context in which the instance matching must be applied in  $\mathbf{S}_2.R_2$ . An example of a matching function signature involving schemata of Fig. 2 is as follows:

---


$$\mathbf{match} : ((\mathbf{RM}[\mathbf{CUSTOMER}], \tau_1) \times (\mathbf{G}[\mathbf{CUSTOMER}], \{\tau_2\})) \rightarrow \text{Boolean} \quad (3)$$

The implementation of the matching functions shall be externally provided, since their implementation is very close to the application domain and to the application itself.

## 4.2 Correspondence Assertions

The semantic correspondence between schemata's components is declared in the proposal presented in [4] through the CAs, which are used to formally assert the correspondence between schema components in a declarative way. CAs are classified in four groups: Property Correspondence Assertion (PCA), Extension Correspondence Assertion (ECA), Summation Correspondence Assertion (SCA), and Aggregation Correspondence Assertion (ACA). Examples of CAs are shown in Table 1 and explained in this Sect..

**Table 1.** Examples of correspondence assertions

<b>Property Correspondence Assertions (PCAs)</b>
$\psi_1: \mathbf{P}_{\mathbf{RM} \mathbf{G}}[\mathbf{CUSTOMER}] \bullet \mathbf{idcard}_G \rightarrow \text{numberTOtext}(\mathbf{RM}[\mathbf{CUSTOMER}] \bullet \mathbf{cid}_{\mathbf{RM}})$
$\psi_2: \mathbf{P}_{\mathbf{RM} \mathbf{G}}[\mathbf{CUSTOMER}] \bullet \mathbf{contact}_G \rightarrow \mathbf{RM}[\mathbf{CUSTOMER}] \bullet \mathbf{cphone}_{\mathbf{RM}}$
<b>Extension Correspondence Assertions (ECAs)</b>
$\psi_3: \mathbf{P}_{\mathbf{RM} \mathbf{G}}[\mathbf{CUSTOMER}] \rightarrow \mathbf{RM}[\mathbf{CUSTOMER}]$
$\psi_4: \mathbf{S}_v[\mathbf{CUSTOMER}] \rightarrow \mathbf{S}_1[\mathbf{CUSTOMER}] \sqsupset \mathbf{S}_2[\mathbf{CUSTOMER}]$
<b>Summation Correspondence Assertion (SCA)</b>
$\psi_5: \mathbf{P}_{\mathbf{S}_3 \mathbf{RM}}[\mathbf{PRODUCT}] (\mathbf{pid}_{\mathbf{RM}}) \rightarrow \text{normalise}(\mathbf{S}_3[\mathbf{PRODUCT\_SALES}] (\mathbf{product\_numbers}_{\mathbf{S}_3}))$

PCAs relate properties of a target schema to the properties of base schemata. They allow dealing with several kinds of semantic heterogeneity such as: *naming conflict* (for instance synonyms and homonyms properties), *data representation conflict* (that occur when similar contents are represented by different data types), and *encoding conflict* (that occur when similar contents are represented by different formats of data or unit of measures). For example, the PCAs  $\psi_1$  and  $\psi_2$  (see Table 1) deal with, respectively, *data representation conflict* and *naming conflict*.  $\psi_1$  links the property  $\mathbf{idcard}_G$  to the property  $\mathbf{cid}_{\mathbf{RM}}$  using the function **numberTOtext** to convert the data type from *number* to *text*.  $\psi_2$  assigns **contact<sub>G</sub>** to **cphone<sub>RM</sub>**.

ECAs are used to describe which objects/tuples of a base schema should have a corresponding semantically equivalent object/tuple in the target schema. For instance, the relation  $\mathbf{G.CUSTOMER}$  is linked to relation  $\mathbf{RM.CUSTOMER}$  through the ECA  $\psi_3$  presented in Table 1.  $\psi_3$  determines that  $\mathbf{G.CUSTOMER}$  and  $\mathbf{RM.CUSTOMER}$  are equivalent (i.e., for each tuple of  $\mathbf{CUSTOMER}$  of the schema

---

**RM** there is one semantically equivalent tuple in **CUSTOMER** of the schema **G**, and vice-versa).

There are five different kinds of ECAs: equivalence, selection, difference, union, and intersection, being the ECA of union similar to the *natural outer-join* of the usual relational models. For instance, consider the view schema  $S_v$  (not presented in any figure) with the relation **CUSTOMER**, which is related to the relations **CUSTOMER** of the schemata  $S_1$  and  $S_2$  through the ECA  $\psi_4$  shown in Table 1.  $\psi_4$  determines that **CUSTOMER** in  $S_v$  is the union/join of **CUSTOMER** in  $S_1$  and **CUSTOMER** in  $S_2$  (i.e., for each tuple in **CUSTOMER** of  $S_1$  there is one semantically equivalent tuple in **CUSTOMER** of  $S_v$ , or for each tuple in **CUSTOMER** of  $S_2$  there is one semantically equivalent tuple in **CUSTOMER** of  $S_v$ , and vice-versa). In an ECA, any relation/class can appear with a selection condition, which determines the subset of instances of the class/relation being considered. This kind of ECA is especially important to the DW because through it the current instances of the DW can be selected and related to the instances of their sources (which usually do not have historical data).

SCAs are used to describe the summary of a class/relation whose instances are related to the instances of another class/relation by breaking them into logical groups that belong together. They are used to indicate that the relationship between classes/relations involve some type of aggregate functions (called SCA of groupby) or a normalisation process (called SCA of normalisation)<sup>1</sup>. For example, consider the source schema  $S_3$  (not presented in any figure), which contains a denormalised relation **PRODUCT\_SALES**(**product\_number** <sub>$S_3$</sub> , **product** <sub>$S_3$</sub> , **quantity** <sub>$S_3$</sub> , **price** <sub>$S_3$</sub> , **purchase\_order** <sub>$S_3$</sub> ) and the schema **RM** presented in Fig. 2. **PRODUCT\_SALES** holds information about sold items in a purchase order as well as information logically related to products themselves, which could be in another relation, as occurring in schema **RM**. The SCA  $\psi_5$ , displayed in Table 1, determines the relationship between **PRODUCT\_SALES** and **RM.PRODUCT** when a normalisation process is involved (i.e., it determines that **RM.PRODUCT** is a normalisation of  $S_3$ .**PRODUCT\_SALES** based on distinct values of property **product\_number** <sub>$S_3$</sub> ).

ACAs link properties of the target schema to the properties of the base schema when a SCA is used. ACAs associated to SCAs of groupby contains aggregation functions supported by most of the queries languages, like SQL-99 [34] (i.e., *summation*, *maximum*, *minimum*, *average* and *count*). The ACAs, similar to the PCAs, allow for the description of several kinds of situations; therefore, the aggregate expressions can be more detailed than simple property references. Calculations performed can include, for example, ordinary functions (such as sum or concatenate two or more properties' values before applying the aggregate function), and Boolean conditions (e.g., count all male students whose grades are greater or equal to 10).

---

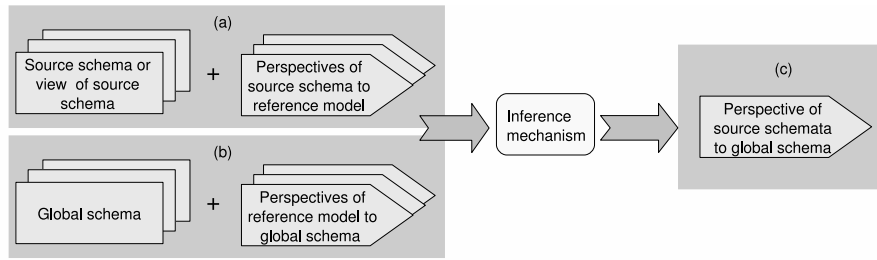
<sup>1</sup> This research also deals with denormalisations, which is defined using *path expressions* (component of the language  $L_S$ ).



## 5 Inference Mechanism

This proposal provides an inference mechanism to automatically infer a new perspective schema (see Fig. 3(c)), given:

1. a set of *origin* schemata and their associated perspective schemata, which take the *origin* schemata as *base* and the reference model as *target* (see Fig. 3(a));
2. a *destination* schema and its associated perspective schema, which take the reference model as *base* and the *destination* schema as *target* (see Fig. 3(b)).



**Fig. 3.** Sketch of the inference mechanism

In context of the Fig. 1, the perspective schema  $\mathbf{P}_{s'1, s'2, s4, \dots, sn|G}$  can be inferred taking as *origin* the schemata  $\mathbf{S}'_1, \mathbf{S}'_2, \mathbf{S}_4, \dots, \mathbf{S}_n$  as well as the perspective schemata  $\mathbf{P}_{s'1|RM}, \mathbf{P}_{s'2|RM}, \mathbf{P}_{s4|RM}, \dots, \mathbf{P}_{sn|RM}$ , and as *destination* the schema  $\mathbf{G}$  as well as the perspective schema  $\mathbf{P}_{RM|G}$ .

The inferred perspective schema will have as *base* a subset of *origin* schemata, and as *target* the *destination* schema. Its ‘*require*’ *declarations* will be the same ‘*require*’ *declarations* present in the perspective schema associated to the *destination* schema. The *MF signatures* and *CAs* of the inferred perspective schema will be automatically generated using a rule-based rewriting system.

### 5.1 The Rewriting System

The rule-based rewriting system is formed by a set of rules having the general form:

$$\mathbf{Rule} : \frac{X \Rightarrow Y}{Z} \quad (\text{read } X \text{ is rewritten as } Y \text{ if } Z \text{ is valid}) , \quad (4)$$

In (4), **Rule** is the name of the rule.  $X$  and  $Y$  can be formed by any of the following expressions: a *CA* pattern expression, a *MF* pattern signature, or a component pattern expression. *CA* pattern expressions and *MF* pattern signatures are expressions conforming to the  $L_{PS}$  syntax to declare, respectively,

CAs and MF signatures, being that some of their elements are variables to be used in a unification process. Component pattern expressions are expressions conforming to the  $L_S$  or the  $L_{PS}$  syntax to represent components that can appear in CAs or in MF signatures (e.g., properties, path expressions, functions with n-ary arguments, values, or conditions of selection (predicates)), being that some of their elements are variables to be used in a unification process.  $Z$  is a condition formed by a set of CA pattern expressions, or expressions of the form  $A \Rightarrow B$  such that  $A$  and  $B$  are component pattern expressions.

A condition  $Z$  is valid when all of its expressions are valid: a) the CA pattern expression is valid if there is a CA, which is declared in one of the perspective schemata associated to the *origin* or *destination* schemata, that unifies with it; b) the expression of the form  $A \Rightarrow B$ , such that  $A$  and  $B$  are component pattern expressions, is valid if there is a rule which unifies with it and which is recursively applied.

When  $X$  and  $Y$  are CA pattern expressions, the rules are rewritten-rules that rewrite CAs in other CAs (RR-CAs). When  $X$  and  $Y$  are MF pattern signatures, the rules are rewritten-rules that rewrite MFs in other MFs (RR-MFs). When  $X$  and  $Y$  are component pattern expressions, the rules are substitution-rules that rewrite components in other components (RR-Cs). The latter are used as an intermediary process by the RR-CAs and RR-MFs.

An example of a RR-CA is as follows:

$$\mathbf{RR-CA1} : \frac{P_{RM|D} [\underline{C^D}] \rightarrow RM [\underline{C^{RM}}] \Rightarrow P_{S|D} [\underline{C^D}] \rightarrow \underline{K^S}}{P_{S|RM} [\underline{C^{RM}}] \rightarrow \underline{K^S}} . \quad (5)$$

In (5), all variables are indicated by an underline.  $\mathbf{D}$  is the *destination* schema,  $\mathbf{RM}$  is the reference model, and  $\mathbf{S}$  is a variable that will be instantiated with some of the *origin* schemata.  $C^D$  is a variable that will be instantiated with a class/relation of the schema  $\mathbf{D}$ ; mutatis mutandis to  $C^{RM}$ .  $\mathbf{K}$  is a variable that will be instantiated with the right side of a CA pattern expression of extension. The letter  $S$  in  $\mathbf{K}^S$  means that all elements in that expression belong to schema  $\mathbf{S}$ . The value of  $\mathbf{S}$  and  $\mathbf{K}$  will depend on which CA, that is declared in the perspective schema associated to some *origin* schemata, will unify with the condition of the rule. The notation in (5) will be used through the paper to explain examples of rules.

The rule **RR-CA1** rewrites an ECA of equivalence connecting a class/relation  $C^D$  to a class/relation  $C^{RM}$ , into an ECA connecting  $C^D$  to a class/relation  $C^S$ ; when is provided an ECA that connect  $C^{RM}$  to  $C^S$ .

An example of a RR-MF is as follows:

$$\mathbf{RR-MF1} : \frac{\begin{array}{l} \mathbf{match} : ((RM [\underline{C^{RM}}], \tau^{RM}) \times (D [\underline{C^D}], \tau^D)) \rightarrow \text{Boolean} \Rightarrow \\ \mathbf{match} : ((S [\underline{C^S}], \tau^S) \times (D [\underline{C^D}], \tau^D)) \rightarrow \text{Boolean} \end{array}}{P_{S|RM} [\underline{C^{RM}}] \rightarrow S [\underline{C^S}]} . \quad (6)$$

---

In (6),  $\tau$  is a data type. The rule **RR-MF1** rewrites a MF signature that matches a class/relation  $C^{\text{RM}}$  to a class/relation  $C^{\text{D}}$ , into a MF signature that matches a class/relation  $C^{\text{S}}$  to  $C^{\text{D}}$ , when is provided an ECA of equivalence that connects  $C^{\text{RM}}$  to  $C^{\text{S}}$ .

An example of a RR-C is as follows:

$$\mathbf{RR-C1} : \frac{\text{RM} [C^{\text{RM}}] \bullet \underline{p^{\text{RM}}} \Rightarrow \underline{A^{\text{S}}}}{P_{\underline{S}|\text{RM}} [C^{\text{RM}}] \bullet \underline{p^{\text{RM}}} \rightarrow \underline{A^{\text{S}}}} . \quad (7)$$

In (7),  $p^{\text{RM}}$  is a variable that will be instantiated with a property of a class/relation (the symbol “ $\bullet$ ” means that  $p^{\text{RM}}$  is defined in  $C^{\text{RM}}$ ).  $A$  is a variable that will be instantiated with a component pattern expression. Similar to  $K^{\text{S}}$  in (5), the letter S in  $A^{\text{S}}$  means that all elements into that expression belong to schema  $S$ . The value of  $S$  and  $A$  will depend on which CA declared in the perspective schema associated to some *origin* schemata will unify with the condition of the rule.

The rule **RR-C1** rewrites a property  $p^{\text{RM}}$  into a property, a path expression, or a function of some *origin* schema; when is provided an PCA that connects  $p^{\text{RM}}$  to that property, path expression, or function. The whole set of proposed rules can be found in [35].

## 5.2 Implementation Issues

A pseudo-code detailing as new CAs are deduced is shown in Fig. 4.

```

1: procedure INFER_CAs( $A^{\text{G}} \rightarrow A^{\text{RM}}, CAs$ )
2:   repeat
3:     find  $A^{\text{G}} \rightarrow A^{\text{Si}}$  applying the inference rule  $R$ :
4:      $R: \frac{A^{\text{G}} \rightarrow A^{\text{RM}} \Rightarrow A^{\text{G}} \rightarrow A^{\text{Si}}}{\text{conditions}};$ 
5:     add  $A^{\text{G}} \rightarrow A^{\text{Si}}$  to  $CAs$ ;
6:   until all rules for rewriting CAs have been tested
7: end procedure

```

**Fig. 4.** A pseudo-code of the inference mechanism to generate new CAs

In Fig. 4,  $G$  is a *destination* schema,  $RM$  the reference model, and  $S_i$ ,  $i \geq 1$ , *origin* schemata. The algorithm tries to find, for each CA of the general form  $A^{\text{G}} \rightarrow A^{\text{RM}}$  (assigning the global schema to the reference model), one or more CAs  $A^{\text{G}} \rightarrow A^{\text{Si}}$  as a result of applying to  $A^{\text{G}} \rightarrow A^{\text{RM}}$  some rule for rewriting CAs. Notice that, in the condition of the rule can appear expressions of the form  $A \Rightarrow B$ . In this case, the recursivity will be present. In order to reduce the search space, all rules of the inference mechanism are oriented. Let us see an example. A new ECA:

$$\mathbf{P}_{\mathbf{S1|G}}[\text{CUSTOMER}] \rightarrow \mathbf{S1}[\text{CUSTOMER}]$$

can be created based on  $\psi_3$  (see Table 1) by using the rule **RR-CA1**, since the CA  $\psi_6$  is defined in perspective schema  $\mathbf{P}_{\mathbf{S1|RM}}$  (see Table 2).

**Table 2.** More examples of correspondence assertions

Extension Correspondence Assertion (ECA)
$\psi_6: \mathbf{P}_{\mathbf{S1 RM}}[\text{CUSTOMER}] \rightarrow \mathbf{S1}[\text{CUSTOMER}]$
$\psi_7: \mathbf{P}_{\mathbf{S2 RM}}[\text{CUSTOMER}] \rightarrow \mathbf{S2}[\text{CUSTOMER}]$

A pseudo-code detailing as new MF signatures are deduced is shown in Fig. 5. In Fig. 5, **K** and **L** are pairs (classes/relations, data type) of the reference model or of the *destination* schema, while **K'** and **L'** are pairs (classes/relations, data type) of some *origin* or *destination* schemata. For each MF **M** that is declared in the perspective schema associated to the *destination* schema, the algorithm tries to find one or more MFs as a result of applying to **M** some rule for rewriting MFs. For instance, two new MF signatures:

$$\begin{aligned} \text{match}: ((\mathbf{S1}[\text{CUSTOMER}], \tau_1) \times (\mathbf{G}[\text{CUSTOMER}], \{\tau_2\})) &\rightarrow \text{Boolean} \\ \text{match}: ((\mathbf{S2}[\text{CUSTOMER}], \tau_1) \times (\mathbf{G}[\text{CUSTOMER}], \{\tau_2\})) &\rightarrow \text{Boolean} \end{aligned}$$

can be created based on MF signature presented in (3) by using the rule **RR-MF1** twice, since as the CAs  $\psi_6$  and  $\psi_7$  are defined, respectively, in perspective schemata  $\mathbf{P}_{\mathbf{S1|RM}}$  and  $\mathbf{P}_{\mathbf{S2|RM}}$  (see Table 2).

```

1: procedure INFER_MFs(match: (K × L) → Boolean, MFs)
2:   repeat
3:     find match: (K' × L') → Boolean applying the inference rule R:
4:        $R: \frac{\text{match}: (K \times L) \rightarrow \text{Boolean} \Rightarrow \text{match}: (K' \times L') \rightarrow \text{Boolean}}{\text{conditions}};$ 
5:     add match: (K' × L') → Boolean to MFs;
6:   until all rules for rewriting MFs have been tested
7: end procedure

```

**Fig. 5.** A pseudo-code of the inference mechanism to generate new MFs

A pseudo-code with the iteration of the process to generate a new perspective schema is shown in Fig. 6. In Fig. 6,  $P_T$  is a perspective schema from the reference model to the global model;  $P_j$ ,  $1 \leq j \leq n$ , are perspective schemata from the sources to the reference model; and  $P_I$  is the inferred perspective schema from the sources to the global model. All elements of the perspective schemata are grouped in lists: *classList*, *relationList*, *keyList*, *caList*, and *mfList*. The three first lists hold ‘require’ declarations of, respectively, classes, relations, and keys

---

```

1: procedure GENERATE_NEW_PERSPECTIVE( $P_T, P_1, \dots, P_n, P_I$ )
2:   for each CA  $A^G \rightarrow A^{RM}$  in  $P_T.caList$  do
3:     infer_CAs( $A^G \rightarrow A^{RM}, \{A^G \rightarrow A^{Si}\}$ );
4:     add_CAs  $A^G \rightarrow A^{Si}$  to  $P_I.caList$ ;
5:   end for
6:   for each MF  $m$  in  $P_T.mfList$  do
7:     infer_MFs( $m, \{m'_i\}$ );
8:     add_MFs  $m'_i$  to  $P_I.mfList$ 
9:   end for
10:  for each  $E$  in  $classList/relationList/keyList$  do
11:    create a require declaration to  $P_I$ ;
12:    add it, appropriately, to  $P_I.classList/$ 
13:       $P_I.relationList/P_I.keyList$ 
14:  end for
15: end procedure

```

**Fig. 6.** A pseudo-code to the creation of inferred perspective schemata

(including foreign keys). *caList* contains CA declarations, and *mfList* has MF signatures.

This mechanism has been developed as part of a proof-of-concept prototype using a Prolog language. Beside the inference mechanism module, the prototype consists of another five modules, such as the *schema manager*, and the *ISCO translator*. The *schema manager* module is employed by the designer to manage the schemata (in language  $L_S$ ) as well as the perspective schemata (in language  $L_{PS}$ ). The *ISCO translator* performs the mapping between schemata written in  $L_S$  or  $L_{PS}$  languages to schemata defined in a language programming called Information Systems Construction (ISCO) language [36]. ISCO is based on a contextual constraint logic programming that allows the construction of information systems. It can define (object) relational schemata, represent data, and transparently access data from various heterogeneous sources in a uniform way, like a mediator system [37]. Thus, it is possible to access data from information sources using the perspective schema in ISCO. Furthermore, once the perspective schema from source schemata to the global schema has been inferred, as well as the new match functions have been implemented, it can be translated to ISCO language and so the data of the global schema can be queried. Details about the prototype can be found in [38].

## 6 Conclusions and Future Works

In this paper, the authors have presented a proposal to automatically connect a global schema to its sources by using an inference mechanism taking into account a reference model. The proposal approach makes clear the mappings that there are in a DW system, and uncouples them in order to make their maintenance easier. Besides, the relationship between the global schema and the

---

source schemata is made explicitly and declaratively through correspondence assertions.

The current approach is particularly useful in data integration systems that define a common or canonical schema, such as in DW systems and in FDBSs. An advantage of the proposed approach is that by using the reference model the designer does not need to have an in depth knowledge of all schemata involved in the DW system or in the FDBSs, since each schema (source or global) needs to be related only to the reference model. Thus, the effort to describe the mappings between schemata is reduced, since mappings between the DW and each sources (and between sources themselves) is automatically done by the inference mechanism. Besides, the DW designer can describe the global system without concerns about where the sources are or how they are stored. The inference mechanism also allows that changes changes in the actual source schemata, in the global schema, or in the mapping between schemata, which are common in the life cycle of any system, are completely transparent to the DW systems (or FDBSs).

Another advantage of our approach is that the process of data integration can be incrementally done in two ways:

1. View schemata can be created as a middle process to relate sections of data that have been integrated (those view schemata, in turn, are related to the reference model). Thus the data integration process can be divided in small parts, rather than being seen as a whole, making the integration task easier.
2. New source schemata can be added gradually, due to the inference mechanism.

A prototype Prolog-based has been developed to allow the description of schemata and perspective schemata in the proposed language as well as to infer new perspective schemata based on other ones. The matching functions can be implemented using Prolog itself or external functions. In addition, the prototype includes translators from the proposed language to the ISCO one. ISCO [36] allows access to heterogeneous data sources and to perform arbitrary computations. Thus, user-queries can be done, in a transparent way, to access the information sources, like occurs in mediator systems [37].

For future work, investigations will be made into how the perspective schemata can be used to automate the materialisation of the data in the DWs or in another repository of a data integration environment. Another important direction for future work is the development of a graphical user-friendly interface to declare the schemata in the proposed language, and thus, to hide some syntax details.

## References

1. Imhoff, C., Galemme, N., Geiger, J.G.: Mastering Data Warehouse Design - Relational and Dimensional Techniques. Wiley Publishing (2003)
2. Geiger, J.G.: Why build a data model? Information Management Magazine (June 2009)

- 
3. Moody, D.L.: From enterprise models to dimensional models: A methodology for data warehouse and data mart design. In: Proc. of the Intl. Workshop on Design and Management of Data Warehouses. (2000)
  4. Pequeno, V.M., Pires, J.C.G.M.: Using perspective schemata to model the ETL process. In: ICMIS 2009 :Intl. Conf. on Management Information Systems, France (June 2009)
  5. Halevy, A.Y., Rajaraman, A., Ordille, J.J.: Data integration: The teenage years. In: VLDB. (2006) 9–16
  6. Stumptner, M., Schrefl, M., Grossmann, G.: On the road to behavior-based integration. In: APCCM: First Asia-Pacific Conf. on Conceptual Modelling. (2004) 15–22
  7. Louie, B., Mork, P., Martin-Sanchez, F., Halevy, A., Tarczy-Hornoch, P.: Data integration and genomic medicine. *Journal of Biomedical Informatics* **40** (2007) 5–13
  8. Naidu, P.G., Palakal, M.J., Hartanto, S.: On-the-fly data integration models for biological databases. In: SAC'07: Proc. of the 2007 ACM symp. on Applied computing, USA, ACM (2007) 118–122
  9. Yoakum-Stover, S., Malyuta, T.: Unified architecture for integrating intelligence data. In: DAMA: Europe Conf., UK (2008)
  10. Vidal, V.M.P., Lóscio, B.F., Salgado, A.C.: Using correspondence assertions for specifying the semantics of XML-based mediators. In: Workshop on Information Integration on the Web. (2001) 3–11
  11. Ives, Z.G., Knoblock, C.A., Minton, S., Jacob, M., Talukdar, P.P., Tuchinda, R., Ambite, J.L., Muslea, M., Gazen, C.: Interactive data integration through smart copy & paste. In: CIDR:4th Biennial Conference on Innovative Data Systems Research, [www.cdrdb.org](http://www.cdrdb.org) (2009)
  12. Mccann, R., Doan, A., Varadarajan, V., Kramnik, E.: Building data integration systems via mass collaboration. In: WebDB: Intl. Workshop on the Web and Databases, USA (2003)
  13. Berger, S., Schrefl, M.: From federated databases to a federated data warehouse system. In: HICSS'08: 41st Annual Hawaii Intl. Conf. on System Sciences, USA, IEEE Computer Society (2008) 394
  14. Dori, D., Feldman, R., Sturm, A.: From conceptual models to schemata: An object-process-based data warehouse construction method. *Inf. Syst.* **33**(6) (2008) 567–593
  15. Malinowski, E., Zimányi, E.: A conceptual model for temporal data warehouses and its transformation to the ER and the object-relational models. *Data knowl. eng.* **64**(1) (2008) 101–133
  16. Pérez, J.M., Berlanga, R., Aramburu, M.J., Pedersen, T.B.: A relevance-extended multi-dimensional model for a data warehouse contextualized with documents. In: DOLAP'05: Proc. of the 8th ACM Intl. Workshop on Data Warehousing and OLAP, USA, ACM (2005) 19–28
  17. Golfarelli, M., Maniezzo, V., Rizzi, S.: Materialization of fragmented views in multidimensional databases. *Data Knowl. Eng.* **49**(3) (2004) 325–351
  18. Husemann, B., Lechtenborger, J., Vossen, G.: Conceptual data warehouse modeling. In: DMDW: Design and Management of Data Warehouses. (2000) 6
  19. Rizzi, S.: Conceptual modeling solutions for the data warehouse. In *Data Warehousing and Mining: Concepts, Methodologies, Tools, and Applications* (2008) 208–227 copyright 2008 by Information Science Reference, formerly known as Idea Group Reference (an imprint of IGI Global).

- 
20. Wrembel, R.: On a formal model of an object-oriented database with views supporting data materialisation. In: Proc. of the Conf. on Advances in Databases and Information Systems. (1999) 109–116
  21. Franconi, E., Kamble, A.: A data warehouse conceptual data model. In: SS-DBM'04: Proc. of the 16th Intl. Conf. on Scientific and Statistical Database Management, USA, IEEE Computer Society (2004) 435–436
  22. Kamble, A.S.: A conceptual model for multidimensional data. In: APCCM'08: Proc. of the 15th on Asia-Pacific Conf. on Conceptual Modelling, Australia, Australian Computer Society, Inc. (2008) 29–38
  23. Sapia, C., Blaschka, M., Höfling, G., Dinter, B.: Extending the E/R model for the multidimensional paradigm. In: Proc. of the Workshops on Data Warehousing and Data Mining. (1999) 105–116
  24. Tryfona, N., Busborg, F., Christiansen, J.G.B.: starER: a conceptual model for data warehouse design. In: DOLAP'99: Proc. of the 2nd ACM Intl. Workshop on Data warehousing and OLAP, USA, ACM (1999) 3–8
  25. Luján-Mora, S., Trujillo, J., Song, I.Y.: A UML profile for multidimensional modelling in data warehouses. *Data Knowl. Eng.* **59**(3) (2005) 725–769
  26. Nguyen, T.B., Tjoa, A.M., Wagner, R.: An object oriented multidimensional data model for OLAP. In: Web-Age Inf. Management. (2000) 69–82
  27. Trujillo, J., Palomar, M., Gómez, J.: Applying object-oriented conceptual modeling techniques to the design of multidimensional databases and OLAP applications. In: WAIM'00: Proc. of the 1st Intl. Conf. on Web-Age Information Management, UK, Springer-Verlag (2000) 83–94
  28. Skoutas, D., Simitsis, A.: Designing ETL processes using semantic web technologies. In: DOLAP'06: Proceedings of the 9th ACM international workshop on Data warehousing and OLAP, USA, ACM (2006) 67–74
  29. Calvanese, D., Dragone, L., Nardi, D., Rosati, R., Trisolini, S.M.: Enterprise modeling and data warehousing in TELECOM ITALIA. *Inf. Syst.* **31**(1) (2006) 1–32
  30. Codd, E.F.: A relational model of data for large shared data banks. In: Communications of the ACM. (1970) 377–387
  31. Cattell, R.G., Barry, D., eds.: The Object Database Standard ODMG 3.0. Morgan Kaufmann Publishers (2000)
  32. Pequeno, V.M., Pires, J.C.G.M.: A formal object-relational data warehouse model. Technical report, Universidade Nova de Lisboa (November 2007)
  33. Pequeno, V.M., Pires, J.C.G.M.: Using perspective schemata to model the ETL process. Technical report, Universidade Nova de Lisboa (2009)
  34. Elmasri, R., Navathe, S.B.: Fundamentals of database systems. 5th edn. Addison Wesley (2006)
  35. Pequeno, V.M., Pires, J.C.G.M.: Reference model and perspective schemata inference for enterprise data integration. Technical report, Universidade Nova de Lisboa (2009)
  36. Abreu, S., Nogueira, V.: Using a logic programming language with persistence and contexts. In: INAP'05: 16th Intl. Conf. on applications of declarative programming and knowledge management. Volume 4369 of Lecture Notes in Computer Science., Springer (2006) 38–47 (Revised Selected Papers).
  37. Wiederhold, G.: Mediators in the architecture of future information systems. *Computer* **25**(3) (1992) 38–49
  38. Pequeno, V.M., Abreu, S., Pires, J.C.G.M.: Using contextual logic programming language to access data in warehousing systems. In: 14th Portuguese Conference on Artificial Intelligence, Portugal (October 2009) (to appear).
-



---

# A Very Compact and Efficient Representation of List Terms for Tabled Logic Programs

João Raimundo and Ricardo Rocha

DCC-FC & CRACS, University of Porto  
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal  
{jraimundo,ricroc}@dcc.fc.up.pt

**Abstract.** Tabling is an implementation technique that overcomes some limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion. A critical component in the implementation of an efficient tabling system is the design of the data structures and algorithms to access and manipulate tabled data. Arguably, the most successful data structure for tabling is tries, which is regarded as a very compact and efficient data structure for term representation. Despite these good properties, we found that, for list terms, we can design even more compact and efficient representations. We thus propose a new representation of list terms for tries that avoids the recursive nature of the WAM representation of list terms in which tries are based. Our experimental results using the YapTab tabling system show a significant reduction in the memory usage for the trie data structures and impressive gains in the running time for storing and loading list terms.

**Key words:** Tabling Logic Programming, Table Space, Implementation.

## 1 Introduction

Tabling [1] is an implementation technique that overcomes some limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion. Tabling has become a popular and successful technique thanks to the ground-breaking work in the XSB Prolog system and in particular in the SLG-WAM engine [2]. The success of SLG-WAM led to several alternative implementations that differ in the execution rule, in the data-structures used to implement tabling, and in the changes to the underlying Prolog engine. Implementations of tabling are now widely available in systems like Yap Prolog, B-Prolog, ALS-Prolog, Mercury and more recently Ciao Prolog.

A critical component in the implementation of an efficient tabling system is the design of the data structures and algorithms to access and manipulate tabled data. Arguably, the most successful data structure for tabling is *tries* [3]. Tries are trees in which common prefixes are represented only once. The trie data structure provides complete discrimination for terms and permits lookup and possibly insertion to be performed in a single pass through a term, hence resulting in a very compact and efficient data structure for term representation.

---

When representing terms in the trie, most tabling engines, like XSB Prolog, Yap Prolog and others, try to mimic the WAM [4] representation of these terms in the Prolog stacks in order to avoid unnecessary transformations when storing/loading these terms to/from the trie. Despite this idea seems straightforward for almost all type of terms, we found that this is not the case for *list terms* (also known as *pair terms*) and that, for list terms, we can design even more compact and efficient representations.

In Prolog, a non-empty list term is formed by two sub-terms, the *head of the list*, which can be any Prolog term, and the *tail of the list*, which can be either a non-empty list (formed itself by a head and a tail) or the *empty list*. WAM based implementations explore this recursive nature of list terms to design a very simple representation at the engine level that allows for very robust implementations of key features of the WAM, like the unification algorithm, when manipulating list terms. However, when representing terms in the trie, the recursive nature of the WAM representation of list terms is negligible as we are most interested in having a compact representation with fast lookup and insertion capabilities.

In this paper, we thus propose a new representation of list terms for tabled data that gets around the recursive nature of the WAM representation of list terms. In our new proposal, a list term is simply represented as the ordered sequence of the term elements in the list, i.e., we only represent the head terms in the sub-lists and avoid representing the sub-lists' tails themselves. Our experimental results show a significant reduction in the memory usage for the trie data structures and impressive gains in the running time for storing and loading list terms with and without compiled tries. We will focus our discussion on a concrete implementation, the YapTab system [5], but our proposals can be easily generalized and applied to other tabling systems.

The remainder of the paper is organized as follows. First, we briefly introduce some background concepts about tries and the table space. Next, we introduce YapTab's new design for list terms representation. Then, we discuss the implications of the new design and describe how we have extended YapTab to provide engine support for it. At last, we present some experimental results and we end by outlining some conclusions.

## 2 Tabling Tries

The basic idea behind tabling is straightforward: programs are evaluated by storing answers for tabled subgoals in an appropriate data space, called the *table space*. Repeated calls to tabled subgoals<sup>1</sup> are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in their table entries. During this process, as further new answers are found, they are stored in their tables and later returned to all repeated calls.

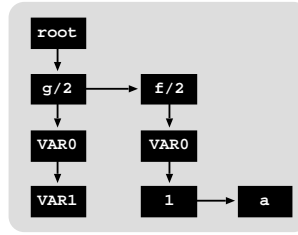
Within this model, the table space may be accessed in a number of ways: **(i)** to find out if a subgoal is in the table and, if not, insert it; **(ii)** to verify whether

---

<sup>1</sup> A subgoal repeats a previous subgoal if they are the same up to variable renaming.

a newly found answer is already in the table and, if not, insert it; and (iii) to load answers to repeated subgoals. With these requirements, a correct design of the table space is critical to achieve an efficient implementation. YapTab uses *tries* which is regarded as a very efficient way to implement the table space [3].

A trie is a tree structure where each different path through the trie data units, the *trie nodes*, corresponds to a term described by the tokens labelling the nodes traversed. For example, the tokenized form of the term  $f(X, g(Y, X), Z)$  is the sequence of 6 tokens  $\langle f/3, VAR_0, g/2, VAR_1, VAR_0, VAR_2 \rangle$  where each variable is represented as a distinct  $VAR_i$  constant [6]. An essential property of the trie structure is that common prefixes are represented only once. Two terms with common prefixes will branch off from each other at the first distinguishing token. Figure 1 shows an example for a trie with three terms. Initially, the trie contains the root node only. Next, we store the term  $f(X, a)$  and three trie nodes are inserted: one for the functor  $f/2$ , a second for variable  $X$  ( $VAR_0$ ) and one last for constant  $a$ . The second step is to store  $g(X, Y)$ . The two terms differ on the main functor, so tries bring no benefit here. In the last step, we store  $f(Y, 1)$  and we save the two common nodes with  $f(X, a)$ .



**Fig. 1.** Representing terms  $f(X, a)$ ,  $g(X, Y)$  and  $f(Y, 1)$  in a trie

To increase performance, YapTab implements tables using two levels of tries: one for subgoal calls; the other for computed answers. More specifically:

- each tabled predicate has a *table entry* data structure assigned to it, acting as the entry point for the predicate’s *subgoal trie*.
- each different subgoal call is represented as a unique path in the subgoal trie, starting at the predicate’s table entry and ending in a *subgoal frame* data structure, with the argument terms being stored within the path’s nodes. The subgoal frame data structure acts as an entry point to the *answer trie*.
- each different subgoal answer is represented as a unique path in the answer trie. Contrary to subgoal tries, answer trie paths hold just the substitution terms for the free variables which exist in the argument terms of the corresponding subgoal call. This optimization is called *substitution factoring* [3].

An example for a tabled predicate  $t/2$  is shown in Fig. 2. Initially, the subgoal trie is empty<sup>2</sup>. Then, the subgoal  $t(X, f(1))$  is called and three trie nodes are inserted: one for variable  $X$  ( $VAR_0$ ), a second for functor  $f/1$  and one last for constant 1<sup>3</sup>. The subgoal frame is inserted as a leaf, waiting for the answers. Next, the subgoal  $t(X, Y)$  is also called. The two calls differ on the second argument, so we need an extra node to represent variable  $Y$  ( $VAR_1$ ) followed by a new subgoal frame. At the end, the answers for each subgoal are stored in the corresponding answer trie as their values are computed. Subgoal  $t(X, f(1))$  has two answers,  $X = f(1)$  and  $X = f(Z)$ , so we need three trie nodes to represent both: a common node for functor  $f/1$  and two nodes for constant 1 and variable  $Z$  ( $VAR_0$ )<sup>4</sup>. For subgoal  $t(X, Y)$  we have four answers, resulting from the combination of the answers  $f(1)$  and  $f(Z)$  for variables  $X$  and  $Y$ , which requires nine trie nodes.

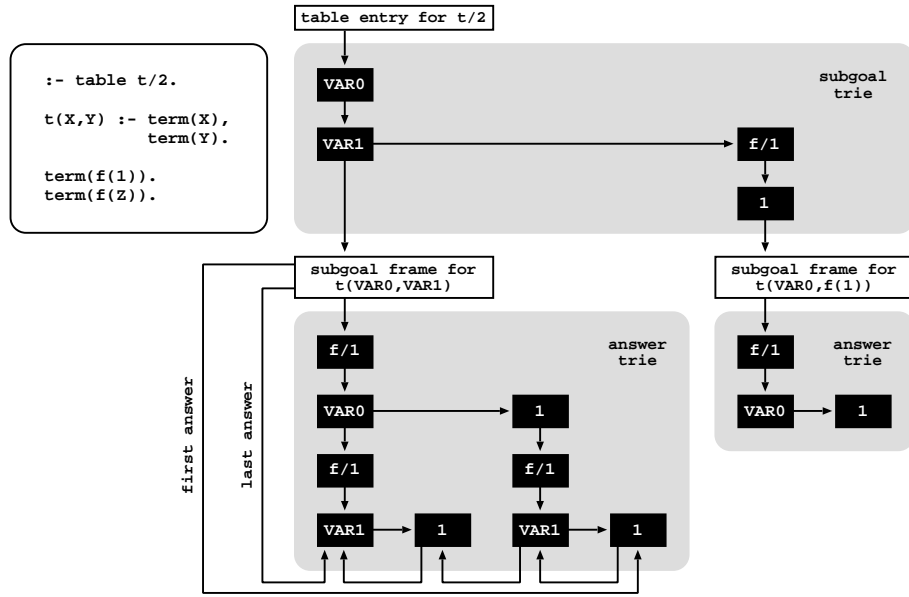


Fig. 2. YapTab table organization

Leaf answer trie nodes are chained in a linked list in insertion time order, so that we can recover answers in the same order they were inserted. The subgoal frame points to the first and last answer in this list. Thus, a repeated call

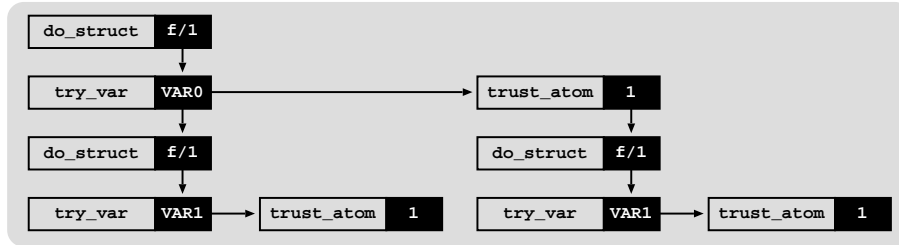
<sup>2</sup> In order to simplify the presentation of the following illustrations, we will omit the representation of the trie root nodes.

<sup>3</sup> Note that for subgoal tries, we can avoid inserting the predicate name, as it is already represented in the table entry.

<sup>4</sup> The way variables are numbered in a trie is specific to each trie and thus there is no correspondence between variables sharing the same number in different tries.

only needs to point at the leaf node for its last loaded answer, and consumes more answers by just following the chain. To load an answer, the trie nodes are traversed in bottom-up order and the answer is reconstructed.

On completion of a subgoal, a strategy exists that avoids answer recovery using bottom-up unification and performs instead what is called a *completed table optimization*. This optimization implements answer recovery by top-down traversing the completed answer tries and by executing dynamically compiled WAM-like instructions from the answer trie nodes. These dynamically compiled instructions are called *trie instructions* and the answer tries that consist of these instructions are called *compiled tries* [3]. Compiled tries are based on the observation that all common prefixes of the terms in a trie are shared during execution of the trie instructions. Thus, when backtracking through the terms of a trie that is represented using the trie instructions, each edge of the trie is traversed only once. Figure 3 shows the compiled trie for subgoal call  $t(VAR_0, VAR_1)$  in Fig. 2.



**Fig. 3.** Compiled trie for subgoal call  $t(VAR_0, VAR_1)$  in Fig. 2

Each trie node is compiled accordingly to its position in the list of sibling nodes and to the term type it represents. For each term type there are four specialized trie instructions. First nodes in a list of sibling nodes are compiled using *try\_?* instructions, intermediate nodes are compiled using *retry\_?* instructions, and last nodes are compiled using *trust\_?* instructions. Trie nodes without sibling nodes are compiled using *do\_?* instructions. For example, for atom terms, the trie instructions are: *try\_atom*, *retry\_atom*, *trust\_atom* and *do\_atom*. As the *try\_?*/*retry\_?*/*trust\_?* instructions denote the choice possibilities when traversing top-down an answer trie, at the engine level, they allocate and manipulate a choice point in a manner similar to the generic *try/retry/trust* WAM instructions, but here the failure continuation points to the next sibling node. The *do\_?* instructions denote no choice and thus they don't allocate choice points.

The implementation of tries requires the following fields per trie node: a first field (**token**) stores the token for the node, a second (**child**), third (**parent**) and fourth (**sibling**) fields store pointers respectively to the first child node, to the parent node, and to the next sibling node. For the answer tries, an additional fifth field (**code**) is used to support compiled tries.

### 3 Representation of List Terms

In this section, we introduce YapTab's new design for the representation of list terms. In what follows, we will refer to the original design as *standard lists* and to our new design as *compact lists*. Next, we start by briefly introducing how standard lists are represented in YapTab and then we discuss in more detail the new design for representing compact lists.

#### 3.1 Standard Lists

YapTab follows the seminal WAM representation of list terms [4]. In YapTab, list terms are recursive data structures implemented using *pairs*, where the first pair element, the *head of the list*, represents a list element and the second pair element, the *tail of the list*, represents the list continuation term or the end of the list. In YapTab, the end of the list is represented by the empty list atom `[]`. At the engine level, a pair is implemented as a pointer to two contiguous cells, the first cell representing the head of the list and the second the tail of the list. In YapTab, as we will see next, the tail of a list can be any term. Figure 4(a) shows YapTab's WAM representation for lists in more detail.

Alternatively to the standard notation for list terms, we can use the pair notation  $[H|T]$ , where  $H$  denotes the head of the list and  $T$  denotes its tail. For example, the list term `[1, 2, 3]` in Fig. 4 can be alternatively denoted as  $[1|[2, 3]]$ ,  $[1|[2|[3]]]$  or  $[1|[2|[3|[]]]]$ . The pair notation is also useful when the tail of a list is neither a continuation list nor the empty list. See, for example, the list term `[1, 2|3]` in Fig. 4(a) and its corresponding WAM representation. In what follows, we will refer to these lists as *term-ending lists* and to the lists ending with the empty list atom as *empty-ending lists*.

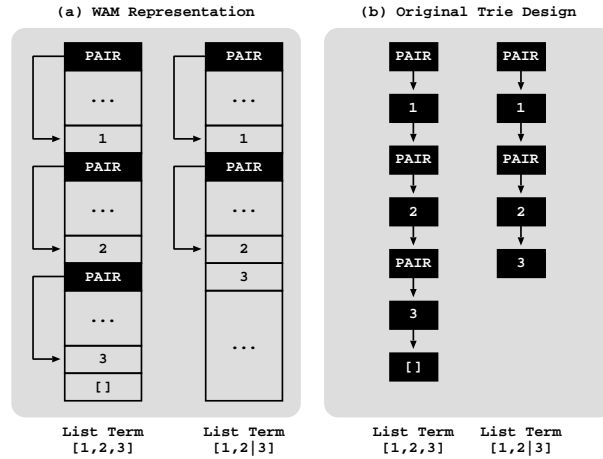


Fig. 4. YapTab's WAM representation and original trie design for standard lists

Regarding the trie representation of lists, the original YapTab design, as most tabling engines, including XSB Prolog, tries to mimic the corresponding WAM representation. This is done by making a direct correspondence between each pair pointer at the engine level and a trie node labelled with the special token PAIR. For example, the tokenized form of the list term  $[1, 2, 3]$  is the sequence of 7 tokens  $\langle \text{PAIR}, 1, \text{PAIR}, 2, \text{PAIR}, 3, [] \rangle$ . Figure 4(b) shows in more detail YapTab’s original trie design for the list terms represented in Fig. 4(a).

### 3.2 Compact Lists

In this section, we introduce the new design for the representation of list terms. The discussion we present next tries to follow the different approaches that we have considered until reaching our current final design. The key idea common to all these approaches is to avoid the recursive nature of the WAM representation of list terms and have a more compact representation where the unnecessary intermediate PAIR tokens are removed.

Figure 5 shows our initial approach. In this first approach, all intermediate PAIR tokens are removed and a compact list is simply represented by its term elements surrounded by a begin and an end list mark, respectively, the BLIST and ELIST tokens. Figure 5(a) shows the tokenized form of the empty-ending list  $[1, 2, 3]$  that now is the sequence of 6 tokens  $\langle \text{BLIST}, 1, 2, 3, [], \text{ELIST} \rangle$  and the tokenized form of the term-ending list  $[1, 2|3]$  that now is the sequence of 5 tokens  $\langle \text{BLIST}, 1, 2, 3, \text{ELIST} \rangle$ .

Our approach clearly outperforms the standard lists representation when representing individual lists (except for the base cases of list terms of sizes 1 to 3). It requires about half the nodes when representing individual lists. For an empty-ending list of  $S$  elements, standard lists require  $2S + 1$  trie nodes and

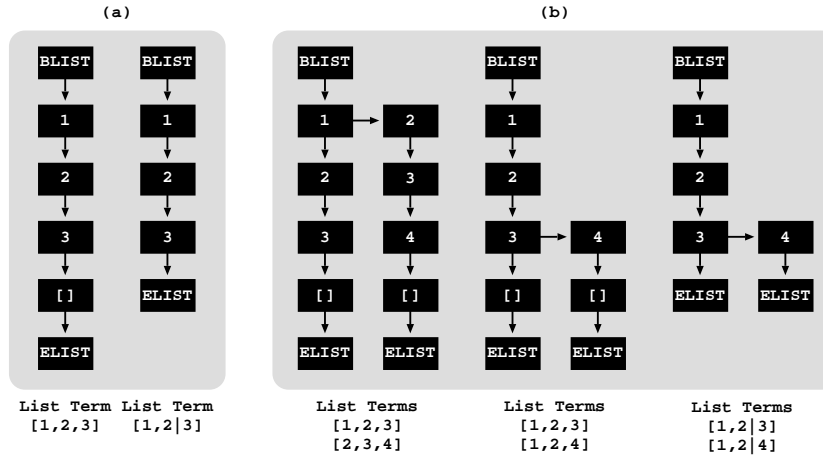


Fig. 5. Trie design for compact lists: initial approach

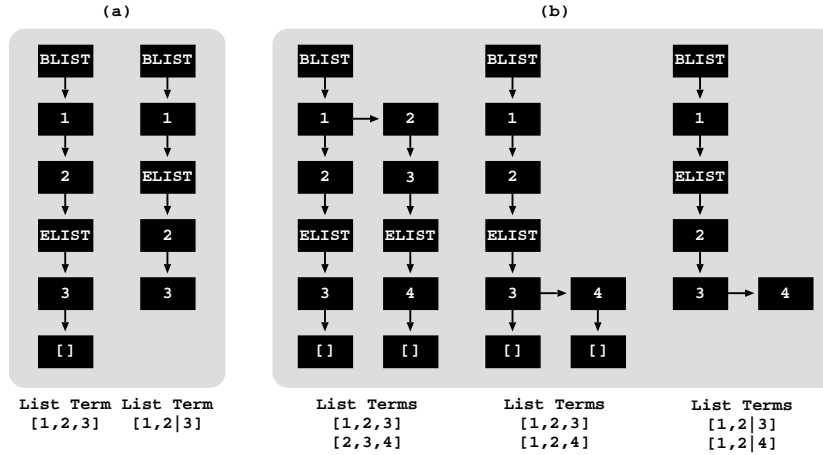
compact lists require  $S + 3$  nodes. For a term-ending list of  $S$  elements, standard lists require  $2S - 1$  trie nodes and compact lists require  $S + 2$  nodes.

Next, in Fig. 5(b) we try to illustrate how this approach behaves when we represent more than a list in the same trie. It presents three different situations: the first situation shows two lists with the first element different (a kind of worst case scenario); the second and third situations show, respectively, two empty-ending and two term-ending lists with the last element different (a kind of best case scenario).

Now consider that we generalize these situations and represent in the same trie  $N$  lists of  $S$  elements each. Our approach is always better for the first situation, but this may not be the case for the second and third situations. For the second situation (empty-ending lists with last element different), standard lists require  $2N + 2S - 1$  trie nodes and compact lists require  $3N + S$  nodes and thus, if  $N > S - 1$  then standard lists is better. For the third situation (term-ending lists with last element different), standard lists require  $N + 2S - 2$  trie nodes and compact lists require  $2N + S$  nodes and again, if  $N > S - 2$  then standard lists is better.

The main problem with this approach is that it introduces an extra token in the end of each list, the ELIST token, that do not exists in the representation of standard lists. To avoid this problem, we have redesigned our compact lists representation in such a way that the ELIST token appears only once for lists with the last element different. Figure 6 shows our second approach for the representation of compact lists.

In this second approach, a compact list still contains the begin and end list tokens, BLIST and ELIST, but now the ELIST token plays the same role of the last PAIR token in standard lists, i.e., it marks the last pair term in the list. Figure 6(a) shows the new tokenized form of the empty-ending list  $[1, 2, 3]$



**Fig. 6.** Trie design for compact lists: second approach

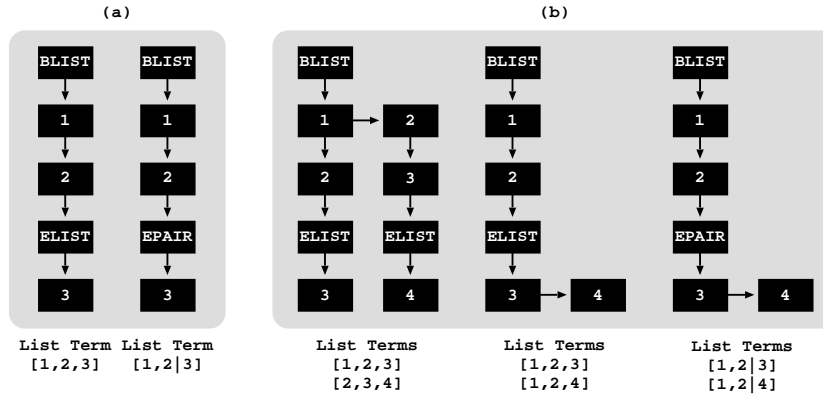


that now is  $\langle \text{BLIST}, 1, 2, \text{ELIST}, 3, [ ] \rangle$ , and the new tokenized form of the term-ending list  $[1, 2|3]$  that now is  $\langle \text{BLIST}, 1, \text{ELIST}, 2, 3 \rangle$ .

Figure 6(b) illustrates again the same three situations showing how this second approach behaves when we represent more than a list in the same trie. For the first situation, the second approach is identical to the initial approach. For the second and third situations, the second approach is not only better than the initial approach, but also better than the standard lists representation (except for the base cases of list terms of sizes 1 and 2).

Consider again the generalization to represent in the same trie  $N$  lists of  $S$  elements each. For the second situation (empty-ending lists with last element different), compact lists now require  $2N + S + 1$  trie nodes (the initial approach for compact lists require  $3N + S$  nodes and standard lists require  $2N + 2S - 1$  nodes). For the third situation (term-ending lists with last element different), compact lists now require  $N + S + 1$  trie nodes (the initial approach for compact lists require  $2N + S$  nodes and standard lists require  $N + 2S - 2$  nodes). Despite these better results, this second approach still contains some drawbacks that can be improved. Figure 7 shows our final approach for the representation of compact lists.

In this final approach, we have redesigned our previous approach in such a way that the empty list token  $[ ]$  was avoided in the representation of empty-ending lists. Note that, in our previous approaches, the empty list token is what allows us to distinguish between empty-ending lists and term-ending lists. As we need to maintain this distinction, we cannot simply remove the empty list token from the representation of compact lists. To solve that, we use a different end list token, EPAIR, for term-ending lists. Hence, the ELIST token marks the last element in an empty-ending list and the EPAIR token marks the last element in a term-ending list. Figure 7(a) shows the new tokenized form of the empty-ending list  $[1, 2, 3]$  that now is  $\langle \text{BLIST}, 1, 2, \text{ELIST}, 3 \rangle$ , and the new tokenized form of the term-ending list  $[1, 2|3]$  that now is  $\langle \text{BLIST}, 1, 2, \text{EPAIR}, 3 \rangle$ .



**Fig. 7.** Trie design for compact lists: final approach

Figure 7(b) illustrates again the same three situations showing how this final approach behaves when we represent more than a list in the same trie. For the three situations, this final approach clearly outperforms all the other representations for standard and compact lists. For lists with the first element different (first situation), it requires  $NS + N + 1$  trie nodes for both empty-ending and term-ending lists. For lists with the last element different (second and third situations), it requires  $N + S + 1$  trie nodes for both empty-ending and term-ending lists. Table 1 summarizes the comparison between all the approaches regarding the number of trie nodes required to represent in the same trie  $N$  list terms of  $S$  elements each.

<i>List Terms</i>	<i>Standard Lists</i>	<i>Compact Lists</i>		
		<i>Initial</i>	<i>Second</i>	<i>Final</i>
<i>First element different</i>				
$N [E_1, \dots, E_{S-1}, E_S]$	$2NS + 1$	$NS + 2N + 1$	$NS + 2N + 1$	$NS + N + 1$
$N [E_1, \dots, E_{S-1}   E_S]$	$2NS - 2N + 1$	$NS + N + 1$	$NS + N + 1$	$NS + N + 1$
<i>Last element different</i>				
$N [E_1, \dots, E_{S-1}, E_S]$	$2N + 2S - 1$	$3N + S$	$2N + S + 1$	$N + S + 1$
$N [E_1, \dots, E_{S-1}   E_S]$	$N + 2S - 2$	$2N + S$	$N + S + 1$	$N + S + 1$

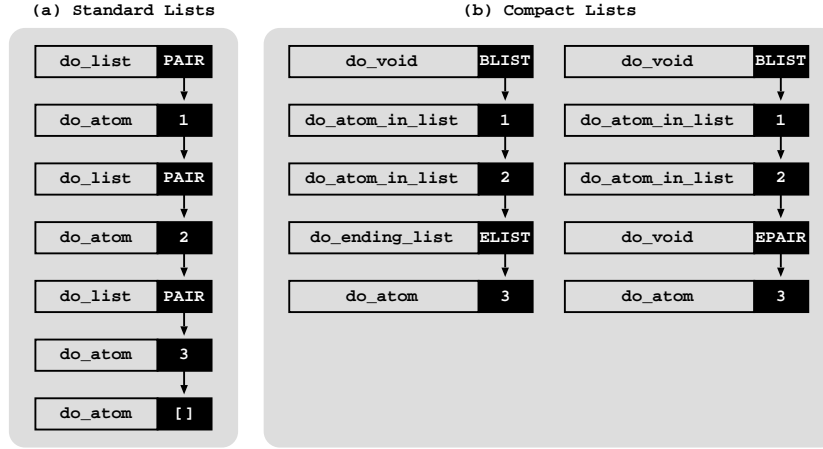
**Table 1.** Number of trie nodes to represent in the same trie  $N$  list terms of  $S$  elements each, using the standard lists representation and the three compact lists approaches

## 4 Compiled Tries for Compact Lists

We then discuss the implications of the new design in the completed table optimization and describe how we have extended YapTab to support compiled tries for compact lists.

We start by presenting in Fig. 8(a) the compiled trie code for the standard list  $[1, 2, 3]$ . For standard lists, each PAIR token is compiled using one of the *?\_list* trie instructions. At the engine level, these instructions create a new pair term in the heap stack and then they bind the previous tail element of the list being constructed to the new pair.

Figure 8(b) shows the new compiled trie code for compact lists. In the new representation for compact lists, the PAIR tokens were removed. Hence, we need to include the pair terms creation step in the trie instructions associated with the elements in the list, except for the last list element. To do that, we have extended the set of trie instructions for each term type with four new specialized trie instructions: *try\_?\_in\_list*, *retry\_?\_in\_list*, *trust\_?\_in\_list* and *do\_?\_in\_list*. For example, for atom terms, the new set of trie instructions is: *try\_atom\_in\_list*, *retry\_atom\_in\_list*, *trust\_atom\_in\_list* and *do\_atom\_in\_list*. The ELIST tokens are compiled using *?\_ending\_list* instructions and the BLIST and EPAIR tokens are compiled using *?\_void* instructions. At the engine level, the *?\_ending\_list*



**Fig. 8.** Comparison between the compiled trie code for standard and compact lists

instructions also create a new pair term in the heap stack to be bound with the previous tail element of the list being constructed. Besides, in order to denote the end of the list, they bind the tail of the new pair with the empty list atom `[]`. The `?_void` instructions do nothing. Note however that the trie nodes for the tokens `BLIST` and `EPAIR` cannot be avoided because they are necessary to mark the beginning and the end of list terms when traversing the answer tries bottom-up, and to distinguish between a term  $t$  and the list term whose first element is  $t$ .

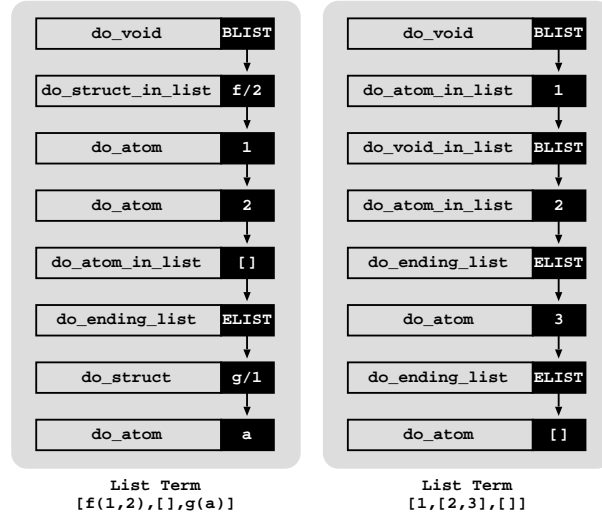
Next we present in Fig. 9, two more examples showing how list terms including compound terms, the empty list term and sub-lists are compiled using the compact lists representation. The tokenized form of the list term  $[f(1, 2), [], g(a)]$  is the sequence of 8 tokens  $\langle \text{BLIST}, f/2, 1, 2, [], \text{ELIST}, g/1, a \rangle$  and the tokenized form of the list term  $[1, [2, 3], []]$  is the sequence of 8 tokens  $\langle \text{BLIST}, 1, \text{BLIST}, 2, \text{ELIST}, 3, \text{ELIST}, [] \rangle$ . To see how the new trie instructions for compact lists are associated with the tokens representing list elements, please consider a tokenized form where the tokens representing common list elements are explicitly aggregated:

$$\begin{aligned}
 [f(1, 2), [], g(a)]: & \quad \langle \text{BLIST}, \langle f/2, 1, 2 \rangle, [], \text{ELIST}, \langle g/1, a \rangle \rangle \\
 [1, [2, 3], []]: & \quad \langle \text{BLIST}, 1, \langle \text{BLIST}, 2, \text{ELIST}, 3 \rangle, \text{ELIST}, [] \rangle.
 \end{aligned}$$

The tokens that correspond to first tokens in each list element are the ones that need to be compiled with the new `?_in_list` trie instructions (please see Fig. 9 for full details).

## 5 Experimental Results

We next present some experimental results comparing YapTab with and without support for compact lists. The environment for our experiments was an Intel(R)



**Fig. 9.** Compiled trie code for the compact lists  $[f(1,2), [], g(a)]$  and  $[1, [2,3], []]$

Core(TM)2 Quad 2.66GHz with 2 GBytes of main memory and running the Linux kernel 2.6.24-24-generic with YapTab 6.0.0.

To put the performance results in perspective, we have defined a top query goal that calls recursively a tabled predicate `list_terms/1` that simply stores in the table space list terms facts. We experimented the `list_terms/1` predicate using 100,000 list terms of sizes 60, 80 and 100 for empty-ending and term-ending lists with the first and with the last element different.

Tables 2 and 3 show the table memory usage (columns *Mem*), in KBytes, and the running times, in milliseconds, to store (columns *Store*) the tables (first execution) and to load from the tables (second execution) the complete set of answers without (columns *Load*) and with (columns *Cmp*) compiled tries for YapTab using standard lists (column *YapTab*) and using the final design for compact lists (column *YapTab+CL / YapTab*). For compact lists, we only show the memory and running time ratios over YapTab using standard lists. The running times are the average of five runs.

As expected, the memory results obtained in these experiments are consistent with the formulas presented in Table 1. The results in Tables 2 and 3 clearly confirm that the new trie design based on compact lists can decrease significantly memory usage when compared with standard lists. In particular, for empty-ending lists, with the first and with the last element different, and for term-ending lists with the first element different, the results show an average reduction of 50%. For term-ending lists with the last element different, memory usage is almost the same. This happens because the memory reduction obtained in the representation of the common list elements (respectively 59, 79 and 99 elements

<i>Empty-Ending Lists</i>	<i>YapTab</i>				<i>YapTab+CL / YapTab</i>			
	<i>Mem</i>	<i>Store</i>	<i>Load</i>	<i>Cmp</i>	<i>Mem</i>	<i>Store</i>	<i>Load</i>	<i>Cmp</i>
<i>First element different</i>								
100,000 [ $E_1, \dots, E_{60}$ ]	234,375	1036	111	105	<b>0.51</b>	<b>0.52</b>	<b>0.71</b>	<b>0.69</b>
100,000 [ $E_1, \dots, E_{80}$ ]	312,500	1383	135	128	<b>0.51</b>	<b>0.52</b>	<b>0.73</b>	<b>0.64</b>
100,000 [ $E_1, \dots, E_{100}$ ]	390,625	1733	166	170	<b>0.51</b>	<b>0.53</b>	<b>0.67</b>	<b>0.55</b>
<i>Last element different</i>								
100,000 [ $E_1, \dots, E_{60}$ ]	3,909	138	50	7	<b>0.50</b>	<b>0.75</b>	<b>0.64</b>	<b>0.56</b>
100,000 [ $E_1, \dots, E_{80}$ ]	3,909	171	71	8	<b>0.50</b>	<b>0.81</b>	<b>0.61</b>	<b>0.40</b>
100,000 [ $E_1, \dots, E_{100}$ ]	3,910	211	82	9	<b>0.50</b>	<b>0.76</b>	<b>0.62</b>	<b>0.44</b>

**Table 2.** Table memory usage (in KBytes) and store/load times (in milliseconds) for empty-ending lists using YapTab with and without support for compact lists

<i>Term-Ending Lists</i>	<i>YapTab</i>				<i>YapTab+CL / YapTab</i>			
	<i>Mem</i>	<i>Store</i>	<i>Load</i>	<i>Cmp</i>	<i>Mem</i>	<i>Store</i>	<i>Load</i>	<i>Cmp</i>
<i>First element different</i>								
100,000 [ $E_1, \dots, E_{59} E_{60}$ ]	230,469	1028	113	97	<b>0.52</b>	<b>0.54</b>	<b>0.67</b>	<b>0.64</b>
100,000 [ $E_1, \dots, E_{79} E_{80}$ ]	308,594	1402	138	134	<b>0.51</b>	<b>0.53</b>	<b>0.69</b>	<b>0.63</b>
100,000 [ $E_1, \dots, E_{99} E_{100}$ ]	386,719	1695	162	163	<b>0.51</b>	<b>0.55</b>	<b>0.66</b>	<b>0.60</b>
<i>Last element different</i>								
100,000 [ $E_1, \dots, E_{59} E_{60}$ ]	1,956	121	45	4	1.00	<b>0.86</b>	<b>0.82</b>	1.00
100,000 [ $E_1, \dots, E_{79} E_{80}$ ]	1,956	150	59	4	1.00	<b>0.88</b>	<b>0.72</b>	1.00
100,000 [ $E_1, \dots, E_{99} E_{100}$ ]	1,957	194	96	4	1.00	<b>0.88</b>	<b>0.53</b>	1.00

**Table 3.** Table memory usage (in KBytes) and store/load times (in milliseconds) for term-ending lists using YapTab with and without support for compact lists

in these experiments) is residual when compared with the number of different last elements (100,000 in these experiments).

Regarding running time, the results in Tables 2 and 3 indicate that compact lists can achieve impressive gains for storing and loading list terms. In these experiments, the storing time using compact lists is around 2 times faster for list terms with the first element different, and around 1.15 to 1.30 times faster for list terms with the last element different. Note that this is the case even for term-ending lists, where there is no significant memory reduction. This happens because the number of nodes to be traversed when navigating the trie data structures for compact lists is considerably smaller than the number of nodes for standard lists.

These results also indicate that compact lists can outperform standard lists for loading terms, both with and without compiled tries, and that the reduction on the running time seems to decrease proportionally to the size of the list terms being considered. The exception is compiled tries for term-ending lists with the last element different, but the 4 milliseconds of the execution time in these experiments is too small to be taken into consideration.

---

## 6 Conclusions

We have presented a new and more compact representation of list terms for tabled data that avoids the recursive nature of the WAM representation by removing unnecessary intermediate pair tokens. Our presentation followed the different approaches that we have considered until reaching our current final design. We focused our discussion on a concrete implementation, the YapTab system, but our proposals can be easily generalized and applied to other tabling systems. Our experimental results are quite interesting, they clearly show that with compact lists, it is possible not only to reduce the memory usage overhead, but also the running time of the execution for storing and loading list terms, both with and without compiled tries.

Further work will include exploring the impact of our proposal in real-world applications, such as, the recent works on Inductive Logic Programming [7] and probabilistic logic learning with the ProbLog language [8], that heavily use list terms to represent, respectively, hypotheses and proofs in trie data structures.

## Acknowledgements

This work has been partially supported by the FCT research projects STAMPA (PTDC/EIA/67738/2006) and JEDI (PTDC/EIA/66924/2006).

## References

1. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM* **43**(1) (1996) 20–74
2. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems* **20**(3) (1998) 586–634
3. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Programming* **38**(1) (1999) 31–54
4. Ait-Kaci, H.: Warren’s Abstract Machine – A Tutorial Reconstruction. The MIT Press (1991)
5. Rocha, R., Silva, F., Santos Costa, V.: On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming* **5**(1 & 2) (2005) 161–205
6. Bachmair, L., Chen, T., Ramakrishnan, I.V.: Associative Commutative Discrimination Nets. In: *International Joint Conference on Theory and Practice of Software Development*. Number 668 in LNCS, Springer-Verlag (1993) 61–74
7. Fonseca, N.A., Camacho, R., Rocha, R., Costa, V.S.: Compile the hypothesis space: do it once, use it often. *Fundamenta Informaticae* **89**(1) (2008) 45–67
8. Kimmig, A., Costa, V.S., Rocha, R., Demoen, B., Raedt, L.D.: On the Efficient Execution of ProbLog Programs. In: *International Conference on Logic Programming*. Number 5366 in LNCS, Springer-Verlag (2008) 175–189

---

# Inspection Points and Meta-Abduction in Logic Programs

Luís Moniz Pereira and Alexandre Miguel Pinto  
{lmp|amp}@di.fct.unl.pt

Centro de Inteligência Artificial (CENTRIA)  
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal

**Abstract.** In the context of abduction in Logic Programs, when finding an abductive solution for a query, one may want to check too whether some other literals become *true* (or *false*) as a consequence, strictly within the abductive solution found, that is without performing additional abductions, and without having to produce a complete model to do so. That is, such consequence literals may consume, but not produce, the abduced literals of the solution. We show how this type of reasoning requires a new mechanism, not provided by others already available. To achieve it, we present the concept of Inspection Point in Abductive Logic Programs, and show, by means of examples, how one can employ it to investigate side-effects of interest (the *inspection points*) in order to help choose among abductive solutions. The touchstone of enabling inspection points can be construed as meta-abduction, by (meta-)abducting an “abduction” to check (i.e. to passively verify) that a certain concrete abduction is indeed adopted in a purported abductive solution. We show how to implement inspection points on top of already existing abduction solving systems — ABDUAL and XSB-XASP — in a way that can be adopted by other systems too.

**Keywords:** Logic Programs, Abduction, Side-Effects.

## 1 Introduction

We begin by presenting the motivation, plus some background notation and definitions follow. Then issues of reasoning with logic programs are addressed in section 2, in particular, we take a look at abductive reasoning and the nature of backward and forward chaining and their relationship to query answering in an abductive framework. In section 3 we introduce inspection points, illustrate their need and their use with examples, and provide a declarative semantics. In section 4 we describe in detail our implementation of inspection points and illustrate its workings with an example. To close the paper we add conclusions, comparisons, and an elaboration on the possible use of inspection points in future work is sketched.

### 1.1 Motivation

Sometimes, besides needing to abductively discover which hypotheses to assume in order to satisfy some condition, we may also want to know some of the side-effects of those assumptions; in fact, this is rather a rational thing to do. But, most of the

---

time, we do not wish to know *all* possible side-effects of our assumptions, as some of them will be irrelevant to our concern. Likewise, the side-effects inherent in abductive explanations might not all be of interest. One application of abductive reasoning is that of finding which actions to perform, their names being coded as abducibles.

**Example 1. Relevant and irrelevant side-effects.** Consider this logic program where *drink\_water* and *drink\_beer* are abducibles.

```

← thirsty, not drink.           % This is an Integrity Constraint
wet_glass ← use_glass.         use_glass ← drink.
drink ← drink_water.           drink ← drink_beer.
thirsty.                        drunk ← drink_beer.
unsafe_drive ← drunk.

```

Suppose we want to satisfy the Integrity Constraint(IC), and also to check if we get drunk or not. However, we do not care about the glass becoming wet — that being completely irrelevant to our current concern. In this case, computation of whole models is a waste of time, because we are interested only in a subset of the program’s literals. Moreover, in this example, we may simply want to know the side-effects of the possible actions in order to decide (to drive or not to drive) **after** we know which side-effects are true. In such a case, we do not want to simply introduce an IC expressed as  $\leftarrow \text{not } \text{unsafe\_drive}$  because that would always impose abducting *not drink\_beer*. We want to allow all possible solutions for the single IC  $\leftarrow \text{thirsty, not drink}$  and then check for the side-effects of each abductive solution.

What we need is an inspection mechanism which permits to check the truth value of given literals as a consequence of the abductions made to satisfy a given query plus any ICs, but without further abducting. This will be achieved just through the *inspect/1* meta-predicate, by introducing the IC  $\leftarrow \text{inspect}(\text{not } \text{unsafe\_drive})$ .

## 1.2 Background Notation and Definitions

**Definition 1. Logic Rule.** A Logic Rule has the general form

$$H \leftarrow B_1, \dots, B_n, \text{not } C_1, \dots, \text{not } C_m$$

where  $H$  is an atom, and the  $B_i$  and  $C_j$  are atoms.

We call  $H$  the head of the rule, and  $B_1, \dots, B_n, \text{not } C_1, \dots, \text{not } C_m$  its body. Throughout this paper we use ‘*not*’ to denote default negation. When the body of a rule is empty, we say its head is a fact and we write the rule just as  $H$ . When the head is empty, we designate the rule an Integrity Constraint (IC).

**Definition 2. Logic Program.** A Logic Program (LP for short)  $P$  is a (possibly infinite) set of ground Logic Rules, where non-ground rules stand for all their ground instances.

In this paper we consider solely Normal LPs (NLPs), those whose heads of rules are positive literals, i.e. simple atoms, or empty. In the next sections we focus on abductive logic programs, i.e., those with abducibles. Abducibles are chosen (by a system specific declaration) literals, not defined by any rules and correspond to hypotheses that one can independently assume or not — apart from eventual ICs affecting them. Abducibles or their default negations may appear in bodies of rules, just like any other literal.



---

## 2 Abductive Reasoning with Logic Programs

Logic Programs have been used for a few decades now in knowledge representation and reasoning. Amongst the most common kinds of reasoning performed using them, we can find deduction, induction and abduction. When query answering, if we know that the underlying semantics is *relevant*, i.e. guarantees it is enough to use only the rules relevant to the query (those in its call-graph) to assess its truthfulness, then we need not compute a whole model in order to find an answer to our query: it suffices to use just the call-graph relevant part of the program. This way of top-down finding a solution to a query, dubbed “backward chaining”, is possible only when the underlying semantics is *relevant* in the above sense, because the existence of a full model is guaranteed.

Currently, the standard 2-valued semantics used by the logic programming community is Stable Models [8]. Its properties are well known and there are efficient implementations (such as *DLV* and *SModels* [3, 11]). However, Stable Models (SMs) miss some important properties, both from the theoretical and practical perspectives: guarantee of model existence for every NLP, relevancy and cumulativity. Since SMs do not enjoy relevancy they cannot use just backward chaining for query answering. This means that it may incur in waste of computational resources, when extra time and memory are used to compute parts of the model which may be irrelevant to the query.

When performing abductive reasoning, we want to find, by need only (via backward chaining), one possible set of conditions (abductive literals of the program to be assumed either *true* or *false*) sufficient to entail our query. However, sometimes we also want to know which are (some of) the consequences (side-effects, so to speak) of such conditions. I.e., we want to know the truth value of some other literals, not part of the query’s call graph, whose truth-value may be determined by the abductive conditions found. In some cases, we might be interested in knowing every possible side-effect — the truth-value of every literal in a complete model satisfying the query and ICs. In other situations though, our focus is only in some specific side-effects of abductions performed.

In our approach, the side-effects of interest are explicitly indicated by the user by wrapping the corresponding goals within reserved construct *inspect/1*. It is advantageous, from a computational point of view, to be able to compute only the truth-value of the important side-effects instead of the whole model, so as not to waste precious time and computational resources. This is possible whenever the underlying semantics guarantees model existence, and enjoys relevance.

### 2.1 Abduction

Abduction, or inference to the best explanation, is a reasoning method whereby one chooses the hypotheses that would, if true, best explain the observed evidence. In LPs, abductive hypotheses (or *abducibles*) are named literals of the program which have no rules. They can be considered *true* or *false* for the purpose of answering a query. Abduction in LPs ([1, 4, 5, 9, 10]) can naturally be used in a top-down query-oriented proof-procedure to find an (abductive) answer to a query, where abducibles are leafs in the call dependency graph. The Well-Founded Semantics (WFS) [7], which enjoys relevancy, allows for abductive query answering. We used it in the specific implementation

---

described in section 4 based on ABDUAL [1]. Though WFS is 3-valued, the abduction mechanism it employs can be, and in our case is, 2-valued.

Because they do not depend on any other literal in the program, abducibles can be modeled in a Logic Program system without specific abduction mechanisms by including for each abducible an even loop over default negation, e.g.,

$$abducible \leftarrow not\ abducible\_not. \quad abducible\_not \leftarrow not\ abducible.$$

where *neg\_abducible* is a new abducible atom, representing the (abducible) negation of the abducible. This way, under the SM semantics, a program may have models where some *abducible* is *true* and another where it is *false*, i.e. *neg\_abducible* is *true*. If there are  $n$  abducibles in the program, there will be  $2^n$  models corresponding to all the possible combinations of *true* and *false* for each. Under the WFS without a specific abduction mechanism, e.g. the one available in ABDUAL, both *abducible* and *neg\_abducible* remain *undefined* in the Well-Founded Model (WFM), but may hold (as alternatives) in some Partial Stable Models.

Using the SM semantics abduction is done by guessing the truth-value of each abducible and providing the whole model and testing it for stability; whereas using the WFS with abduction, it can be performed *by need*, induced by the top-down query solving procedure, solely for the relevant abducibles — i.e., irrelevant abducibles are left unconsidered. Thus, top-down abductive query answering is a means of finding those abducible values one might commit to in order to satisfy a query.

An additional situation, addressed in this paper, is when one wishes to only passively determine which abducibles would be sufficient to satisfy some goal but without actually abducting them, just consuming other goals' needed and produced abductions. The difference is subtle but of importance, and it requires a new construct. Its mechanism, of *inspecting without abducting*, can be conceived and implemented through *meta-abduction*, and is discussed in detail in the sequel.

## 2.2 Backward and Forward Chaining

Abductive query-answering is intrinsically a backward-chaining process, a top-down dependency-graph oriented proof-procedure. Finding the side-effects of a set of abductive assumptions may be conceptually envisaged as forward-chaining, as it consists of progressively deriving conclusions from the assumptions until the truth value of the chosen side-effect literals is determined.

The problem with full-fledged forward-chaining is that too many (often irrelevant) conclusions of a model are derived. Wasting time and resources deriving them only to be discarded afterwards is a flagrant setback. Worse, there may be many alternative models satisfying an abductive query (and the ICs) whose differences just repose on irrelevant conclusions. So unnecessary computation of irrelevant conclusions can be compounded by the need to discard irrelevant alternative complete models too.

A more intelligent solution would be afforded by selective forward-chaining, where the user would be allowed to specify those conclusions she is focused on, and only those would be computed in forward-chaining fashion. Combining backward-chaining with selective forward-chaining would allow for a greater precision in specifying what we wish to know, and improve efficiency altogether. In the sequel we show how such a selective forward chaining from a set of abductive hypotheses can be replaced by

---

backward chaining from the focused on conclusions — the inspection points — by virtue of a controlled form of abduction which, never performing extra abductions, just checks for abducibles assumed elsewhere.

### 3 Inspection Points

Meta-abduction is used in *abduction inhibited inspection*. Intuitively, when an abducible is considered under mere inspection, meta-abduction abduces only the intention to a *posteriori* check for its abduction elsewhere, i.e. it abduces the intention of verifying that the abducible is indeed adopted, but elsewhere. In practice, when we want to meta-abduce some abducible ‘*x*’, we abduce a literal ‘*consume(x)*’ (or ‘*abduced(x)*’), which represents the intention that ‘*x*’ is eventually abduced elsewhere in the process of finding an abductive solution. The check is performed after a complete abductive answer to the top query is found. Operationally, ‘*x*’ will already have been or will be later abduced as part of the ongoing solution to the top goal.

**Example 2. Police and Tear Gas Issue.** Consider this NLP, where ‘*tear\_gas*’, ‘*fire*’, and ‘*water\_cannon*’ are the only abducibles. Notice that *inspect* is applied to calls.

$\leftarrow police, riot, not\ contain.$	$\% \text{ this is an Integrity Constraint}$
$contain \leftarrow tear\_gas.$	$contain \leftarrow water\_cannon.$
$smoke \leftarrow fire.$	$smoke \leftarrow inspect(tear\_gas).$
$police.$	$riot.$

Notice the two rules for ‘*smoke*’. The first states that one explanation for smoke is fire, when assuming the hypothesis ‘*fire*’. The second states ‘*tear\_gas*’ is also a possible explanation for smoke. However, the presence of tear gas is a much more unlikely situation than the presence of fire; after all, tear gas is only used by police to contain riots and that is truly an exceptional situation. Fires are much more common and spontaneous than riots. For this reason, ‘*fire*’ is a much more plausible explanation for ‘*smoke*’ and, therefore, in order to let the explanation for ‘*smoke*’ be ‘*tear\_gas*’, there must be a plausible reason — imposed by some other likely phenomenon. This is represented by *inspect(tear\_gas)* instead of simply ‘*tear\_gas*’. The ‘*inspect*’ construct disallows regular abduction — only allowing meta-abduction — to be performed whilst trying to solve ‘*tear\_gas*’. I.e., if we take tear gas as an abductive solution for smoke, this rule imposes that the step where we abduce ‘*tear\_gas*’ is performed elsewhere, not under the derivation tree for ‘*smoke*’. Thus, ‘*tear\_gas*’ is an *inspection point*. The IC, because there is ‘*police*’ and a ‘*riot*’, forces ‘*contain*’ to be *true*, and hence, ‘*tear\_gas*’ or ‘*water\_cannon*’ or both, must be abduced. ‘*smoke*’ is only explained if, at the end of the day, ‘*tear\_gas*’ is abduced to enact containment. Abductive solutions should be plausible, and ‘*smoke*’ is plausibly explained by ‘*tear\_gas*’ if there is a reason, a best explanation, that makes the presence of tear gas plausible; in this case the riot and the police. Plausibility is an important concept in science, for lending credibility to hypotheses. Assigning plausibility measures to situations is an orthogonal issue.

---

In this example, another way of viewing the need for the new mechanism embodied by the *inspect* predicate is to consider we have 2 agents: one is a police officer and has the possibility of abducting (corresponding to actually throwing) *tear\_gas*; the other agent is a civilian who, obviously, does not have the possibility of abducting (throwing) *tear\_gas*. For the police officer agent, having the  $smoke \leftarrow inspect(tear\_gas)$  rule, with the *inspect* is unnecessary: the agent knows that *tear\_gas* is the explanation for smoke because it was himself who abducted (threw) *tear\_gas*; but for the civilian agent the *inspect* in the  $smoke \leftarrow inspect(tear\_gas)$  rule is absolutely indispensable, since he cannot abduce *tear\_gas* and therefore cannot know, without *inspecting*, if that is the real explanation for *smoke*.

**Example 3. Nuclear Power Plant Decision Problem.** This example was extracted from [12] and adapted to our current designs, and its abducibles do not represent actions. In a nuclear power plant there is decision problem: cleaning staff will dust the power plant on cleaning days, but only if there is no alarm sounding. The alarm sounds when the temperature in the main reactor rises above a certain threshold, or if the alarm itself is faulty. When the alarm sounds everybody must evacuate the power plant immediately! Abducible literals are *cleaning\_day*, *temperature\_rise* and *faulty\_alarm*.

$$\begin{aligned}
 dust & \leftarrow cleaning\_day, inspect(not\ sound\_alarm) \\
 sound\_alarm & \leftarrow temperature\_rise \\
 sound\_alarm & \leftarrow faulty\_alarm \\
 evacuate & \leftarrow sound\_alarm \\
 & \leftarrow not\ cleaning\_day
 \end{aligned}$$

Satisfying the unique IC imposes *cleaning\_day true* and gives us three minimal abductive solutions:  $S_1 = \{dust, cleaning\_day\}$ ,  $S_2 = \{cleaning\_day, sound\_alarm, temperature\_rise, evacuate\}$ , and  $S_3 = \{cleaning\_day, sound\_alarm, faulty\_alarm, evacuate\}$ . If we pose the query  $? - not\ dust$  we want to know what could justify the cleaners dusting not to occur given that it is a cleaning day (enforced by the IC). However, we do not want to abduce the rise in temperature of the reactor nor to abduce the alarm to be faulty in order to prove *not dust*. Any of these justifying two abductions must result as a side-effect of the need to explain something else, for instance the observation of the sounding of the alarm, expressible by adding the IC  $\leftarrow not\ sound\_alarm$ , which would then abduce one or both of those two abducibles as plausible explanations. The *inspect/1* in the body of the rule for *dust* prevents any abduction below *sound\_alarm* to be made just to make *not dust* true. One other possibility would be for two observations, coded by ICs  $\leftarrow not\ temperature\_rise$  or  $\leftarrow not\ faulty\_alarm$ , to be present in order for *not dust* to be true as a side-effect. A similar argument can be made about evacuating: one thing is to explain why evacuation takes place, another altogether is to justify it as necessary side-effect of root explanations for the alarm to go off. These two pragmatic uses correspond to different queries:  $? - evacuate$  and  $? - inspect(evacuate)$ , respectively.

---

### 3.1 Declarative Semantics of Inspection Points

A simple transformation maps programs with inspection points into programs without them. Mark that the Stable Models of the transformed program where each *abducible*(*X*) is matched by the abducible *X* (*X* being a literal *a* or its default negation *not a*) clearly correspond to the intended procedural meanings ascribed to the inspection points of the original program.

**Definition 3. Transforming Inspection Points.** *Let  $P$  be a program containing rules whose body possibly contains inspection points. The program  $\Pi(P)$  consists of:*

1. *all the rules obtained by the rules in  $P$  by systematically replacing:*
  - *inspect(not  $L$ ) with not inspect( $L$ );*
  - *inspect( $a$ ) or inspect(abduced( $a$ )) with abduced( $a$ ) if  $a$  is an abducible, and keeping inspect( $a$ ) otherwise.*
2. *for every rule  $A \leftarrow L_1, \dots, L_t$  in  $P$ , the additional rule:*  

$$\text{inspect}(A) \leftarrow L'_1, \dots, L'_t \text{ where for every } 1 \leq i \leq t:$$

$$L'_i = \begin{cases} \text{abduced}(L_i) & \text{if } L_i \text{ is an abducible} \\ \text{inspect}(X) & \text{if } L_i \text{ is inspect}(X) \\ \text{inspect}(L_i) & \text{otherwise} \end{cases}$$

The semantics of the *inspect* predicate is exclusively given by the generated rules for *inspect*

**Example 4. Transforming a Program  $P$  with Nested Inspection Levels.**

$$\begin{array}{ll} x \leftarrow a, \text{inspect}(y), b, c, \text{not } d & y \leftarrow \text{inspect}(\text{not } a) \\ z \leftarrow d & y \leftarrow b, \text{inspect}(\text{not } z), c \end{array}$$

Then,  $\Pi(P)$  is:

$$\begin{array}{ll} x & \leftarrow a, \text{inspect}(y), b, c, \text{not } d \\ \text{inspect}(x) & \leftarrow \text{abduced}(a), \text{inspect}(y), \text{abduced}(b), \text{abduced}(c), \text{not abduced}(d) \\ y & \leftarrow \text{not inspect}(a) \\ y & \leftarrow b, \text{not inspect}(z), c \\ \text{inspect}(y) & \leftarrow \text{not abduced}(a) \\ \text{inspect}(y) & \leftarrow \text{abduced}(b), \text{not inspect}(z), \text{abduced}(c) \\ z & \leftarrow d \\ \text{inspect}(z) & \leftarrow \text{abduced}(d) \end{array}$$

The abductive stable model of  $\Pi(P)$  respecting the inspection points is:  
 $\{x, a, b, c, \text{abduced}(a), \text{abduced}(b), \text{abduced}(c), \text{inspect}(y)\}$ .

Note that for each *abduced*(*a*) the corresponding *a* is in the model.

---

## 4 Implementation

We based our practical work on a formally defined, XSB-implemented, true and tried abduction system — ABDUAL [1]. ABDUAL lays the foundations for efficiently computing queries over ground three-valued abductive frameworks for extended logic programs with integrity constraints, on the well-founded semantics and its partial stable models.

The query processing technique in ABDUAL relies on a mixture of program transformation and tabled evaluation. A transformation removes default negative literals (by making them positive) from both the program and the integrity rules. Specifically, a dual transformation is used, that defines for each objective literal  $O$  and its set of rules  $R$ , a dual set of rules whose conclusions *not* ( $O$ ) are true if and only if  $O$  is false in  $R$ . Tabled evaluation of the resulting program turns out to be much simpler than for the original program, whenever abduction over negation is needed. At the same time, termination and complexity properties of tabled evaluation of extended programs are preserved by the transformation, when abduction is not needed. Regarding tabled evaluation, ABDUAL is in line with SLG [13] evaluation, which computes queries to normal programs according to the well-founded semantics. To it, ABDUAL tabled evaluation adds mechanisms to handle abduction and deal with the dual programs.

ABDUAL is composed of two modules: the preprocessor which transforms the original program by adding its dual rules, plus specific abduction-enabling rules; and a meta-interpreter allowing for top-down abductive query solving. When solving a query, abducibles are dealt with by means of extra rules the preprocessor added to that effect. These rules just add the name of the abducible to an ongoing list of current abductions, unless the negation of the abducible was added before to the lists failing in order to ensure abduction consistency. Meta-abduction is implemented adroitly by means of a reserved predicate, ‘*inspect/1*’ taking some literal  $L$  as argument, which engages the abduction mechanism to try and discharge any meta-abductions performed under  $L$  by matching with the corresponding abducibles, adopted elsewhere outside any ‘*inspect/1*’ call. The approach taken can easily be adopted by other abductive systems, as we had the occasion to check, e.g., with system [2]. We have also enacted an alternative implementation, relying on XSB-XASP and the declarative semantics transformation above, which is reported below.

Procedurally, in the ABDUAL implementation, the checking of an inspection point corresponds to performing a top-down query-proof for the inspected literal, but with the specific proviso of disabling new abductions during that proof. The proof for the inspected literal will succeed only if the abducibles needed for it were already adopted, or will be adopted, in the present ongoing solution search for the top query. Consequently, this check is performed after a solution for the query has been found. At inspection-point-top-down-proof-mode, whenever an abducible is encountered, instead of adopting it, we simply adopt the intention to *a posteriori* check if the abducible is part of the answer to the query (unless of course the negation of the abducible has already been adopted by then, allowing for immediate failure at that search node.) That is, one (meta-) abduces the checking of some abducible  $A$ , and the check consists in confirming that  $A$  is part of the abductive solution by matching it with the object of the check. According to our method, the side-effects of interest are explicitly indicated by the user by wrap-

---

ping the corresponding goals subject to inspection mode, with the reserved construct *'inspect/1'*.

#### 4.1 ABDUAL with Inspection Points

Inspection points in ABDUAL function mainly by means of controlling the general abduction step, which involves very few changes, both in the pre-processor and the meta-interpreter. Whenever an *'inspect(X)'* literal is found in the body of a rule, where *'X'* is a goal, a meta-abduction-specific counter — the *'inspect\_counter'* — is increased by one, in order to keep track of the allowed character, active or passive, of performed abductions. The top-down evaluation of the query for *'X'* then proceeds normally. Actual abductions are only allowed if the counter is set to zero, otherwise only meta-abductions are allowed. After finding an abductive solution for the query *'X'* the counter is decreased by one. Backtracking over counter assignments is duly accounted for. Of course, this way of implementing the inspection points (with one *'inspect\_counter'*) presupposes the abductive query answering process is carried out “depth-first”, guaranteeing the order of the literals in the bodies of rules actually corresponds to the order they are processed. We assume such a “depth-first” discipline in the implementation of inspection points, described in detail below. We lift this restriction at the end of the subsection.

##### Changes to the pre-processor:

1. A new dynamic predicate was added: the *'inspect\_counter/1'*. This is initialized to zero (*'inspect\_counter(0)'*) via an assert, before a top-level query is launched.
2. The original rules for the normal abduction step are now preceded by an additional condition checking that the *'inspect\_counter'* is indeed set to zero.
3. Extra rules for the “inspection” abduction step are added, preceded by a condition checking the *'inspect\_counter'* is set to greater than zero. When these rules are called, the corresponding abducible *'A'* is not abducted as it would happen in the original rules; instead, *'consume(A)'* (or *'abduced(A)'*) is abducted. This corresponds to the meta-abduction: we abduce the need to abduce *'A'*, the need to ‘consume’ the abduction of *'A'*, which is finally checked when derivation for the very top goal is finished.

The changes to the meta-interpreter include all the remaining processing needed to correctly implement inspection points, namely matching the meta-abduction of *'consume(X)'* against the abduction of *'X'*.

**Changes to the meta-interpreter:** The semantics we chose for the inspection points in ABDUAL is actually very close to that of the *deontic verifiers* mentioned before (and also below), in the sense that if a meta-abduction on *'X'* (resulting from abducting *'consume(X)'*) is not matched by an actual abduction on *'X'* when we reach the end of solving the top query, the candidate abductive answer is considered invalid and the query solving fails. On backtracking, another alternative abductive solution (possibly with other meta-abductions) will be sought.

In detail, the changes to the meta-interpreter include:

- 
1. Two ‘quick-kill’ rules for improved efficiency that detect and immediately solve trivial cases for meta-abduction:
    - When literal ‘ $X$ ’ about to be meta-abduced (‘ $consume(X)$ ’ about to be added to the abductions list) has actually been abduced already (‘ $X$ ’ is in the abductions list) the meta-abduction succeeds immediately and ‘ $consume(X)$ ’ is not added to the abductions list;
    - When the situation in the previous point occurs, but with ‘ $not X$ ’ already abduced instead, the meta-abduction immediately fails.
  2. Two new rules for the general case of meta-abduction, that now specifically treat the ‘ $inspect(not X)$ ’ and ‘ $inspect(X)$ ’ literals. In either rule, first we increase the ‘ $inspect\_counter$ ’ mentioned before, then proceed with the usual meta-interpretation for ‘ $not X$ ’ (‘ $X$ ’, respectively), and, when this evaluation succeeds, we then decrease ‘ $inspect\_counter$ ’.
  3. After an abductive solution is found to the top query, check (impose) that every meta-abduction, i.e., every ‘ $consume(X)$ ’ literal abduced, is matched by a respective and consistent abduction, i.e., is matched by the abducible ‘ $X$ ’ in the abductions list; otherwise the tentative solution found fails.

A counter — ‘ $inspect\_counter$ ’ — is used instead of a toggle because several ‘ $inspect(X)$ ’ literals may appear at different graph-depth levels under each other, and resetting a toggle after solving a lower-level meta-abduction would allow actual abductions under the higher-level meta-abduction. An example clarifies this.

**Example 5. Nested Inspection Points.** Consider again the program of the previous example, where the abducibles are  $a, b, c, d$ :

$$\begin{array}{ll} x \leftarrow a, inspect(y), b, c, not\ d. & y \leftarrow inspect(not\ a). \\ z \leftarrow d. & y \leftarrow b, inspect(not\ z), c. \end{array}$$

When we want to find an abductive solution for  $x$ , skipping over the low-level technical details we proceed as follows:

1.  $a$  is an abducible and since the ‘ $inspect\_counter$ ’ is still set initially to 0 we can abduce  $a$  by adding it to the running abductions list;
2.  $y$  is not an abducible and so we cannot use any ‘quick kill’ rule on it. We increase the ‘ $inspect\_counter$ ’ — which now takes the value 1 — and proceed to find an abductive solution for  $y$ ;
3. since the ‘ $inspect\_counter$ ’ is different from 0, only meta-abductions are allowed;
4. using the first rule for  $y$  we need to ‘ $inspect(not\ a)$ ’, but since we have already abduced  $a$  a ‘quick-kill’ is applicable here: we already know that this ‘ $inspect(not\ a)$ ’ will fail. The value of the ‘ $inspect\_counter$ ’ will remain 1;
5. on backtracking, the second rule for  $y$  is selected, and now we meta-abduce  $b$  by adding ‘ $consume(b)$ ’ to the ongoing abductions list;
6. increase ‘ $inspect\_counter$ ’ again, making it take the value 2, and continue on, searching an abductive solution for  $not\ z$ ;
7. the only solution for  $not\ z$  is by abducting  $not\ d$ , but since the ‘ $inspect\_counter$ ’ is greater than 0, we can only meta-abduce  $not\ d$ , i.e., ‘ $consume(not\ d)$ ’ is added to the running abductions list;



- 
8. returning to  $y$ 's rule: the meta-interpretation of ' $inspect(not\ z)$ ' succeeds and so we decrease the ' $inspect\_counter$ ' by one — it takes the value 1 again. Now we proceed and try to solve  $c$ ;
  9.  $c$  is an abducible, but since the  $inspect\_counter$  is set to 1, we only meta-abduce  $c$  by adding ' $consume(c)$ ' to the running abductions list;
  10. returning to  $x$ 's rule: the meta-interpretation of ' $inspect(y)$ ' succeeds and so we decrease the ' $inspect\_counter$ ' once more, and it now takes the value 0. From this point onwards regular abductions will take place instead of meta-abductions;
  11. we abduce  $b$ ,  $c$ , and  $not\ d$  by adding them to the abductions list;
  12. a tentative abductive solution is found to the initial query. It consists of the abductions:  $[a, consume(b), consume(not\ d), consume(c), b, c, not\ d]$ ;
  13. the abductive solution is now checked for matches between meta-abductions and actual abductions. In this case, for every ' $consume(A)$ ' in the abduction list there is an  $A$  also in the abduction list, i.e., every intention of abduction ' $consume(A)$ ' is satisfied by the actual abduction of  $A$ . Because this final checking step succeeds, the whole answer is actually accepted. Note it is irrelevant which order a ' $consume(A)$ ' and the corresponding  $A$  appear and were placed in the abductions list. The  $A$  in  $consume(A)$  is just any abducible literal  $a$  or its default negation  $not\ a$ .

In this example, we can see clearly that the *inspect* predicate can be used on any arbitrary literal, and not just on abducibles.

The correctness of this implementation against the declarative semantics provided before can be sketched by noticing that whenever the *inspect\_counter* is set to 0 the meta-interpreter performs actual abduction which corresponds to the use of the original program rules; whenever the *inspect\_counter* is set to some value greater than 0 the meta-interpreter just abduces  $consume(A)$  (where  $A$  is the abducible being checked for its abduction being produced elsewhere), and this corresponds to the use of the program transformation rules for the *inspect* predicate.

The implementation of ABDUAL with inspection points is available on request.

**More general query solving** In case the “depth-first” discipline is not followed, either because goal delaying is taking place, or multi-threading, or co-routining, or any other form of parallelism is being exploited, then each queried literal will need to carry its own list of ancestors with their individual '*inspect\_counters*'. This is necessary so as to have a means, in each literal, to know which and how many *inspects* there are between the root node and the currently being processed literal, and which *inspect\_counter* to update; otherwise there would be no way to know if abductions or meta-abductions should be performed.

## 4.2 Alternative Implementation Method

The method presented here is an avenue for implementing the inspection points mechanism through a simple syntactic transformation which can be readily used by any SMs system like SModels or DLV. Using an SMs implementation alone, one can get the abductive SMs of some program  $P$  by computing the SMs of  $P'$  where  $P'$  is obtained

---

from  $P$  by applying the program transformation we presented for the declarative semantics of the inspection points, and then adding an even loop over negation for each abducible (like the one depicted in section 2.1). Using XSB-Prolog's XSB-XASP interface, the process would be the same as for using an SMs implementation alone, but instead of sending the whole  $P'$  to the SMs engine, only the residual program, relevant for the query at hand, would be sent. This way, abductive reasoning can benefit from the relevance property enjoyed by the Well-Founded Semantics implemented by the XSB-Prolog's SLG-WAM.

Given the top-down proof procedure for abduction, implementing inspection points for program  $P$  becomes just a matter of adapting the evaluation of derivation subtrees falling under '*inspect/1*' literals, at meta-interpreter level, subsequent to performing the transformation  $\Pi(P)$  presented before, which defines the declarative semantics. Basically, any considered abducibles evaluated under '*inspect/1*' subtrees, say  $A$ , are codified as '*abduced(A)*', where:

$$\begin{aligned} \text{abduced}(A) &\leftarrow \text{not abduced\_not}(A) \\ \text{abduced\_not}(A) &\leftarrow \text{not abduced}(A) \end{aligned}$$

All *abduced/1* literals collected during computation of the residual program are later checked against the stable models themselves. Every '*abduced(a)*' must pair with a corresponding abducible  $a$  for the model to be accepted.

## 5 Conclusions, Comparisons, and Future Work

In the context of abductive logic programs, we have presented a new mechanism of inspecting literals that can be used to check for side-effects, by relying on meta-abduction. We have implemented the inspection mechanism within the Abdual [1] meta-interpreter, and also in XSB-XASP, and checked that it can be ported to other systems [2].

HyProlog [2] is an abduction/assumption system which allows for the user to specify if an abducible is to be consumed only once or many times. In HyProlog, as the query solving proceeds, when abducibles/assumptions consumptions take place they are executed as storing the respective consumption intention in a store. After an abductive solution for a query is found, the actual abductions/assumptions are matched against the consumption intentions. In general, there is not such a big difference between the operational semantics of HyProlog and the inspection points implementation we present; however, there is a major functionality difference: in HyProlog we can only require consumption directly on abducibles, and with inspection points we can inspect any literal, not just abducibles.

In [12], the authors detect a problem with the IFF abductive proof procedure [6] of Fung and Kowalski, in what concerns the treatment of negated abducibles in integrity constraints (e.g. in their examples 2 and 3). They then specialize IFF to avoid such problems and prove correctness of the new procedure. The problems detected refers to the active use of an IC of some not  $A$ , where  $A$  is an abducible, whereas the intended use should be a passive one, simply checking whether  $A$  is proved in the abductive solution found. To that effect they replace such occurrences of not  $A$  by not provable( $A$ ), in order to ensure that no new abductions are allowed during the checking. Our own work

---

generalizes the scope of the problem they solved and solves the problems involved in this wider scope. For one we allow for passive checking not just of negated abducibles but also of positive ones, as well as passive checking of any literal, whether or not abducible, and allow also to single out which occurrences are passive or active. Thus, we can cater for both passive and active ICs, depending on the use desired. Our solution uses abduction itself to solve the problem, making it general for use in other abductive frameworks and procedures.

A future application of inspection points is planning in a multi-agent setting. An agent may have abducted a plan and, in the course of carrying out its abducted actions, it may find that another agent has undone some of its already executed actions. So, before executing an action, the agent should check all necessary preconditions hold. Note it should only *check*, thereby avoiding abducting again a plan for them: this way, if the preconditions hold the agent can continue and execute the planned action. The agent should only take measures to enforce the preconditions again whenever the check fails. Clearly, an *inspection* of the preconditions is what we need here.

## 6 Acknowledgements

We thank Robert Kowalski, Verónica Dahl and Henning Christiansen for discussions, Pierangelo Dell’Acqua for the declarative semantics, and Gonalo Lopes for help with the XSB-XASP implementation.

## References

1. J. J. Alferes, L. M. Pereira, and T. Swift. Abduction in well-founded semantics and generalized stable models via tabled dual programs. *Theory and Practice of Logic Programming*, 4(4):383–428, July 2004.
2. H. Christiansen and V. Dahl. Hyprolog: A new logic programming language with assumptions and abduction. In M. Gabbrielli and G. Gupta, editors, *ICLP*, volume 3668 of *LNCS*, pages 159–173. Springer, 2005.
3. S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The dl<sub>v</sub> system: Model generator and advanced frontends (system description). In *12th Workshop on Logic Programming*, 1997.
4. M. Denecker and D. De Schreye. Sldnfa: An abductive procedure for normal abductive programs. In Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 686–700, Washington, USA, 1992. The MIT Press.
5. T. Eiter, G. Gottlob, and N. Leone. Abduction from logic programs: semantics and complexity. *Theoretical Computer Science*, 189(1–2):129–177, 1997.
6. T. H. Fung and R. Kowalski. The iff proof procedure for abductive logic programming. *J. Log. Prog.*, 33(2):151 – 165, 1997.
7. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. of ACM*, 38(3):620–650, 1991.
8. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.
9. K. Inoue and C. Sakama. A fixpoint characterization of abductive logic programs. *Journal of Logic Programming*, 27(2):107–136, 1996.

- 
10. A. Kakas, R. Kowalski, and F. Toni. The role of abduction in logic programming. In *Handbook of Logic in AI and LP*, volume 5, pages 235–324. Oxford University Press, 1998.
  11. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Procs. 4th Intl. Conf. Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 420–429, July 1997.
  12. F. Sadri and F. Toni. Abduction with negation as failure for active and reactive rules. In E. Lamma and P. Mello, editors, *AI\*IA*, volume 1792 of *LNCS*, pages 49–60. Springer, 1999.
  13. T. Swift and D. S. Warren. An abstract machine for slg resolution: Definite programs. In *Symp. on Logic Programming*, pages 633–652, 1994.

---

# Stable Model implementation of Layer Supported Models by program transformation

Luís Moniz Pereira and Alexandre Miguel Pinto  
{lmp|amp}@di.fct.unl.pt

Centro de Inteligência Artificial (CENTRIA)  
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal

**Abstract.** For practical applications, the use of top-down query-driven proof-procedures is convenient for an efficient use and computation of answers using Logic Programs as knowledge bases. A 2-valued semantics for Normal Logic Programs (NLPs) allowing for top-down query-solving is thus highly desirable, but the Stable Models semantics (SM) does not allow it, for lack of the relevance property. To overcome this limitation we introduced in [11], and summarize here, a new 2-valued semantics for NLPs — the Layer Supported Models semantics — which conservatively extends the SM semantics, enjoys relevance and cumulativity, guarantees model existence, and respects the Well-Founded Model. In this paper we exhibit a space and time linearly complex transformation, TR, from one propositional NLP into another, whose Layer Supported Models are precisely the Stable Models of the transform, which can then be computed by extant Stable Model implementations, providing a tool for the immediate generalized use of the new semantics and its applications. TR can be used to answer queries but is also of theoretical interest, for it may be used to prove properties of programs, say. Moreover, TR can be employed in combination with the top-down query procedure of XSB-Prolog, and be applied just to the residual program corresponding to a query (in compliance with the relevance property of Layer Supported Model). The XSB-XASP interface then allows the program transform to be sent for Smodels for 2-valued evaluation.

**Keywords:** Stable Models, Layer Supported Models, Relevance, Layering, Program Transformation.

## 1 Introduction and Motivation

The semantics of Stable Models (SM) [7] is a cornerstone for the definition of some of the most important results in logic programming of the past two decades, providing an increase in logic programming declarativity and a new paradigm for program evaluation. When we need to know the 2-valued truth value of all the literals in a logic program for the problem we are modeling and solving, the only solution is to produce complete models. Depending on the intended semantics, in such cases, tools like *SModels* [9] or *DLV* [2] may be adequate because they can indeed compute whole models according to the SM semantics. However, the lack of some important properties of language semantics, like relevance, cumulativity and guarantee of model existence (enjoyed by, say, Well-Founded Semantics [6] (WFS)), somewhat reduces its applicability in practice,

---

namely regarding abduction, creating difficulties in required pre- and post-processing. But WFS in turn does not produce 2-valued models, though these are often desired, nor guarantees 2-valued model existence.

SM semantics does not allow for top-down query-solving precisely because it does not enjoy the relevance property — and moreover, does not guarantee the existence of a model. Furthermore, frequently there is no need to compute whole models, like its implementations do, but just the partial models that sustain the answer to a query. Relevance would ensure these could be extended to whole models.

To overcome these limitations we developed in [11] (summarized here) a new 2-valued semantics for NLPs — the Layer Supported Models (LSM) — which conservatively extends the SMs, enjoys relevance and cumulativity, guarantees model existence, and respects the Well-Founded Model (WFM) [6]. Intuitively, a program is conceptually partitioned into “layers” which are subsets of its rules, possibly involved in mutual loops. An atom is considered *true* if there is some rule for it at some layer, where all the literals in its body which are supported by rules of lower layers are also *true*. Otherwise that conclusion is false. That is, a rule in a layer must, to be usable, have the support of rules in the layers below.

The core reason SM semantics fails to guarantee model existence for every NLP is that the stability condition it imposes on models is impossible to be complied with by Odd Loops Over Negation (OLONs)<sup>1</sup>. In fact, the SM semantics community uses such inability as a means to impose Integrity Constraints (ICs).

**Example 1. OLON as IC.** Indeed, using SMs, one would write an IC in order to prevent  $X$  being in any model with the single rule for some atom ‘ $a$ ’:  $a \leftarrow \text{not } a, X$ . Since the SM semantics cannot provide a semantics to this rule whenever  $X$  holds, this type of OLON is used as IC. When writing such ICs under SMs one must be careful and make sure there are no other rules for  $a$ . But the really unintuitive thing about this kind of IC used under SM semantics is the meaning of the atom  $a$ . What does  $a$  represent?

The LSM semantics instead provides a semantics to all NLPs. ICs are implemented with rules for reserved atom *falsum*, of the form  $\text{falsum} \leftarrow X$ , where  $X$  is the body of the IC we wish to prevent being true. This does not prevent *falsum* from being in some models. To avoid them, the user must either conjoin goals with *not falsum* or, if inconsistency examination is desired, *a posteriori* discard such models. LSM semantics separates OLON semantics from IC compliance.

After notation and background definitions, we summarize the formal definition of LSM semantics and its properties. Thereafter, we present a program transformation, TR, from one propositional (or ground) NLP into another, whose Layer Supported Models are precisely the Stable Models of the transform, which can be computed by extant Stable Model implementations, which also require grounding of programs. TR’s linear space and time complexities are then examined. The transformation can be used to answer queries but is also of theoretical interest, for it may be used to prove properties of programs, say. In the Implementation section we show how TR can be employed, in combination with the top-down query procedure of XSB-Prolog, it being sufficient to apply it solely to the residual program corresponding to a query (in compliance with

---

<sup>1</sup> OLON is a loop with an odd number of default negations in its circular call dependency path.

---

the relevance property of Layer Supported Model ). The XSB-XASP interface allows the program transform to be sent for Smodels for 2-valued evaluation. Conclusions and future work close the paper.

## 2 Background Notation and Definitions

**Definition 1. Logic Rule.** A Logic Rule  $r$  has the general form

$H \leftarrow B_1, \dots, B_n, \text{not } C_1, \dots, \text{not } C_m$  where  $H$ , the  $B_i$  and the  $C_j$  are atoms.

We call  $H$  the head of the rule — also denoted by  $\text{head}(r)$ . And  $\text{body}(r)$  denotes the set  $\{B_1, \dots, B_n, \text{not } C_1, \dots, \text{not } C_m\}$  of all the literals in the body of  $r$ . Throughout this paper we will use ‘not’ to denote default negation. When the body of the rule is empty, we say the head of rule is a fact and we write the rule just as  $H$ . An IC or denial is a rule with head  $\text{falsum}$ .

**Definition 2. Logic Program.** A Logic Program (LP for short)  $P$  is a (possibly infinite) set of ground Logic Rules of the form in Definition 1.

In this paper we focus only on NLPs, those whose heads of rules are positive literals, i.e., simple atoms; and there is default negation just in the bodies of the rules. Hence, when we write simply “program” or “logic program” we mean an NLP. The shifting rule [5, 8] may be used to reduce disjunctive programs into NLPs, as may other known transformations, say from Extended LPs into NLPs ([3]).

## 3 Layering of Logic Programs

The well-known notion of stratification of LPs has been studied and used for decades now. But the common notion of stratification does not cover all LPs, i.e., there are some LPs which are non-stratified. The usual syntactic notions of dependency are mainly focused on atoms. They are based on a dependency graph induced by the rules of the program. Useful as these notions might be, for our purposes they are insufficient since they leave out important structural information about the call-graph of  $P$ . To encompass that information we define below the notion of a rule’s dependency. Indeed, layering puts rules, not atoms, in layers. An atom  $B$  directly depends on atom  $A$  in  $P$  iff there is at least one rule with head  $B$  and with  $A$  or  $\text{not } A$  in the body. An atom’s dependency is just the transitive closure of the atom’s direct dependency. A rule directly depends on atom  $B$  iff any of  $B$  or  $\text{not } B$  is in its body. A rule’s dependency is just the transitive closure of the rule’s direct dependency. The *relevant* part of  $P$  for some atom  $A$ , represented by  $\text{Rel}_P(A)$ , is the subset of rules of  $P$  with head  $A$  plus the set of rules of  $P$  whose heads the atom  $A$  depends on, cf. [4]. The relevant part of  $P$  for rule  $r$ , represented by  $\text{Rel}_P(r)$ , is the set containing the rule  $r$  itself plus the set of rules relevant for each atom  $r$  depends on.

**Definition 3. Parts of the body of a rule.** Let  $r = H \leftarrow B_1, \dots, B_n, \text{not } C_1, \dots, \text{not } C_m$  be a rule of  $P$ . Then,  $r^l = \{B_i, \text{not } C_j : B_i \text{ depends on } H \wedge C_j \text{ depends on } H\}$ . Also,  $r^B = \{B_i : B_i \in (\text{body}(r) \setminus r^l)\}$ , and  $r^C = \{C_j : \text{not } C_j \in (\text{body}(r) \setminus r^l)\}$ .

---

**Definition 4. HighLayer function.** The *HighLayer* function is defined over a set of literals: its result is the highest layer number of all the rules for all the literals in the set, or zero if the set is empty. The highest layer number of a given literal is zero if there are no rules for it in the program, otherwise it is the maximum of the layer numbers for all the rules having that literal as head.

**Definition 5. Layering of a Logic Program  $P$ .** Given a logic program  $P$  a layering function  $L/1$  is just any function defined over the rules of  $P'$ , where  $P'$  is obtained from  $P$  by adding a rule of the form  $H \leftarrow \text{falsum}$  for every atom  $H$  with no rules in  $P$ , assigning each rule  $r \in P'$  a positive integer, such that:

- $L(r) = 0$  if  $\text{falsum} \in \text{body}(r)$ , otherwise
- $L(r) \geq \max(\text{HighLayer}(r^l), \text{HighLayer}(r^B), (\text{HighLayer}(r^C) + 1))$

A layering of program  $P$  is a partition  $P^1, \dots, P^n$  of  $P$  such that  $P^i$  contains all rules  $r$  having  $L(r) = i$ , i.e., those which depend only on the rules in the same layer or layers below it.

Amongst the several possible layerings of a program  $P$  we can always find the least one, i.e., the layering with least number of layers and where the integers of the layers are smallest. In the remainder of the paper when referring to the program's layering we mean such least layering (easily seen to be unique).

**Example 2. Layering example.** Consider the following program  $P$ , depicted along with the layer numbers for its least layering:

$c \leftarrow \text{not } d, \text{not } y, \text{not } a$		Layer 3
$d \leftarrow \text{not } c$		
$y \leftarrow \text{not } x$	$b \leftarrow \text{not } x$	Layer 2
$x \leftarrow \text{not } x$	$b$	Layer 1
$a \leftarrow \text{falsum}$		Layer 0

Atom  $a$  has no rules so its now created unique rule  $a \leftarrow \text{falsum}$  is placed in Layer 0. Atom  $b$  has a fact rule  $r_{b_1}$ : its body is empty, and therefore all  $\text{HighLayer}(r_{b_1}^l)$ ,  $\text{HighLayer}(r_{b_1}^B)$ , and  $\text{HighLayer}(r_{b_1}^C)$  are 0 (zero). Hence,  $L(r_{b_1}) = \max(0, 0, (0 + 1)) = \max(0, 0, 1) = 1$ , where  $r_{b_1}$  is the fact rule for  $b$ , placed in Layer 1.

The unique rule for  $x$ ,  $r_x$  is also placed in Layer 1 in the least layering of  $P$  because  $\text{HighLayer}(r_x^l) = L(r_x)$ ,  $\text{HighLayer}(r_x^B) = 0$ , and  $\text{HighLayer}(r_x^C) = 0$ . So,  $L(r_x) = \max(L(r_x), 0, (0 + 1)) = \max(L(r_x), 0, 1) = 1$ , in the least layering.

The unique rule for  $c$ ,  $r_c$  is placed in Layer 3 because  $\text{HighLayer}(r_c^C) = 2$ ,  $\text{HighLayer}(r_c^B) = 0$ , and  $\text{HighLayer}(r_c^l) = \text{HighLayer}(r_d) = 3$ . By the same token,  $r_d$  is placed in the same Layer 3. Both  $r_{b_2}$  and  $r_y$  are placed in Layer 2.

This program has two LSMs:  $\{b, c, x\}$ , and  $\{b, d, x\}$ .

## 4 Layer Supported Models Semantics

The Layer Supported Models semantics we now present is the result of the two new notions we introduced: the layering, formally introduced in section 3, which is a generalization of stratification; and the layered support, as a generalization of classical



---

support. These two notions are the means to provide the desired 2-valued semantics which respects the WFM, as per below.

An interpretation  $M$  of  $P$  is classically supported iff every atom  $a$  of  $M$  is classically supported in  $M$ , i.e., *all* the literals in the body of some rule for  $a$  are *true* under  $M$  in order for  $a$  to be supported under  $M$ .

**Definition 6. Layer Supported interpretation.** *An interpretation  $M$  of  $P$  is layer supported iff every atom  $a$  of  $M$  is layer supported in  $M$ , and this holds iff  $a$  has a rule  $r$  where all literals in  $(\text{body}(r) \setminus r^l)$  are true in  $M$ . Otherwise, it follows that  $a$  is false.*

**Theorem 1. Classical Support implies Layered Support.** *Given an NLP  $P$ , an interpretation  $M$ , and an atom  $a$  such that  $a \in M$ , if  $a$  is classically supported in  $M$  then  $a$  is also layer supported in  $M$ .*

*Proof.* Trivial from the definitions of classical support and layered support.  $\square$

In programs without odd loops layered supported models are classically supported too.

Intuitively, the minimal layer supported models up to and including a given layer, respect the minimal layer supported models up to the layers preceding it. It follows trivially that layer supported models are minimal models, by definition. This ensures the truth assignment to atoms in loops in higher layers is consistent with the truth assignments in loops in lower layers and that these take precedence in their truth labeling. As a consequence of the layered support requirement, layer supported models of each layer comply with the WFM of the layers equal to or below it. Combination of the (merely syntactic) notion of layering and the (semantic) notion of layered support makes the LSM semantics.

**Definition 7. Layer Supported Model of  $P$ .** *Let  $P^1, \dots, P^n$  be the least layering of  $P$ . A layer supported interpretation  $M$  is a Layer Supported Model of  $P$  iff*

$$\forall 1 \leq i \leq n \ M|_{\leq i} \text{ is a minimal layer supported model of } \cup_{1 \leq j \leq i} P^j$$

where  $M|_{\leq i}$  denotes the restriction of  $M$  to heads of rules in layers less or equal to  $i$ :

$$M|_{\leq i} \subseteq M \cap \{\text{head}(r) : L(r) \leq i\}$$

*The Layer Supported semantics of a program is just the intersection of all of its Layer Supported Models.*

**Example 3. Layer Supported Models semantics.** Consider again the program from example 2. Its LS models are  $\{b, c, x\}$ , and  $\{b, d, x\}$ . According to LSM semantics  $b$  and  $x$  are *true* because they are in the intersection of all models.  $c$  and  $d$  are *undefined*, and  $a$  and  $y$  are *false*.

Layered support is a more general notion than that of perfect models [12], with similar structure. Perfect model semantics talks about “least models” rather than “minimal models” because in strata there can be no loops and so there is always a unique least model which is also the minimal one. Layers, as opposed to strata, may contain loops and thus there is not always a least model, so layers resort to minimal models, and these are guaranteed to exist (it is well known every NLP has minimal models).

The arguments in favor of the LSM semantics are presented in [10, 11], and are not detailed here. This paper assumes the LSM semantics and focuses on a program transformation.

---

#### 4.1 Respect for the Well-Founded Model

**Definition 8.** *Interpretation  $M$  of  $P$  respects the WFM of  $P$ .* An interpretation  $M$  respects the WFM of  $P$  iff  $M$  contains the set of all the true atoms of WFM, and it is contained by the set of true or undefined atoms of the WFM. Formally,  $WFM^+(P) \subseteq M \subseteq WFM^{+u}(P)$ .

**Theorem 2.** *Layer Supported Models respect the WFM.* Let  $P$  be a NLP, and  $P^{\leq i}$  denote  $\bigcup_{1 \leq j \leq i} P^j$ . Each sub-LSM  $M|_{\leq i}$  of LSM  $M$  respects the WFM of  $P^{\leq i}$ .

*Proof.* (The reader can skip this proof without loss for the following). By definition, each  $M|_{\leq i}$  is a full LSM of  $P^{\leq i}$ . Consider  $P^{\leq 1}$ . Every  $M|_{\leq 1}$  contains the facts of  $P$ , and their direct positive consequences, since the rules for all of these are necessarily placed in the first layer in the least layering of  $P$ . Necessarily,  $M|_{\leq 1}$  contains all the *true* atoms of the WFM of  $P^{\leq 1}$ . Layer 1 also contains whichever loops that do not depend on any other atoms besides those which are the heads of the rules forming the loop. These loops that have no negative literals in the bodies are deterministic and, therefore, the heads of the rules forming the loop will be all *true* or all *false* in the WFM, depending if the bodies are fully supported by facts in the same layer, or not. In any case, a minimal model of this layer will necessarily contains all the *true* atoms of the WFM of  $P^{\leq 1}$ , i.e.,  $WFM^+(P^{\leq 1})$ . For loops involving default negation, the atoms head of rules forming such loops are *undefined* in the WFM; some of them might be in  $M|_{\leq i}$  too. Assume now there is some atom *a false* in the WFM of  $P^{\leq 1}$  such that  $a \in M|_{\leq 1}$ . *a* can only be *false* in the WFM of  $P^{\leq 1}$  if either it has no rules or if every rule for *a* has a false body. In the first case *a*'s unique rule  $a \leftarrow false$  is placed in the first layer making it impossible for *a* to be in any LSM. In the second case, since in a LSM every atom must be layer supported, if all the bodies of all rules for *a* are false, *a* will not be layer supported and so it will not be in any LSM, in particular, not in  $M|_{\leq 1}$ . Since  $M|_{\leq 1}$  contains all *true* atoms of WFM of  $P^{\leq 1}$  and it contains no *false* atoms, it must be contained by the *true* or *undefined* atoms of WFM of  $P^{\leq 1}$ . Consider now  $P^{\leq i+1}$ , and  $M|_{\leq i}$  a LSM of  $P^{\leq i}$ . Assuming in  $P^{i+1}$  all the atoms of  $M|_{\leq i}$  as *true*, there might be some bodies of rules of  $P^{i+1}$  which are true. In such case, a minimal model of  $P^{i+1}$  will also consider the heads of such rules to be *true* — these will necessarily comply with the layered support requirement, and will be *true* in the WFM of  $P^{i+1} \cup M|_{\leq i}$ . For the same reasons indicated for layer 1, no *false* atom in the WFM of  $P^{i+1} \cup M|_{\leq i}$  could ever be considered *true* in  $M|_{\leq i+1}$ .  $\square$

## 5 Program Transformation

The program transformation we now define provides a syntactical means of generating a program  $P'$  from an original program  $P$ , such that the SMs of  $P'$  coincide with the LSMs of  $P$ . It engenders an expedite means of computing LSMs using currently available tools like Smodels [9] and DLV [2]. The transformation can be query driven and performed on the fly, or previously preprocessed.

## 5.1 Top-down transformation

Performing the program transformation in top-down fashion assumes applying the transformation to each atom in the program in the call-graph of a query. The transformation involves traversing the call-graph for the atom, induced by its dependency rules, to detect and “solve” the OLONs, via the specific LSM-enforcing method described below. When traversing the call-graph for an atom, one given traverse branch may end by finding (1) a fact literal, or (2) a literal with no rules, or (3) a loop to a literal (or its default negation conjugate) already found earlier along that branch.

To produce  $P'$  from  $P$  we need a means to detect OLONs. The OLON detection mechanism we employ is a variant of Tarjan’s Strongly Connected Component (SCC) detection algorithm [14], because OLONs are just SCCs which happen to have an odd number of default negations along its edges. Moreover, when an OLON is detected, we need another mechanism to change its rules, that is to produce and add new rules to the program, which make sure the atoms  $a$  in the OLON now have “stable” rules which do not depend on any OLON. We say such mechanism is an “OLON-solving” one. Trivial OLONs, i.e. with length 1 like that in Example 1 ( $a \leftarrow \text{not } a, X$ ), are “solved” simply by removing the  $\text{not } a$  from the body of the rule. General OLONs, i.e. with length  $\geq 3$ , have more complex (non-deterministic) solutions, described below.

**Minimal Models of OLONs** In general, an OLON has the form

$$\begin{aligned} R_1 &= \lambda_1 \leftarrow \text{not } \lambda_2, \Delta_1 \\ R_2 &= \lambda_2 \leftarrow \text{not } \lambda_3, \Delta_2 \\ &\vdots \\ R_n &= \lambda_n \leftarrow \text{not } \lambda_1, \Delta_n \end{aligned}$$

where all the  $\lambda_i$  are atoms, and the  $\Delta_j$  are arbitrary conjunction of literals which we refer to as “contexts”. Assuming any  $\lambda_i$  *true* alone in some model suffices to satisfy any two rules of the OLON: one by rendering the head *true* and the other by rendering the body *false*.

$$\begin{aligned} \lambda_{i-1} &\leftarrow \sim \lambda_i, \Delta_{i-1}, \text{ and} \\ \lambda_i &\leftarrow \sim \lambda_{i+1}, \Delta_i \end{aligned}$$

A minimal set of such  $\lambda_i$  is what is needed to have a minimal model for the OLON. Since the number of rules  $n$  in OLON is odd we know that  $\frac{n-1}{2}$  atoms satisfy  $n-1$  rules of OLON. So,  $\frac{n-1}{2} + 1 = \frac{n+1}{2}$  atoms satisfy all  $n$  rules of OLON, and that is the minimal number of  $\lambda_i$  atoms which are necessary to satisfy all the OLON’s rules. This means that the remaining  $n - \frac{n+1}{2} = \frac{n-1}{2}$  atoms  $\lambda_i$  must be *false* in the model in order for it to be minimal.

Taking a closer look at the OLON rules we see that  $\lambda_2$  satisfies both the first and second rules; also  $\lambda_4$  satisfies the third and fourth rules, and so on. So the set  $\{\lambda_2, \lambda_4, \lambda_6, \dots, \lambda_{n-1}\}$  satisfies all rules in OLON except the last one. Adding  $\lambda_1$  to this set, since  $\lambda_1$  satisfies the last rule, we get one possible minimal model for OLON:  $M_{OLON} = \{\lambda_1, \lambda_2, \lambda_4, \lambda_6, \dots, \lambda_{n-1}\}$ . Every atom in  $M_{OLON}$  satisfies 2 rules of OLON alone, except  $\lambda_1$ , the last atom added.  $\lambda_1$  satisfies alone only the last rule of OLON. The first rule of OLON —  $\lambda_1 \leftarrow \text{not } \lambda_2, \Delta_1$  — despite being satisfied by  $\lambda_1$ , was already satisfied by  $\lambda_2$ . In this case, we call  $\lambda_1$  the *top literal* of the OLON under

$M$ . The other Minimal Models of the OLON can be found in this manner simply by starting with  $\lambda_3$ , or  $\lambda_4$ , or any other  $\lambda_i$  as we did here with  $\lambda_2$  as an example. Consider the  $M_{OLON} = \{\lambda_1, \lambda_2, \lambda_4, \lambda_6, \dots, \lambda_{n-1}\}$ . Since  $\sim \lambda_{i+1} \in \text{body}(R_i)$  for every  $i < n$ , and  $\sim \lambda_1 \in \text{body}(R_n)$ ; under  $M_{OLON}$  all the  $R_1, R_3, R_5, \dots, R_n$  will have their bodies false. Likewise, all the  $R_2, R_4, R_6, \dots, R_{n-1}$  will have their bodies true under  $M_{OLON}$ .

This means that all  $\lambda_2, \lambda_4, \lambda_6, \dots, \lambda_{n-1}$  will have classically supported bodies (all body literals true), namely via rules  $R_2, R_4, R_6, \dots, R_{n-1}$ , but not  $\lambda_1$  — which has only layered support (all body literals of strictly lower layers true). “Solving an OLON” corresponds to adding a new rule which provides classical support for  $\lambda_1$ . Since this new rule must preserve the semantics of the rest of  $P$ , its body will contain only the conjunction of all the “contexts”  $\Delta_j$ , plus the negation of the remaining  $\lambda_3, \lambda_5, \lambda_7, \dots, \lambda_n$  which were already considered *false* in the minimal model at hand.

These mechanisms can be seen at work in lines 2.10, 2.15, and 2.16 of the Transform Literal algorithm below.

**Definition 9. Top-down program transformation.**

<pre> <b>input</b> : A program P <b>output</b>: A transformed program P'  1.1 context = <math>\emptyset</math> 1.2 stack = empty stack 1.3 P' = P 1.4 <b>foreach</b> atom <math>a</math> of P <b>do</b> 1.5     Push (<math>a</math>, stack) 1.6     P' = P' <math>\cup</math> Transform Literal (<math>a</math>) 1.7     Pop (<math>a</math>, stack) 1.8 <b>end</b> </pre>
---

**Algorithm 1:** TR Program Transformation

The TR transformation consists in performing this literal transformation, for each individual atom of  $P$ . The Transform Literal algorithm implements a top-down, rule-directed, call-graph traversal variation of Tarjan’s SCC detection mechanism. Moreover, when it encounters an OLON (line 2.9 of the algorithm), it creates (lines 2.13–2.17) and adds (line 2.18) a new rule for each literal involved in the OLON (line 2.11). The newly created and added rule renders its head true only when the original OLON’s context is true, but also only when that head is not classically supported, though being layered supported under the minimal model of the OLON it is part of.

**Example 4. Solving OLONs.** Consider this program, coinciding with its residual:

$$a \leftarrow \text{not } a, b \quad b \leftarrow c \quad c \leftarrow \text{not } b, \text{not } a$$

Solving a query for  $a$ , we use its rule and immediately detect the OLON on  $a$ . The leaf *not a* is removed; the rest of the body  $\{b\}$  is kept as the Context under which the OLON on  $a$  is “active” — if  $b$  were to be false there would be no need to solve the

---

```

input : A literal  $l$ 
output: A partial transformed program  $Pa'$ 

2.1 previous context = context
2.2  $Pa' = P$ 
2.3 atom = atom  $a$  of literal  $l$ ; //removing the eventual not
2.4 if  $a$  has been visited then
2.5     if  $a$  or not  $a$  is in the stack then
2.6         scc root indx = lowest stack index where  $a$  or not  $a$  can be found
2.7         nots seq = sequence of neg. lits from (scc root indx + 1) to top indx
2.8         loop length = length of nots seq
2.9         if loop length is odd then
2.10             # nots in body =  $\frac{(\text{loop length} - 1)}{2}$ 
2.11             foreach 'not  $x$ ' in nots seq do
2.12                 idx = index of not  $x$  in nots seq
2.13                 newbody = context
2.14                 for  $i=1$  to # nots in body do
2.15                     newbody = newbody  $\cup$ 
2.16                         { nots seq  $((idx + 2 * i) \bmod \text{loop length})$  }
2.17                 end
2.18                 newrule =  $x \leftarrow \text{newbody}$ 
2.19                  $Pa' = Pa' \cup \{\text{newrule}\}$ 
2.20             end
2.21         end
2.22     end
2.23 else //  $a$  has not been visited yet
2.24     mark  $a$  as visited
2.25     foreach rule  $r = a \leftarrow b_1, \dots, b_n, \text{not } b_{n+1}, \dots, \text{not } b_m$  of  $P$  do
2.26         foreach (not)  $b_i$  do
2.27             Push ((not)  $b_i$ , stack)
2.28             context = context  $\cup \{b_1, \dots, (\text{not}) b_{i-1}, (\text{not}) b_{i+1}, \dots, \text{not } b_m\}$ 
2.29             Transform Literal ((not)  $b_i$ )
2.30             Pop ((not)  $b_i$ , stack)
2.31             context = previous context
2.32         end
2.33     end
2.34 end

```

**Algorithm 2:** Transform Literal

---

OLON on  $a$ 's rule. After all OLONS have been solved, we use the Contexts to create new rules that preserve the meaning of the original ones, except the new ones do not now depend on OLONS. The current Context for  $a$  is now just  $\{b\}$  instead of the original  $\{not\ a, b\}$ .

Solving a query for  $b$ , we go on to solve  $c$  —  $\{c\}$  being  $b$ 's current Context. Solving  $c$  we find leaf  $not\ b$ . We remove  $c$  from  $b$ 's Context, and add  $c$ 's body  $\{not\ b, not\ a\}$  to it. The OLON on  $b$  is detected and the  $not\ b$  is removed from  $b$ 's Context, which finally is just  $\{not\ a\}$ . As can be seen so far, updating Contexts is similar to performing an unfolding plus OLON detection and resolution by removing the dependency on the OLON. The new rule for  $b$  has final Context  $\{not\ a\}$  for body. I.e., the new rule for  $b$  is  $b \leftarrow not\ a$ . Next, continuing  $a$ 's final Context calculation, we remove  $b$  from  $a$ 's Context and add  $\{not\ a\}$  to it. This additional OLON is detected and  $not\ a$  is removed from  $a$ 's Context, now empty. Since we already exhausted  $a$ 's dependency call-graph, the final body for the new rule for  $a$  is empty:  $a$  will be added as a fact. Moreover, a new rule for  $b$  will be added:  $b \leftarrow not\ a$ . The final transformed program is:

$$a \leftarrow not\ a, b \quad a \quad b \leftarrow c \quad b \leftarrow not\ a \quad c \leftarrow not\ b, not\ a$$

it has only one  $SM = \{a\}$  the only LSM of the program. Mark layering is respected when solving OLONS:  $a$ 's final rule depends on the answer to  $b$ 's final rule.

**Example 5. Solving OLONS (2).** Consider this program, coinciding with its residual:

$$a \leftarrow not\ b, x \quad b \leftarrow not\ c, y \quad c \leftarrow not\ a, z \quad x \quad y \quad z$$

Solving a query for  $a$  we push it onto the stack, and take its rule  $a \leftarrow not\ b, x$ . We go on for literal  $not\ b$  and consider the rest of the body  $\{x\}$  as the current Context under which the OLON on  $a$  is “active”. Push  $not\ b$  onto the stack and take the rule for  $b$ . We go on to solve  $not\ c$ , and add the  $y$  to the current Context which now becomes  $\{x, y\}$ . Once more, push  $not\ c$  onto the stack, take  $c$ 's rule  $c \leftarrow not\ a, z$ , go on to solve  $not\ a$  and add  $z$  to the current Context which is now  $\{x, y, z\}$ . When we now push  $not\ a$  onto the stack, the OLON is detected and it “solving” begins. Three rules are created and added to the program  $a \leftarrow not\ c, x, y, z$ ,  $b \leftarrow not\ a, x, y, z$ , and  $c \leftarrow not\ b, x, y, z$ . Together with the original program's rules they render “stable” the originally “non-stable” LSM  $\{a, b, x, y, z\}$ ,  $\{b, c, x, y, z\}$ , and  $\{a, c, x, y, z\}$ . The final transformed program is:

$$\begin{array}{llll} a \leftarrow not\ b, x & b \leftarrow not\ c, y & c \leftarrow not\ a, z & x \quad y \quad z \\ a \leftarrow not\ c, x, y, z & b \leftarrow not\ a, x, y, z & c \leftarrow not\ b, x, y, z & \end{array}$$

**TR transformation correctness** The TR transformation steps occur only when OLONS are detected, and in those cases the transformation consists in adding extra rules. So, when there are no OLONS, the TR transformation's effect is  $P' = P$ , thus preserving the SMs of the original  $P$ . The additional Layer Supported Models of  $P$  are obtained by “solving” OLONS (by adding new rules in  $P'$ ), so that the order of OLON solving complies with the layers of  $P$ . This is ensured because the top-down search, by its nature, solves OLONS conditional on their Context, and the latter will include same or lower layer literals, but not above layer ones. Finally, note Stable Models evaluation of  $P'$  itself respects the Well-Founded Semantics and hence Contexts evaluation respects layering, by Theorem 2.

## 5.2 Number of Models

Loop detection and the variety of their possible solutions concerns the number of Strongly Connect Components (SCCs) of the residual program.

**Theorem 3.** *Maximum number of SCCs and of LSMs of a strongly connected residual component with  $N$  nodes. They are, respectively,  $\frac{N}{3}$  and  $3^{\frac{N}{3}}$ .*

*Proof.* Consider a component containing  $N$  nodes. A single odd loop with  $N$  nodes, by itself, contains  $N$  LSMs: each one obtained by minimally solving the implicit disjunction of the heads of the rules forming the OLON. Given only two OLONs in the component, with  $N_1 + N_2 = N$  nodes, they could conceivably always be made independent of each other. Independent in the sense that each and every solution of one OLON combines separately with each and every solution of the other. To achieve this, iteratively duplicate as needed the rules of each OLON such that the combination of values of literals from the other loop are irrelevant. For example, in a program like

$$\begin{array}{lll} a \leftarrow \text{not } b & b \leftarrow \text{not } c & c \leftarrow \text{not } a, e \\ e \leftarrow \text{not } d & d \leftarrow \text{not } f & f \leftarrow \text{not } e, c \end{array}$$

add the new rules  $c \leftarrow \text{not } a, \text{not } e$  and  $f \leftarrow \text{not } e, \text{not } c$  so that now the loop on  $a, b, c$  becomes independent of the truth of  $e$ , and the loop on  $e, d, f$  becomes independent of the truth of  $c$ . So, in the worst (more complex) case of two OLONs the number of LSMs is  $N_1 * N_2$ , in this case  $3 * 3 = 9$ .

Each loop over an even number of default negations (ELON), all by itself contains 2 LSMs, independently of the number  $N$  of its nodes. An OLON can always be made independent of an ELON by suitably and iteratively duplicating its rules, as per above. So an OLON with  $N_1$  nodes dependent on a single ELON will, in the worst case, provide  $2 * N_1$  LSMs.

It is apparent the highest number of LSMs in a component with  $N$  nodes can be gotten by combining only OLONs. Moreover, we have seen, the worst case is when these are independent.

Consider a component with  $N$  nodes and two OLONs with nodes  $N_1 + N_2 = N$ . The largest value of  $N_1 * N_2$  is obtained when  $N_1 = N_2 = \frac{N}{2}$ . Indeed, since  $N_2 = N - N_1$ , take the derivative  $\frac{d(N_1 * (N - N_1))}{dN_1} = \frac{d(N * N_1)}{dN_1} - \frac{dN_1^2}{dN_1} = N - 2 * N_1$ . To obtain the highest value make the derivative  $N - 2 * N_1 = 0$ , and hence  $N_1 = \frac{N}{2}$ .

Similarly, for  $\sum_i N_i = N$ , so  $N_i = \frac{N}{i}$  gives the maximum value for  $\prod_i N_i$ . Thus, the maximum number of LSMs for a component of  $N$  nodes is obtained when all its (odd) loops have the same size.

And what is the size  $i$  that maximizes this value? Let us again use a derivative in  $i$ , in this case  $\frac{di \frac{N}{i}}{di}$  as the number of LSMs is  $i^{\frac{N}{i}}$ . Now  $\frac{di \frac{N}{i}}{di} = -N * i$ . Equating it to zero we have  $i = 0$ . But  $i$  must be greater than zero and less than  $N$ . It is easy to see that the  $i$  that affords the value of the derivative closest to zero is  $i = 1$ . But OLONs of length 1 afford no choices hence the least  $i$  that is meaningful is  $i = 3$ .

Hence the maximum number of LSMs of a component with  $N$  nodes is  $3^{\frac{N}{3}}$ .  $\square$

**Theorem 4.** *Maximum number of ELONs and of SMs of a SCC component with  $N$  nodes. These are, respectively,  $\frac{N}{2}$  and  $2^{\frac{N}{2}}$ .*

---

*Proof.* By the same reasoning as above, the maximum number of SMs of a component with  $N$  nodes is  $2^{\frac{N}{2}}$ , since there are no OLONs in SMs and so  $i$  can only be 2.  $\square$

**Corollary 1.** *Comparison between number of possible models of LSMs and SMs. The highest number of models possible for LSMs, #LSMs, is larger than that for SMs, #SMs.*

*Proof.* By the two previous theorems, we know that for a component with  $N$  nodes,

$$\frac{\#LSMs}{\#SMs} = \frac{3^{\frac{N}{3}}}{2^{\frac{N}{2}}} = 3^{(N * [\frac{1}{3} - \frac{1}{2} * (\log_3 2)])}. \quad \square$$

## 6 Implementation

The XSB Prolog system<sup>2</sup> is one of the most sophisticated, powerful, efficient and versatile implementations, with a focus on execution efficiency and interaction with external systems, implementing program evaluation following the WFS for NLPs. The XASP interface [1] (standing for XSB Answer Set Programming), is included in XSB Prolog as a practical programming interface to Smodels [9], one of the most successful and efficient implementations of the SMs over generalized LPs. The XASP system allows one not only to compute the models of a given NLP, but also to effectively combine 3-valued with 2-valued reasoning. The latter is achieved by using Smodels to compute the SMs of the so-called residual program, the one that results from a query evaluated in XSB using tabling [13]. A residual program is represented by delay lists, that is, the set of undefined literals for which the program could not find a complete proof, due to mutual dependencies or loops over default negation for that set of literals, detected by the XSB tabling mechanism. This coupling allows one to obtain a two-valued semantics of the residual, by completing the three-valued semantics the XSB system produces. The integration also allows to make use of and benefit from the relevance property of LSM semantics by queries.

In our implementation, detailed below, we use XASP to compute the query relevant residual program on demand. When the TR transformation is applied to it, the resulting program is sent to Smodels for computation of stable models of the relevant sub-program provided by the residue, which are then returned to the XSB-XASP side.

**Residual Program** After launching a query in a top-down fashion we must obtain the relevant residual part of the program for the query. This is achieved in XSB Prolog using the `get_residual/2` predicate. According to the XSB Prolog’s manual “the predicate `get_residual/2` unifies its first argument with a tabled subgoal and its second argument with the (possibly empty) delay list of that subgoal. The truth of the subgoal is taken to be conditional on the truth of the elements in the delay list”. The delay list is the list of literals whose truth value could not be determined to be *true* nor *false*, i.e., their truth value is *undefined* in the WFM of the program.

---

<sup>2</sup> XSB-Prolog and Smodels are freely available, at: <http://xsb.sourceforge.net> and <http://www.tcs.hut.fi/Software/smodels>.



---

It is possible to obtain the *residual* clause of a solution for a query literal, and in turn the *residual* clauses for the literals in its body, and so on. This way we can reconstruct the complete relevant residual part of the KB for the literal — we call this a *residual program* or *reduct* for that solution to the query.

More than one such *residual program* can be obtained for the query, on backtracking. Each *reduct* consists only of partially evaluated rules, with respect to the WFM, whose heads are atoms relevant for the initial query literal, and whose bodies are just the *residual* part of the bodies of the original KB's rules. This way, not only do we get just the relevant part of the KB for the literal, we also get precisely the part of those rules bodies still *undefined*, i.e., those that are involved in Loops Over Negation.

**Dealing with the Query and Integrity Constraints** ICs are written as just  $falsum \leftarrow IC\_Body$ . An Smodels IC preventing *falsum* from being *true* ( $:- falsum$ ) is enforced whenever a transformed program is sent to Smodels. Another two rules are added to the Smodels clause store through XASP: one creates an auxiliary rule for the initially posed query; with the form: `lsmGoal :- Query`, where *Query* is the query conjunct posed by the user. The second rule just prevents Smodels from having any model where the `lsmGoal` does not hold, having the form:  $:- not\ lsmGoal$ .

The XSB Prolog source code for the meta-interpreter, based on this program transformation, is available at <http://centria.di.fct.unl.pt/~amp/software.html>

## 7 Conclusions and Future Work

We have recapped the LSMs semantics for *all* NLPs, complying with desirable requirements: 2-valued semantics, conservatively extending SMs, guarantee of model existence, relevance and cumulativity, plus respecting the WFM.

We have exhibited a space and time linearly complex transformation, TR, from one propositional NLP into another, whose Layer Supported Models are precisely the Stable Models of the transform, which can then be computed by extant Stable Model implementations. TR can be used to answer queries but is also of theoretical interest, for it may be used to prove properties of programs. Moreover, it can be employed in combination with the top-down query procedure of XSB-Prolog, and be applied solely to the residual program corresponding to a query. The XSB-XASP interface subsequently allows the program transform to be sent for Smodels for 2-valued evaluation.

The applications afforded by LSMs are *all* those of SMs, plus those where odd loops over default negation (OLONs) are actually employed for problem domain representation, as we have shown in examples 4 and 5. The guarantee of model existence is essential in applications where knowledge sources are diverse (like in the Semantic Web), and wherever the bringing together of such knowledge (automated or not) can give rise to OLONs that would otherwise prevent the resulting program from having a semantics, thereby brusquely terminating the application. A similar situation can be brought about by self- and mutually-updating programs, including in the learning setting, where unforeseen OLONs would stop short an ongoing process if the SM semantics were in use. Hence, apparently there is only to gain in exploring the adept move from SM semantics

---

to the more general LSM one, given that the latter is readily implementable through the program transformation TR, introduced here for the first time.

Work under way concerns an XSB engine level efficient implementation of the LSM semantics, and the exploration of its wider scope of applications with respect to ASP, and namely in combination with abduction and constructive negation.

Finally, the concepts and techniques introduced in this paper are readily adoptable by other logic programming systems and implementations.

## References

1. L. Castro, T. Swift, and D. S. Warren. *XASP: Answer Set Programming with XSB and Smodels*. <http://xsb.sourceforge.net/packages/xasp.pdf>.
2. S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The dl<sub>v</sub> system: Model generator and advanced frontends (system description). In *Workshop in Logic Programming*, 1997.
3. C.V. Damásio and L. M. Pereira. Default negated conclusions: Why not? In R. Dyckhoff et al, editor, *Extensions of Logic Programming, ELP'96*, volume 1050 of *LNAI*, pages 103–118. Springer-Verlag, 1996.
4. J. Dix. A Classification-Theory of Semantics of Normal Logic Programs: I, II. *Fundamenta Informaticae*, XXII(3):227–255, 257–288, 1995.
5. J. Dix, G. Gottlob, V.W. Marek, and C. Rauszer. Reducing disjunctive to non-disjunctive semantics by shift-operations. *Fundamenta Informaticae*, 28:87–100, 1996.
6. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. of ACM*, 38(3):620–650, 1991.
7. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, pages 1070–1080. MIT Press, 1988.
8. M. Gelfond, H. Przymusinska, V. Lifschitz, and M. Truszczyński. Disjunctive defaults. In *KR-91*, pages 230–237, 1991.
9. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Procs. LPNMR'97*, LNAI 1265, pages 420–429, 1997.
10. L.M. Pereira and A.M. Pinto. Layered models top-down querying of normal logic programs. In *Procs. PADL'09*, volume 5418 of *LNCS*, pages 254–268. Springer, January 2009.
11. Luís Moniz Pereira and Alexandre Miguel Pinto. Layer supported models of logic programs. In E. Erdem, F. Lin, and T. Schaub, editors, *Procs. 10th LPNMR*, LNAI. Springer, September 2009. <http://centria.di.fct.unl.pt/~lmp/publications/online-papers/LSMs.pdf> (long version).
12. T.C. Przymusiński. Perfect model semantics. In *ICLP/SLP*, pages 1081–1096, 1988.
13. T. Swift. Tabling for non-monotonic programming. *AMAI*, 25(3-4):201–240, 1999.
14. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1(2):146–160, 1972.

---

# Towards Computing Revised Models for FO Theories

Johan Wittocx\*, Broes De Cat, and Marc Denecker

Department of Computer Science, K.U. Leuven, Belgium  
{johan.wittocx,broes.decat,marc.denecker}@cs.kuleuven.be

**Abstract.** In many real-life computational search problems, one is not only interested in finding a solution, but also in maintaining it under varying circumstances. E.g., in the area of network configuration, an initial configuration of a computer network needs to be obtained, as well as a new configuration when one of the machines in the network breaks down. Currently, most such revision problems are solved manually, or with highly specialized software.

A recent declarative approach to solve (hard) computational search problems involving a lot of domain knowledge, is by finite model generation. Here, the domain knowledge is specified as a logic theory  $T$ , and models of  $T$  correspond to solutions of the problem. In this paper, we extend this approach to also solve revision problems. In particular, our method allows to use the same theory to describe the search problem and the revision problem, and applies techniques from current model generators to find revised solutions.

## 1 Introduction

In many real-life search problems, one searches for objects of a complex nature, such as an assignment, a plan or a schedule. Often, these objects can be represented by a finite structure and implicitly described by a logic theory. This observation led to declarative problem solving paradigms based on finite model generation such as Answer Set Programming [7, 11], NP-SPEC [2] and the Model Expansion framework [9]. Several efficient solvers for these frameworks have been developed, making them applicable in practice.

Often, one is not (only) interested in a single solution to a search problem, but also in *revising* this solution under varying circumstances. E.g., a network administrator is interested in computing an initial configuration of a computer network as well as in maintaining the configuration when one of the machines in the network breaks down. Typically, the following are requirements for a revised solution:

1. To allow a fast reaction on new circumstances, computing a revision should be efficient.

---

\* Research assistant of the Fund for Scientific Research - Flanders (FWO-Vlaanderen)

- 
2. Executing a proposed revision in the problem domain usually has a certain cost. E.g., it takes time to move mail servers from one computer to another. A good revision should preferably have a low cost. I.e., it should be executable by a small number of cheap operations.

Most existing approaches to solve revision problems are tailored towards a specific application such as train rescheduling, and hence, they are not (entirely) declarative. In this paper, we present a revision method that is much closer to the declarative problem solving frameworks mentioned above. Formally, we describe a revision problem by a theory  $T$ , a finite model  $M$  of  $T$  and a set of atoms  $C$ :  $T$  describes the original search problem,  $M$  a solution to that problem and  $C$  the atoms that should swap truth value compared to  $M$ . A solution to the revision problem is a model  $M'$  of  $T$  such that for every atom in  $C$ , its truth value in  $M'$  is opposite to its truth value in  $M$ . Revision problems where the theory  $T$  or the domain of  $M$  is changed can be reduced to this case, as we show in Section 3.2.

In this paper, we describe a method to solve revision problems where  $T$  is a first-order logic (FO) theory. Like finite model generation for FO, these problems can easily be reduced to SAT. An off-the-shelf SAT solver can then be used to solve the resulting SAT problem. However, directly using this approach does not satisfy the two requirements mentioned above. It does not guarantee that the revised model is close to the original model and it can take too much time and space to create and store the SAT problem. The method we propose avoids these problems by first constructing a set of atoms  $S'$ , called a *search bound*. The bound contains all the atoms that we allow to swap truth value to obtain a revised model  $M'$  from the original one  $M$ . Then the method tries to find, by reducing to SAT, such a revised model  $M'$ . If it does not succeed, the bound  $S'$  is enlarged and the process is repeated. However, if it succeeds and  $S'$  is relatively small, the two requirements are met. Only a small number of operations should be performed to obtain  $M'$ , because the only changes are on atoms in  $S'$ . Also, reducing to SAT can be efficient if  $S'$  is small.

To make the approach work, the consecutive search bounds should be constructed such that there is a *reasonable chance* that a revised model bounded by them exists. In Section 5, we present a non-deterministic algorithm to compute such search bounds. Experiments with a prototype implementation are discussed in Section 6. They indicate that the algorithm often finds small search bounds containing a revision. We end with conclusions and topics for future work.

## 2 Preliminaries

We assume the reader is familiar with classical first-order logic (FO), see, e.g., [4]. We introduce the conventions and notations used in this paper. Without loss of generality, we consider function-free FO in this paper and assume that all negations ( $\neg$ ) occur directly in front of atoms. I.e., we assume that every formula is in negation normal form.

A vocabulary  $\Sigma$  consists of variables and predicate symbols. Variables are denoted by lowercase letters, predicate symbols by uppercase letters. Sets and

tuples of variables are denoted by  $\bar{x}, \bar{y}, \dots$ . For a formula  $\varphi$ , we often write  $\varphi[\bar{x}]$  to denote that  $\bar{x}$  are its free variables. If  $\varphi[x]$  is a formula and  $d$  an element from a domain  $D$ , we denote by  $\varphi[d]$  the result of replacing all free occurrences of  $x$  in  $\varphi$  by  $d$ . This notation is extended to tuples of variables and domain elements of the same length. We say that a formula  $\psi$  occurs positively (negatively) in another formula  $\varphi$  if it occurs in the scope of an even (odd) number of negations.

A literal is an atom (positive literal) or its negation (negative literal). By  $|L|$  we denote the atom  $P(\bar{x})$  if  $L$  is the literal  $P(\bar{x})$  or  $\neg P(\bar{x})$ . For an atom (literal)  $L[\bar{x}]$  and a tuple of domain elements  $\bar{d}$  from a domain  $D$ ,  $L[\bar{d}]$  is called a *domain atom (literal) over  $D$* . We denote the set of all domain atoms over  $D$  by  $\mathcal{A}(D)$ .

A  $\Sigma$ -structure  $I$  with domain  $D$  is an assignment of a relation  $P^I \subseteq D^n$  to every  $n$ -ary predicate symbol  $P \in \Sigma$ . A structure is called *finite* if its domain is finite. In this paper, all structures are finite. We say that a domain atom  $P(\bar{d})$  is *true*, respectively *false*, in  $I$  if  $\bar{d} \in P^I$ , respectively  $\bar{d} \notin P^I$ . If  $R$  is a set of domain atoms, we denote by  $\text{swap}(I, R)$  the structure obtained from  $I$  by swapping the truth values of the domain atoms in  $R$ . I.e.,  $\bar{d} \in P^{\text{swap}(I, R)}$  if one of the following holds:

- $\bar{d} \in P^I$  and  $P(\bar{d}) \notin R$ ;
- $\bar{d} \notin P^I$  and  $P(\bar{d}) \in R$ .

The satisfaction relation  $\models$  is defined as usual [4].

### 3 The Revision Problem

#### 3.1 Basic Revision Problems

For the rest of this paper, we assume a fixed vocabulary  $\Sigma$  and finite domain  $D$ . Unless stated otherwise, every structure has domain  $D$ . We now formally define the revision problem. Let  $T$  be a theory,  $M$  a finite model of  $T$  and  $C$  a set of domain atoms. A *revision* for input  $\langle M, T, C \rangle$  is a set  $R$  of domain atoms such that  $R \cap C = \emptyset$  and  $\text{swap}(M, R \cup C) \models T$ . Intuitively,  $M$  describes the original solution to a problem,  $C$  the changes that occurred, and  $R$  the changes that should be made to repair the currently wrong solution  $\text{swap}(M, C)$ . We call  $\text{swap}(M, R \cup C)$  a *revised model*.

*Example 1.* Consider the problem of placing  $n$  non-attacking rooks on an  $n \times n$  chessboard. A model with domain  $\{1, 2, \dots, n\}$  of the following theory  $T_1$  describes a solution to this problem. Here, atom  $R(r, c)$  means “there is a rook on square  $(r, c)$ ”.

$$\begin{aligned} & \forall r \exists c R(r, c). \\ & \forall c \exists r R(r, c). \\ & \forall r \forall c_1 \forall c_2 (R(r, c_1) \wedge R(r, c_2) \supset c_1 = c_2). \\ & \forall r_1 \forall r_2 \forall c (R(r_1, c) \wedge R(r_2, c) \supset r_1 = r_2). \end{aligned}$$

---

For  $n = 3$ ,  $M_1 = \{R(1, 1), R(2, 3), R(3, 2)\}$  is a model. Assume we have computed  $M_1$ , but for some reason, we do not want a rook on position  $(1, 1)$ . That means we have to search for a revision for  $\langle T_1, M_1, C_1 \rangle$ , where  $C_1 = \{R(1, 1)\}$ . An example revision is the set  $R_1 = \{R(1, 3), R(2, 1), R(2, 3)\}$ , which yields the revised model  $\{R(1, 3), R(2, 1), R(3, 2)\}$ .

In practice, it is often the case that not all atoms are allowed to swap truth value in order to obtain a revised model. E.g., in a train rescheduling problem, the truth value of atoms that state the positions of the railway stations should never change, because the position of the stations cannot be changed in reality. To formally describe such a problem, let  $S$  be a set of domain atoms, disjoint from  $C$ . Intuitively,  $S$  is the set of atoms that are allowed to swap truth value, i.e. the search space to find a revision. A revision  $R$  for  $\langle T, M, C \rangle$  is *bounded* by  $S$  if  $R \subseteq S$ . The *bounded revision problem* with input  $\langle T, M, C, S \rangle$  is the problem of finding a revision that is bounded by  $S$ .

### 3.2 Domain and Theory Changes

Besides the revision problem as described above, i.e., the problem of computing a new model when the truth values of some of the domain atoms change, one could also consider the problem of computing a new model when either<sup>1</sup>:

1. A new sentence is added to the theory.
2. A domain element is left out of the old model.
3. A new domain element is added to the old model.

All three problems can easily be reduced to the revision problem defined above.

To reduce (1), let  $\varphi$  be the sentence that is added, let  $P$  be a new propositional atom and  $M$  a model of  $T$  such that  $P$  is false in  $M$ . Then a revision for  $\langle T \cup \{P \supset \varphi\}, M, \{P\} \rangle$  is a model for  $T \cup \{\varphi\}$ . To reduce (2), let  $Used$  be a new unary predicate and denote by  $T'$  the theory obtained by replacing each subformula in  $T$  of the form  $(\forall x \varphi)$ , respectively  $(\exists x \varphi)$ , by  $(\forall x (Used(x) \supset \varphi))$ , respectively  $(\exists x (Used(x) \wedge \varphi))$ . Let  $U$  be the set of all domain atoms of the form  $Used(d)$ ,  $M$  a model of  $T$  such that each atom in  $U$  is true in  $M$ , and denote by  $d'$  the domain element that is left out. A revision for  $\langle T', M, \{Used(d')\}, \mathcal{A}(D) \setminus U \rangle$ , restricted to the atoms not mentioning  $d'$ , is a model for  $T$ . In a similar way, (3) can be reduced.

### 3.3 Weighted Revision Problems

In real-life revision problems, one is often more interested in a revision with a *low cost*, than in a small revision. E.g., suppose some problem in a network can be solved by either moving multiple DHCP servers, or by moving only one mail server. Although the revision describing the first solution will have a higher

---

<sup>1</sup> Observe that, due to the monotonicity of FO, it is trivial to compute a new model when a sentence of  $T$  is left out.

---

cardinality than the second one, the cost of performing the second one is higher, since moving a mail server involves moving all mailboxes of users. Moving a DHCP server only involves copying a single script.

To model revision problems where the cost of the revision plays an important role, a pair  $(c_+, c_-)$  of positive numbers is assigned to each domain atom  $P(\bar{d})$ . These two numbers indicate the cost of changing the truth value of  $P(\bar{d})$  from false to true, respectively from true to false. E.g., if  $P(d)$  means that there is a mail server on machine  $d$ ,  $c_+$  indicates the cost of installing a mail server on  $d$ , while  $c_-$  indicates the cost of uninstalling it. A good revision only contains atoms that are false, respectively true, in the original model and are assigned a low  $c_+$ , respectively  $c_-$ .

## 4 Solving the Revision Problem

In the rest of this paper, we assume a fixed  $T$ ,  $M$ ,  $C$  and  $S$  and consider the bounded revision problem for input  $\langle T, M, C, S \rangle$ . The problem  $\langle T, M, C, S \rangle$  can be solved by reducing it to the problem of finding a model of a propositional theory  $T_g$ , called a *grounding*. The model generation problem can then be solved by an off-the-shelf efficient SAT solver [10]. To find revisions with low cardinality (or low cost in case of a weighted revision problem), one can use a SAT solver with support for optimization, such as a max-SAT solver [6, 14].

### 4.1 Grounding

A formula  $\varphi$  is in *ground normal form (GNF)* if it is a boolean combination of domain atoms. I.e.,  $\varphi$  is a sentence containing no quantifiers or variables. A GNF theory is a theory containing only GNF formulas. Observe that a GNF theory is essentially a propositional theory.

**Definition 1.** A grounding for  $\langle T, M, C, S \rangle$  is a GNF theory  $T_g$  such that for every  $R \subseteq S$ ,  $R$  is a revision for  $\langle T, M, C, S \rangle$  iff  $\text{swap}(M, C \cup R) \models T_g$ . A grounding is called *reduced* if all domain atoms that occur in it belong to  $S$ .

A basic grounding algorithm consists of recursively replacing every universal subformula  $\forall x \varphi[x]$  in  $T$  by the conjunction  $\bigwedge_{d \in D} \varphi[d]$  and every existential subformula  $\exists x \varphi[x]$  by  $\bigvee_{d \in D} \varphi[d]$ . The resulting grounding is called the *full grounding*. The size of the full grounding is polynomial in the size of  $D$  and exponential in the maximum width of a formula in  $T$ . A reduced grounding can be obtained by applying the following result.

**Lemma 1.** Let  $T_g$  be a grounding for  $\langle T, M, C, S \rangle$  and  $\varphi$  a subformula of  $T_g$ . If for every  $R \subseteq S$ ,  $\varphi$  is true (false) in  $\text{swap}(M, C \cup R)$ , then the theory obtained by substituting  $\top$  ( $\perp$ ) for  $\varphi$  in  $T_g$  is also a grounding for  $\langle T, M, C, S \rangle$ .

According to this lemma, all formulas in  $T_g$  that do not contain an atom of  $S$  can be replaced by  $\top$  or  $\perp$ , according to their truth value in  $\text{swap}(M, C)$ . The

---

result is a reduced grounding for  $\langle T, M, C, S \rangle$ . Observe that for a small search space  $S$ , the size of a reduced grounding can be considerably smaller than the full grounding size. Smart grounding algorithms avoid creating the full grounding by applying Lemma 1 as soon as possible (see, e.g., [12, 13, 16]).

## 4.2 Optimized Grounding

If the search space  $S$  is small, a grounder that produces a reduced grounding spends most of its time evaluating formulas in  $\text{swap}(M, C)$ . Evaluating a formula  $\varphi$  in a structure takes polynomial space in the size of  $\varphi$ . We can exploit the fact that  $M$  is a model of  $T$  to evaluate some subformulas in  $T$  much more efficiently.

Inductively define the set  $\tau(T)$  by

- If  $\varphi$  is a sentence of  $T$ , then  $\varphi \in \tau(T)$ ;
- If  $\varphi_1 \wedge \varphi_2 \in \tau(T)$ , then  $\varphi_1 \in \tau(T)$  and  $\varphi_2 \in \tau(T)$ ;
- If  $\forall x \varphi[x] \in \tau(T)$ , then  $\varphi[x] \in \tau(T)$ .

If  $\varphi[\bar{x}]$  is a formula of  $\tau(T)$ , then clearly  $M \models \varphi[\bar{d}]$  for every tuple of domain elements  $\bar{d}$ . If  $\varphi$  does not contain *dangerous literals* then it remains true in every revised model:

**Definition 2.** A domain literal  $L[\bar{d}]$  is dangerous in a formula  $\varphi$  with respect to  $S$  if  $L[\bar{d}] \in C \cup S$ ,  $M \models L[\bar{d}]$  and for some tuple of variables  $\bar{x}$ ,  $L[\bar{x}]$  occurs positively in  $\varphi$ .

Intuitively, a literal is dangerous in  $\varphi$  if it has a negative influence on the truth value of  $\varphi$  in  $\text{swap}(M, C \cup S)$ , while it had a positive influence in  $M$ . We denote the set of all dangerous literals in  $\varphi$  with respect to  $S$  by  $\mathcal{D}_S(\varphi)$ .

**Lemma 2.** Let  $\varphi[\bar{x}] \in \tau(T)$  and  $\bar{d}$  a tuple of domain elements. If  $\mathcal{D}_S(\varphi[\bar{d}]) = \emptyset$ , then  $\text{swap}(M, C \cup R) \models \varphi[\bar{d}]$  for every  $R \subseteq S$ .

As such, a grounder can safely substitute formulas that satisfy the conditions of Lemma 2 by  $\top$ . Checking these conditions for a formula  $\varphi$  takes only linear time in the size of  $\varphi$ .

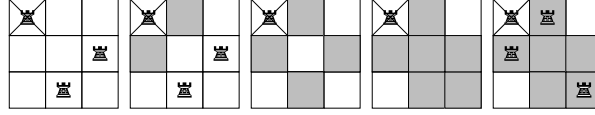
## 5 Reducing the Search Space

### 5.1 Search Bounds

Directly solving a revision problem  $\langle T, M, C, S \rangle$  by reducing it to SAT has the drawback that there is no guarantee that the revised model that is found is *close* to  $M$ , i.e., that the cardinality of the corresponding revision is low. The revision  $R$  can be as large as  $S$ . On the other hand, one can often find a revision with a cardinality much lower than the number of domain atoms that are allowed to swap truth value. E.g., in a network configuration problem, a problem in a certain subnet can often be repaired by only making changes within that subnet,



**Fig. 1.** Enlarging the search bound



not touching the rest of the network. In Example 1, the revision contains three atoms, while there are  $n \times n$  domain atoms in total.

The observation above suggests the following approach to find a revision for  $\langle T, M, C, S \rangle$ . First, construct a *small* subset  $S'$  of  $S$  such that there might exist a revision for  $\langle T, M, C, S' \rangle$ . We will call  $S'$  the *search bound*. Then, try to find a revision for  $\langle T, M, C, S' \rangle$  by reducing to SAT. If a revision  $R$  is found, this is also a revision for  $\langle T, M, C, S \rangle$ , because  $R \subseteq S' \subseteq S$ . If, on the other hand, there is no revision for  $\langle T, M, C, S' \rangle$ , the search bound  $S'$  is enlarged. This process is repeated until a revision is found or  $S' = S$ . An example run of this algorithm for Example 1 is shown in Figure 1, where the grey squares represent the domain atoms in the consecutive  $S'$ . There are several important benefits of using this approach compared to directly reducing to SAT:

- If a small search bound  $S'$  can be detected such that there is a revision for  $\langle T, M, C, S' \rangle$ , such a revision is small. Hence, the corresponding revised model is close to the original model  $M$ .
- The size of the reduced grounding for  $\langle T, M, C, S' \rangle$  is small when  $S'$  is small. Indeed, all atoms that occur in the reduced grounding are atoms of  $S'$ . In general, the smaller the grounding, the faster the SAT solver can produce its answer.
- A small  $S'$  might speed up the grounding considerably, since the number of formulas that satisfy the conditions of Lemma 2 depends on the size of  $S'$ .

## 5.2 Enlarging the Search Bound

Assume that we have constructed a search bound  $S'$  that is not large enough. I.e., there is no revision for  $\langle T, M, C, S' \rangle$ . Lemma 2 implies that this problem is caused by the dangerous literals in the sentences of  $T$  with respect to  $S'$ . If we want to enlarge  $S'$  to a new search bound  $S''$  such that there is a reasonable chance that there is a revision for  $\langle T, M, C, S'' \rangle$ , we should add atoms to  $S'$  that can have a *positive influence* on the sentences of  $T$ . I.e., if they swap truth value compared to  $M$ , then some literals that occur positively in  $T$  become true. More precisely:

**Definition 3.** A domain atom  $P(\bar{d})$  has positive influence in a formula  $\varphi$  if there is a positive occurrence of a literal  $L[\bar{x}]$  in  $T$  such that  $|L| = P$  and  $M \not\models L[\bar{d}]$ .

---

In most cases, it is not beneficial to choose the atoms to add to  $S'$  randomly among the atoms with a positive influence on  $T$ . Rather, atoms that are added should be able to *neutralize* the negative influence of the dangerous literals. E.g., if a literal  $L \in \mathcal{D}_{S'}(\psi_1 \vee \psi_2)$  and  $L$  occurs in  $\psi_1$ , then its negative influence can be neutralized by making  $\psi_2$  true. Hence in this case, one should add atoms to  $S'$  that have a positive influence on  $\psi_2$ . Observe that a dangerous occurrence of a literal  $L$  in a conjunction  $\psi_1 \wedge \psi_2$  cannot be neutralized. If  $\psi_1$  is false, then even if  $\psi_2$  is true, the conjunction is false. These intuitions are formalized in the following definitions.

**Definition 4.** Let  $L[\bar{d}] \in \mathcal{D}_{S'}(\varphi)$  and  $\psi[\bar{x}]$  subformula of  $\varphi$ . We call  $\psi[\bar{d}']$  a neutralizable formula for  $L[\bar{d}]$  in  $\varphi$  if

- $\psi$  is a disjunction or an existentially quantified formula,
- $L[\bar{d}] \in \mathcal{D}_{S'}(\psi[\bar{d}'])$  and
- $\text{swap}(M, C \cup R) \not\models \psi[\bar{d}']$  for some  $R \subseteq S'$ .

The third condition expresses that  $\psi[\bar{d}']$  can become false when some of the atoms in  $S' \cup C$  swap truth value. I.e.,  $\psi[\bar{d}']$  can be a reason that there is no revision bounded by  $S'$ .

**Definition 5.** Let  $\psi$  be a neutralizable formula for  $L[\bar{d}]$  in  $\varphi$ . A domain atom  $P(\bar{d}')$  is a neutralizer for  $L[\bar{d}]$  in  $\psi$  if one of the following holds:

- $\psi = \bigvee_{1 \leq i \leq n} \chi_i$ ,  $L[\bar{d}] \in \mathcal{D}_{S'}(\chi_j)$  and  $P(\bar{d}')$  has positive influence in  $\chi_k$  for some  $k \neq j$ .
- $\psi = \exists x \chi[x]$ ,  $L[\bar{d}] \in \mathcal{D}_{S'}(\chi[d''])$  and  $P(\bar{d}')$  has positive influence in  $\chi[d''']$  for some domain element  $d''' \neq d''$ .

*Example 2.* Let  $\varphi$  be the propositional formula  $((A \vee B) \wedge C) \vee (D \wedge E)$  and let  $M = \{A, C, E\}$ ,  $C = \{A\}$  and  $S' = \emptyset$ . Then  $A$  is dangerous in  $\varphi$ . Its negative influence is clear:  $M \models \varphi$ , but  $\text{swap}(M, C \cup S') \not\models \varphi$ . There are two neutralizable formulas for  $A$  in  $\varphi$ :  $A \vee B$  and  $\varphi$ .  $B$  is a neutralizer for the former,  $D$  for the latter. Observe that both  $\text{swap}(M, C \cup \{B\})$  and  $\text{swap}(M, C \cup \{D\})$  satisfy  $\varphi$ . This suggests to add  $B$  and/or  $D$  to  $S'$  to obtain a new search bound.

*Example 3 (Ctd. from Example 1).* Let  $S' = \{R(2, 1)\}$ . Then  $R(1, 1)$  is dangerous in  $\forall r \exists c R(r, c)$ . The only neutralizing formula in this case is  $\exists c R(1, c)$ , yielding  $R(1, 2)$  and  $R(1, 3)$  as neutralizers.  $\neg R(2, 1)$  is dangerous in  $\forall r \forall c_1 \forall c_2 (R(r, c_1) \wedge R(r, c_2) \supset c_1 = c_2)$ . For every domain element  $d \neq 1$ ,  $R(2, 1) \wedge R(2, d) \supset 1 = d$  is a neutralizing formula, with  $R(2, d)$  as neutralizer.

Our algorithm to enlarge the search bound consists of first computing a non-empty set  $V$  of neutralizers for literals in  $\mathcal{D}_{S'}(T)$  such that  $V \subseteq S$  and  $V \cap S' = \emptyset$ . Then,  $V \cup S'$  is used as the next search bound. Due to the following proposition, this strategy yields a complete algorithm to compute revisions.

---

**Proposition 1.** *Let  $S' \subseteq S$  and let  $V$  be the set of all neutralizers for all literals in  $\mathcal{D}_{S'}(T)$ . If there is no revision for  $\langle T, M, C, S' \rangle$  and  $(V \cap S) \subseteq S'$  then there is no revision for  $\langle T, M, C, S \rangle$ .*

In the above, we did not specify how to choose the set of neutralizers to enlarge the search bound. Several heuristics can be thought of. A thorough theoretical and experimental investigation of heuristics is part of future work. We implemented the following simple heuristics, but none of them yielded significantly better results than making random choices:

- In order to neutralize multiple dangerous literals at once, prefer to add atoms to  $S'$  that are neutralizer for more than one literal in  $\mathcal{D}_{S'}(T)$ .
- Minimize the negative influence in the next iteration by preferring to add literals that are dangerous for only few subformulas of  $T$ .
- A weighted sum of the two heuristics above.

In the experiments of the next section, all results are obtained with random heuristics.

## 6 Implementation and Experiments

In this section, we report on a prototype implementation of the presented revision algorithm. We present experimental results on three benchmark problems.

### 6.1 FO-Implementation

We made an implementation of the revision algorithm on top of the model generation system IDP. The IDP system uses GIDL [16] as grounder and MINISAT(ID) [8] as propositional solver. The latter is an adaption of the SAT solver MiniSAT [3]. Currently, no heuristics are implemented in our system. At each iteration, a random subset  $V$  of the neutralizers for literals in  $\mathcal{D}_{S'}(T)$  is added to the search bound. The cardinality of  $V$  was restricted to a maximum of 10 atoms in each of the experiments below.

We tested the implementation on three different problems:

**$N$ -Rooks:** The  $N$ -Rooks puzzle as described in Example 1. For each of the instances,  $C$  contains three atoms. I.e., at least three atoms swap truth value compared to the given model.

**$N$ -Queens:** The classical  $N$ -Queens puzzle.  $C$  contains exactly one atom for each of the instances.

**$M \times N$ -Queens:** An adaption of the  $N$ -Queens puzzle. Here,  $M$  chess-boards of dimension  $N \times N$  are placed in a circle. On each board, a valid  $N$ -Queens solution is computed. Moreover, if a board contains a queen on square  $(x, y)$ , then the neighbouring boards may not contain a queen on  $(x, y)$ .  $C$  contains one atom for each of the instances. In this case, it often suffices to revise one of the boards to revise the whole problem.

In the tables below, *MG* stands for *Model generation*. Times and sizes in columns marked *MG* denote times and sizes obtained when solving the problem from scratch with the IDP system. *MR* stands for *Model revision*. Numbers in these columns are obtained with our system. All sizes are expressed in the number of propositional literals. The grounding size for model revision is the grounding size for the final search bound. The *full bound size* is the size of the entire search space, the *final bound size* is the average size of the first search bound containing a revision. *Iter* denotes the average number of iterations, *Changes* the average number of changes with respect to the original model. We used a time-out (###) of 600 seconds. All results are averaged over 10 runs.

On the *N*-rooks problems, the revision algorithm scores well. Up to dimension 5000 is solved within 30 seconds, while model generation was impossible for dimensions above 500. Around 3 iterations are needed to find a sufficiently large search bound. This is as expected, since this problem is not strongly constrained: it is possible to find revisions of low cardinality. On average, 5 rooks were moved to obtain a revised model.

The results on the *N*-Queens show the weakness of the random heuristic on strongly constrained problems. Model revision turns out to be far slower than generating a new model from scratch. Also, many queens are moved to obtain the revised model. E.g., for dimension 40 and 50, 15 queens are moved.

The  $M \times N$ -queens problems show the strength of the revision algorithm. For problems consisting of a large number of small subproblems, the algorithm is able to revise an affected subproblem without looking at the other subproblems. On large problems, this results in far better times compared to model generation.

**Table 1.** *N*-Rooks

	Time (sec.)		Grounding size		Bound size		Iter.	Changes
	MG	MR	MG	MR	Full	Final		
10	0	1	$3.8 \cdot 10^3$	585	100	28	2.8	10.0
50	0	1	$5.0 \cdot 10^5$	479	2500	24	2.4	10.4
100	2	1	$3.9 \cdot 10^6$	698	10000	28	2.8	10.4
250	30	1	$6.2 \cdot 10^7$	903	62500	29	2.9	10.6
500	###	1	$5.0 \cdot 10^8$	1116	$2.5 \cdot 10^5$	28	2.8	10.4
1000	###	2	###	1998	$1.0 \cdot 10^6$	28	2.8	11.0
5000	###	29	###	5575	$2.5 \cdot 10^7$	26	2.6	11.0

## 7 Related Work

Most literature on revision of solutions focusses on particular applications such as train rescheduling. We are not aware of papers describing techniques to handle the general revision problem for FO described in this paper. In work dealing with propositional logic, e.g. [5], the heuristics take large parts of the propositional

**Table 2.**  $N$ -Queens

	Time (sec.)		Grounding size		Bound size		Iter.	Changes
	MG	MR	MG	MR	Full	Final		
10	0	1	3140	1943	100	83	8.3	14
20	0	2	25880	6133	400	93	9.3	14
30	0	7	88220	16690	900	121	12.1	26
40	1	12	210160	30083	1600	135	13.5	30
50	2	24	411700	49317	2500	149	14.9	30
60	4	###	712840	###	3600	###	###	###

**Table 3.**  $M \times N$ -Queens

	Time (sec.)		Grounding size		Bound size		Iter.	Changes
	MG	MR	MG	MR	Full	Final		
$10 \times 10$	0	2	33400	4387	1000	95	10.1	14
$100 \times 10$	1	5	334000	5932	10000	127	13.3	17
$1000 \times 10$	###	29	3340000	5932	100000	127	14.6	17
$10 \times 25$	1	58	521000	34872	6250	408	41.4	25
$100 \times 25$	178	104	5210000	39540	62500	475	48.1	27
$1000 \times 25$	###	277	52100000	39540	625000	475	44.6	27

theory into account, which becomes infeasible for problems with large domains. In the area of Answer Set Programming (ASP), [1] presents a method for updating answer sets of a logic program when a rule is added to it. The method is completely different from the one we presented: it does not rely on existing ASP or SAT solvers, it cannot handle the case where a rule is removed from the program and it works on the propositional level.

Instead of using our method with a DPLL based SAT solver as back-end, one could directly use a local search SAT solver [15] on the grounding of  $\langle T, M, C, S \rangle$  and provide the original model  $M$  as starting point for the search. The main difference with our approach is the way conflicts (dangerous literals) are handled. A local search solver immediately swaps the truth value of neutralizers and hence maintains a two-valued structure, while our approach implicitly assigns the truth value *unknown* to all atoms in the search bound and hence maintains a three-valued structure.

## 8 Conclusions and Future Work

We defined the model revision problem for FO and described an algorithm to solve it. Our presentation leaves much freedom to experiment with different heuristics. A prototype implementation of the algorithm produced promising results, even with random choice heuristics.

The following are topics for future work:

- 
- A thorough theoretical and experimental study of heuristics. A promising approach is to analyze the proof of unsatisfiability produced by the SAT-solver on an input  $\langle T, M, C, S' \rangle$ . This analysis then guides the extension of the search bound  $S'$ . Also heuristics applied in local search SAT and constraint solvers could be of use. A different approach consists of developing a more interactive system where the user can guide the search.
  - The current implementation spends most of its time in grounding. For two search bounds  $S' \subseteq S''$ , the grounding for  $\langle T, M, C, S' \rangle$  is a subtheory of the grounding for  $\langle T, M, C, S'' \rangle$ . As such the same subtheory is grounded multiple times. It is part of future work on the implementation to avoid this overhead by grounding in an incremental way.
  - Many search and revision problems cannot be expressed in a natural manner using an FO theory. E.g., this is the case for problems involving reachability. Therefore, the input language of the IDP system [16] extends FO with constructs to express inductive definitions, aggregates, types, etc. We plan to extend the revision algorithm to this richer language.

## References

1. Martin Brain, Richard Watson, and Marina De Vos. An interactive approach to answer set programming. In *Answer Set Programming*, volume 142 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.
2. Marco Cadoli, Giovambattista Ianni, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile. NP-SPEC: an executable specification language for solving all problems in NP. *Comput. Lang.*, 26(2-4):165–195, 2000.
3. Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
4. Herbert B. Enderton. *A Mathematical Introduction To Logic*. Academic Press, 1972.
5. Maria Fox, Alfonso Gerevini, Derek Long, and Ivan Serina. Plan stability: Replanning versus plan repair. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *ICAPS*, pages 212–221. AAAI, 2006.
6. Chu Min Li, Felip Manyà, and Jordi Planes. New inference rules for max-sat. *J. Artif. Intell. Res. (JAIR)*, 30:321–359, 2007.
7. Victor W. Marek and Mirek Truszczyński. Stable models and an alternative logic programming paradigm. In K.R. Apt, V. Marek, M. Truszczyński, and D.S. Warren, editors, *The Logic Programming Paradigm: a 25 Years Perspective*, pages pp. 375–398. Springer-Verlag, 1999.
8. Maarten Mariën, Johan Wittocx, Marc Denecker, and Bruynooghe Maurice. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In *Proceedings of the 11th conference on Theory and Applications of Satisfiability Testing, SAT 2008*, volume 4996 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2008.
9. David Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In *AAAI'05*, pages 430–435. AAAI Press/MIT Press, 2005.
10. David G. Mitchell. A SAT-solver primer. *Bulletin of the EATCS*, 85:112–132, 2005.

- 
11. Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
  12. Murray Patterson, Yongmei Liu, Eugenia Ternovska, and Arvind Gupta. Grounding for model expansion in k-guarded formulas with inductive definitions. In Manuela M. Veloso, editor, *IJCAI*, pages 161–166, 2007.
  13. Simona Perri, Francesco Scarcello, Gelsomina Catalano, and Nicola Leone. Enhancing DLV instantiator by backjumping techniques. *Annals of Mathematics and Artificial Intelligence*, 51(2-4):195–228, 2007.
  14. Knot Pipatsrisawat and Adnan Darwiche. Clone: Solving weighted max-sat in a reduced search space. In Mehmet A. Orgun and John Thornton, editors, *Australian Conference on Artificial Intelligence*, volume 4830 of *Lecture Notes in Computer Science*, pages 223–233. Springer, 2007.
  15. Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532, 1993.
  16. Johan Wittocx, Maarten Mariën, and Marc Denecker. GIDL: A grounder for  $\text{FO}^+$ . In Michael Thielscher and Maurice Pagnucco, editors, *NMR'08*, pages 189–198, 2008.

---



---

# ISTO: a Language for Temporal Organisational Information Systems

Vitor Nogueira and Salvador Abreu

Universidade de Évora and CENTRIA, Portugal  
{vbn,spa}@di.uevora.pt

**Abstract** In this paper we propose to extend the logical framework ISCO (Information System COnstruction language) with an expressive means of representing and implicitly using temporal information. Moreover, we also provide a compilation scheme that targets a logic language with modularity and temporal reasoning.

## 1 Introduction and Motivation

Organisational Information Systems (OIS) have a lot to benefit from Logic Programming (LP) characteristics such as a rapid prototyping ability, the relative simplicity of program development and maintenance, the declarative reading which facilitates both development and the understanding of existing code, the built-in solution-space search mechanism, the close semantic link with relational databases, just to name a few. In [Por03,ADN04] we find examples of LP languages that were used to develop and maintain Organisational Information Systems.

ISCO (Information System COnstruction language) [Abr01] is based on a Constraint Logic Programming framework to define the schema, represent data, access heterogeneous data sources and perform arbitrary computations. In ISCO, processes and data are structured as *classes* which are represented as typed<sup>1</sup> Prolog predicates. An ISCO class may map to an external data source or sink like a table or view in a relational database, or be entirely implemented as a regular Prolog predicate. Operations pertaining to ISCO classes include a *query* which is similar to a Prolog call as well as three forms of *update*.

In this paper we propose to extend ISCO with an expressive means of representing and implicitly using temporal information. This evolution relies upon the formalism called Temporal Contextual Logic Programming [NA07b,NA07a]. Moreover, having simplicity (syntactic, semantic, etc) as a guideline we present a revised and stripped-down version of ISCO, keeping just some of the core features that we consider indispensable in a language for Temporal Organisational Information Systems. Leaving out aspects such as access control [Abr02] does not mean they are deprecated, only not essential for the purpose at hand. The

---

<sup>1</sup> The type system applies to class members, which are viewed as Prolog predicate arguments.

---

main motivation of this paper is to provide a language suited to construct and maintain Temporal OIS where contexts aren't explicit but implicit (they are obtained throughout the compilation of this language).

The remainder of this article is structured as follows: Sect. 2 briefly overviews Temporal Contextual Logic Programming. Section 3 presents a revised and stripped-down version of the logical framework ISCO and Sect. 4 proposes the temporal extension. Section 5 discusses a compilation scheme for this language and Section 6 compares it with other approaches. Finally, Sect. 7 draws some conclusions.

## 2 An Overview of Temporal Contextual Logic Programming

In this section we present an overview of Temporal Contextual Logic Programming (TCxLP). For a more detailed description please consider [NA07b,NA07a]. Since TCxLP combines the modular language Contextual Logic Programming with the temporal paradigm Temporal Annotated Constraint Logic Programming, we begin by briefly overviews these formalisms.

### 2.1 Contextual Logic Programming

Contextual Logic Programming (CxLP) [MP93] is a simple yet powerful language that extends logic programming with mechanisms for modularity. In CxLP a finite set of Horn clauses with a given name is designated by *unit*. Using the syntax of GNU Prolog/CX (recent implementation for CxLP [AD03]) consider a unit named `employee` to represent some basic facts about university employees:

```
:-unit(employee(NAME, POSITION)).

item :- employee(NAME, POSITION).
employee(bill, teaching_assistant).
employee(joe, associate_professor).

name(NAME).
position(POSITION).
```

The main difference between the example above and a plain logic program is the first line that declares the unit name (`employee`) along with the unit arguments (`NAME, POSITION`). Unit arguments help avoid the annoying proliferation of predicate arguments, which occur whenever a global structure needs to be passed around. A unit argument can be interpreted as a “unit global” variable, i.e. one which is shared by all clauses defined in the unit. Therefore, as soon as a unit argument gets instantiated, all the occurrences of that variable in the unit are replaced accordingly.

Suppose another unit to calculate the salary:

---

```

:-unit(salary(SALARY)).

item :-
    position(P),
    index(P, I),
    base_salary(B),
    SALARY is I*B.

index(teaching_assistant, 12).
index(associate_professor, 20).

base_salary(100).

```

A set of units is designated as a *contextual logic program*. With the units above we can build the program  $P = \{\text{employee}, \text{salary}\}$ . Moreover, if we consider that **employee** and **salary** designate sets of clauses, then the resulting program is given by the union of these sets.

For a given CxLP program, we can impose an order on its units, leading to the (run time) notion of *context*. Contexts are implemented as lists of unit designators and each computation has a notion of its *current context*. The program denoted by a particular context is the union of the predicates that are defined in each unit. Moreover, we resort to the *override semantics* to deal with multiple occurrences of a given predicate: only the topmost definition is visible.

To construct contexts, we have the *context extension* operation: the goal  $U :> G$  extends the *current context* with unit  $U$  and resolves goal  $G$  in the new context. As an illustration, consider a goal that computes Joe's salary:

```
?- employee(joe, P) :> (item, salary(S) :> item).
```

A brief explanation of this goal is as follows: we extend the (initially) empty context with unit **employee** obtaining context  $[\text{employee}(\text{joe}, P)]$  and then resolve query **item**. This leads to  $P$  being instantiated with **associate\_professor**. After **salary/1** being added, the context becomes  $[\text{salary}(S), \text{employee}(\text{joe}, \text{associate\_professor})]$ . Afterwards the second **item** is evaluated and the first matching definition is found in unit **salary**. Goal **position(P)** is called and since there is no rule for this goal in the current unit (**salary**), a search in the context is performed. Since **employee** is the topmost unit that has a rule for **position(P)**, this goal is resolved in the (reduced) context  $[\text{employee}(\text{joe}, \text{associate\_professor})]$ . In an informal way, we queried the context for the position of whom we want to calculate the salary, obtaining **associate\_professor**. The remaining is straightforward, leading to the answer  $S = 2000$  and  $P = \text{associate\_professor}$ .

## 2.2 Temporal Annotated Constraint Logic Programming

This section presents a brief overview of Temporal Annotated Constraint Logic Programming (TACLP). For a more detailed explanation see for instance [Frü96].

---

We consider the subset of TACLP where time points are totally ordered, sets of time points are convex and non-empty, and only atomic formulae can be annotated. Moreover clauses are free of negation.

Time can be discrete or dense. Time points are totally ordered by the relation  $\leq$ . We call the set of time points  $D$  and suppose that a set of operations (such as the binary operations  $+$ ,  $-$ ) to manage such points is associated with it. We assume that the time-line is left-bounded by the number 0 and open the future ( $\infty$ ). A *time period* is an interval  $[r, s]$  with  $0 \leq r \leq s \leq \infty$ ,  $r \in D$ ,  $s \in D$  and represents the convex, non-empty set of time points  $\{t \mid r \leq t \leq s\}$ . Therefore the interval  $[0, \infty]$  denotes the whole time line.

In TACLP there are the following *annotated formulae*:

- **A at t** means that  $A$  holds at time point  $t$ .
- **A th I** means that  $A$  holds *throughout*  $I$ , i.e. at *every* time point in the period  $I$ .
- **A in I** means that  $A$  holds at *some* time point(s) in the time period  $I$ , but there is no knowledge when exactly.

The set of annotations is endowed with a partial order relation  $\sqsubseteq$  which turns into a lattice. Given two annotations  $\alpha$  and  $\beta$ , the intuition is that  $\alpha \sqsubseteq \beta$  if  $\alpha$  is “less informative” than  $\beta$  in the sense that for all formulae  $A$ ,  $A\beta \Rightarrow A\alpha$ . Assuming  $r_1 \leq s_1$ ,  $s_1 \leq s_2$  and  $s_2 \leq r_2$ , we can summarise the axioms for the lattice operation  $\sqsubseteq$  by:

$$in[r_1, r_2] \sqsubseteq in[s_1, s_2] \sqsubseteq in[s_1, s_1] = at\ s_1 = th[s_1, s_1] \sqsubseteq th[s_1, s_2] \sqsubseteq th[r_1, r_2]$$

A TACLP *program* is a finite set of TACLP clauses. A TACLP *clause* is a formula of the form  $A\alpha \leftarrow C_1, \dots, C_n, B_1\alpha_1, \dots, B_m\alpha_m$  ( $m, n \geq 0$ ) where  $A$  is an atom,  $\alpha$  and  $\alpha_i$  are optional temporal annotations, the  $C_j$ ’s are the constraints and the  $B_i$ ’s are the atomic formulae.

### 2.3 Temporal Contextual Logic Programming

The basic mechanism of CxLP is called *context search* and can be described as follows: to solve a goal  $G$  in a context  $C$ , a search is performed until the topmost unit of  $C$  that contains clauses for the predicate of  $G$  is found. Temporal Contextual Logic Programming (TCxLP) incorporates temporal reasoning into this mechanism. To accomplish that, temporal annotations are added not only the dynamic part of CxLP (*contexts*) but also to the static one (*units*) and it will be the relation between those two types of annotations that will determine if a given unit is eligible to match a goal during a context search. Although TCxLP has an operational semantics that provides a complete formalisation of that behavior, for space reasons here we present only an illustrative example. For further reading please consider [NA07b, NA07a].

Revisiting the University employees example, unit **employee** with temporal information can be written as:

---

```

:- unit(employee(NAME, POSITION)).

item.
employee(bill, teaching_assistant) th [2004, inf].
employee(joe, teaching_assistant) th [2002, 2006].
employee(joe, associate_professor) th [2007, inf].

name(NAME).
position(POSITION).

```

This way it is possible to represent the *history* of the employees positions: Joe was a teaching assistant between 2002 and 2006. The same person is a associate professor since 2007. Moreover, in this case the rule for predicate `item/0` doesn't need to be "`item :- employee(NAME, POSITION).`" because the goal `item` is true only if the unit is (temporally) eligible and, for that to happen, the unit arguments must be instantiated. To better understand this example, consider the goal that queries Joe's position throughout [2005,2006]:

```
?- [employee(joe, P)] th [2005,2006] :< item.
```

In the goal above we introduced a new operator (`:<`) called *context switch* where `C :< G` allows the execution of goal `G` in the temporal annotated context `C`. For the example above `item` is true as long as the unit is eligible in the current context, and this happens when `P` is instantiated with `teaching_assistant`, therefore we get the answer `P = teaching_assistant`.

Moreover, if there is no explicit temporal reference in the context, TCxLP assumes the current time. For instance, the following goal queries Joe's current salary:

```
?- employee(joe, P) :> (item, salary(S) :> item).
```

As the reader might see, the goal above is identical to one used to illustrate CxLP. The difference is that units `employee` and `salary` must be eligible in the temporal context defined. Since there is no mention of time we assume the current time (`at 2009`). Therefore in order for those two units to be eligible we obtain `P = associate_professor` and `S = 20000`.

For simplicity reasons we considered that unit `salary` is atemporal and therefore `base_salary` and `index` are the same throughout time. Nevertheless, it should be clear that it would reasonable to define new temporal units `base_salary` and `index`.

### 3 Revising the ISCO Programming Language

In this section we present a revised and stripped-down version of ISCO focusing on what we consider the required features in a language to construct and maintain (Temporal) Organisational Information Systems.

---

### 3.1 Classes

When dealing with large amounts of data, one must be able to organise information in a modular way. The ISCO proposal for this point are *classes*. Although ISCO classes can be regarded as equivalent to Prolog predicates, they provide the homonym OO concept, along with the related data encapsulation capability.

Let us see an ISCO class that handles data about persons, namely its name and social security number:

*Example 1.* Class Person.

```
class person.  
    name: text.  
    ssn: int.
```

In this example after defining the class name to be **person**, we state its arguments and types: **name** type **text** and **ssn** (Social Security Number) type **int**(eger).

### 3.2 Methods

By defining class arguments we are also implicitly setting class methods for accessing those arguments. For instance, in class **person** we have the predicates **ssn/1** and **name/1**. Therefore to query Joe's ssn, besides the (positional) Prolog goal `?- person(joe, S).` we can equivalently use `?- person(name(joe), ssn(N)).`

Besides these implicit methods, there can also be explicit ones defined by means of regular Horn clauses. As an illustration of an Horn clause, consider that argument **name** of class **person** is an atom with the structure syntax 'SURNAME FIRST\_NAME'. In order to obtain the person surname we may add the following clause to the class definition:

```
surname(Surname) :-  
    name(Name),  
    atom_chars(Name, N_Chars),  
    append(Surname_Chars, [' ' | _], N_Chars),  
    atom_chars(Surname, Surname_Chars).
```

### 3.3 Inheritance

*Inheritance* is another Object Oriented feature of ISCO that we would like to retain in our language. The reasons for that are quite natural since it allow us to share not only methods, but also data among different classes.

Consider class **person** of Example 1 and that we want to represent some facts (name, social security number and position) about the employees of a given company. Therefore, we define **employee** to be a subclass of **person** with the argument **position**:

---

*Example 2.* Class `employee`.

```
class employee: person.  
    position: text.
```

### 3.4 Composition

In order to have the OO feature of *composition* we retain the ISCO ability to re-use a class definition as a data type. As an illustration suppose that we want to deal with the employees home address and have the possibility of using the address schema in other classes (e.g. for suppliers). For that we define a new class `address` and re-define the `employee` class adding a new argument (`home`) whose type is `address`:

*Example 3.* Class `address`.

```
class address.                class employee: person.  
    street: text.              home: address.  
    number: int.              position: text.
```

The access to compound types is quite natural, for instance suppose that we want to know Joe's street name:

```
?- employee(name(joe), home(address(street(S)))).
```

Actually, as the reader might see, there is no real need for the basic types since we could develop one class for each. Nevertheless, because they are quite intuitive to use, we decided to keep them.

### 3.5 Persistence

Having persistence in a Logic Programming language is a required feature to construct actual OIS; this could conceivably be provided by Prolog's internal database but is best accounted for by software designed to handle large quantities of factual information efficiently, as is the case in relational database management systems. The semantic proximity between relational database query languages and logic programming languages have made the former privileged candidates to provide Prolog with persistence.

ISCO's approach for interfacing to existing RDBMS<sup>2</sup> involves providing declarations for an external database together with defining equivalences between classes and database relations.

As an illustration, consider that the `employee` facts are stored in a homonym table of a PostgreSQL database named `db` running on the `localhost`:

---

<sup>2</sup> ISCO access to frameworks beyond relational databases, such as LDAP directory services or web services is out of scope for the present work. We shall stick with to RDBMS only.

---

*Example 4.* Class employee with persistence.

```
external(db_link, postgres(db, localhost)).

external(db_link, employee) class employee: person.
    home: address.
    position: text.
```

Since the database table has the same name as the class, the `employee` inside the `external` term above is optional.

### 3.6 Data Manipulation Goals

Under the data manipulation operations we include not only insertion, removal and update but also query operations. Since queries were already subject of previous examples in this section we present only the remaining operations.

The modification goals (insert, delete and update) are based on simple queries, non backtrackable and all follow the tuple-at-a-time approach. As an example, suppose that we want to insert the employee Joe, update his address and then remove this employee:

```
?- employee(name(joe), ssn(111),
            position(associate_professor),
            home(address(street(up), number(1)))) +.

?- employee(name(joe)) #
   (home(address(street(down), number(2)))).

?- employee(name(joe)) -.
```

## 4 The ISTO Language

In this section we provide the revised ISCO language of the previous section with the ability to represent and implicitly use temporal information. This temporal evolution of the ISCO language will be called *ISTO* (Information System Temporal cOnstruction language). The capability of representing temporal information will be achieved by extending ISCO classes to their temporal counterpart. Moreover, in order to handle such temporal information ISTO also includes temporal data manipulation operations.

### 4.1 Temporal Classes

Temporal classes are syntactically introduced by adding the keyword `temporal` before the keyword `class`.

Facts of a temporal class have a temporal dimension, i.e. all tuples have an associated *temporal stamp* which represents instants or intervals (their precise



---

definition will be given in Sect. 4.2). As an illustration, class `employee` can be temporal (Example 3), since it makes sense that the position and home address of a given employee evolves throughout time. On the other hand class `person` should continue atemporal since the facts it stores shouldn't evolve over time.

*Example 5.* Temporal class `employee`

```
temporal class employee: person.
    home: address.
    salary: int.
```

## 4.2 Temporal Data Manipulation

In this section we present the temporal equivalent of the data manipulation goals described in Sect. 3.6, i.e. we provide a temporal query, insertion, removal and update goals.

Keeping the syntax as simple as possible, temporal operations add a suffix composed by the operator “@” and a temporal annotation to the *atemporal* counterparts. The interval representation is the standard one  $[T1, T2]$  and stands for all time points between  $T1$  and  $T2$  (where  $T2 \geq T1$ ). Moreover, in order to be able to handle in-annotated information (see Sect. 2.2), we also allow another type of intervals  $[T1; T2]$  to represent some time points (not necessarily all) between  $T1$  and  $T2$ , i.e.  $[T1; T2]$  is a subset of  $[T1, T2]$ .

Although an instant  $T$  can be represented by the interval  $[T, T]$ , to ease the reading we denote it simply by  $T$ . As an illustration consider that we want to know Joe's street name in the year 2007:

*Example 6.* Joe's street name in the year 2007

```
?- employee(name(joe),
            home(address(street(S)))) @ 2007
S = upstreet
```

The goal above is a temporal version of the one presented in Sect. 3.4.

## 5 Compilation Scheme for ISTO

The compilation of the non-temporal part of ISTO yields CxLP and from the temporal extension we target TCxLP. Due to the space limitations and since the temporal aspects are the novelty, in this section we describe (essentially) the compilation scheme for these aspects.

### 5.1 (Temporal) Classes

The translation from an ISTO class to CxLP can be roughly described as: every class is mapped to a corresponding unit, with the class arguments transformed

---

into unit arguments and predicates for accessing these arguments. Moreover, one extra unit argument `OID` standing for *Object Identifier* is added to every unit and is used to discriminate a given element in a class.

Before presenting the compilation of temporal classes and goals one must observe that non-temporal classes must behave as if they were valid throughout the entire time line. Such a behavior can be obtained simply by adding a fact to each nontemporal unit. For instance, to the unit `person` which has been presented, it suffices to add the fact: `person(_OID, _NAME, _SSN) th [0, inf]`.

Let us now see the result of compiling the temporal class `employee` from Example 5:

```
:- unit(employee(OID, HOME_OID, POSITION)).

oid(OID).

home(address(GOAL)) :-
    [address(HOME_OID, STREET, NUMBER)]
    :< (item, GOAL).

position(POSITION).

item :- :^ item.
```

The main difference from the compilation of the nontemporal class is that there is no need for predicate `item` to instantiate the unit arguments, since the temporal context search will do that implicitly. Besides the clauses above there must be also some temporal conditions such as `employee(1, 1, associate_professor) th [2007, inf]` (representing for instance part of Joe's information.) Finally, since `employee` is a subclass of `person`, the context for unit `employee` is of the form `[employee(...), person(...), ...]`, therefore to ensure consistency `item/0` must also be invoked in the context `[person(...), ...]` and that is accomplished by `:^ item`.<sup>3</sup>

## 5.2 Temporal Data Manipulation Goals

The translation of a temporal query will result in a goal in a temporal context. The ISTO query of Joe's street name in 2007 of Sect. 4.2 is translated into:

```
?- [employee(OID, HOME_OID, SALARY),
    person(OID, NAME, SSN)] (at 2007) :<
    (item, name(joe), home(address(street(S))))).
```

Introducing temporal modification goals needs further considerations. First of all, as mentioned, these goals are extra-logical. Moreover, since now the th-annotated facts can change at runtime, to use the (simplified) semantics of TCxLP one must ensure that the backend of a temporal class always stores

---

<sup>3</sup> The operator `:^` is called the supercontext operator.

---

the least upper bound of th-annotated unit temporal conditions. In order to guarantee that, every insertion of a th-annotated temporal condition must induce a recomputation of the least upper bound. As an illustration consider again Joe's positions through time:<sup>4</sup>

```
employee(1, teaching_assistant) th [2002, 2006].
employee(1, associate_professor) th [2007, inf].
```

Suppose the following ISTO goal to remove this employee information between 2005 and 2006:

```
?- (employee(name(joe)) -) @ [2005, 2006].
```

It changes the temporal conditions in the following way:

```
employee(1, teaching_assistant) th [2002, 2004].
employee(1, associate_professor) th [2007, inf].
```

Finally, if we add the information that Joe's position between 2005 and 2006 was `associate_professor`:

```
?- (employee(name(joe),
    position(associate_professor)) +) @ [2005, 2006].
```

then the least upper bound must be recomputed, leading to:

```
employee(1, teaching_assistant) th [2002, 2004].
employee(1, associate_professor) th [2005, inf].
```

## 6 Comparison with other Approaches

The temporal timestamps of ISTO can be regarded as the counterpart of the *valid time* proposed in temporal databases [TCG<sup>+</sup>93]. Although ISTO has no concept similar to the *transaction time*, we consider that one could implement it by adding a log in the ISTO backend unit. However this capability is beyond the scope of the ISTO initial concerns. On the other hand, ISTO *contextual time* enables an expressiveness that lacks in most database products with temporal support. Only in Oracle Workspace Manager [Cor05] we find a concept (workspace) that is related to our temporal context.

As far as we know there are logical languages that have modularity/OO, others that provide temporal reasoning or even persistence. ISTO is the only one that encompasses all those features.

## 7 Conclusions and Future Work

In this paper we presented a revision of a state-of-the-art logical framework for constructing OIS called ISCO. We proceeded to introduce an extension to this

---

<sup>4</sup> For simplicity reasons in this illustration we ignore the home address argument.

---

language called ISTO, that includes the expressive means of representing and implicitly using temporal information. Together with a syntactical definition of ISTO we also presented a compilation scheme from this language into Temporal Contextual Logic Programming.

Finally, we consider that ISTO can take advantage of the experience gained from several real-world application developed in ISCO in order to act as backbone for constructing and maintaining Temporal OIS. Therefore we are currently applying the language proposed in an University Information System.

## References

- Abr01. Salvador Abreu. Isco: A practical language for heterogeneous information system construction. In *Proceedings of INAP'01*, Tokyo, Japan, October 2001. Prolog Association of Japan.
- Abr02. Salvador Abreu. Modeling Role-Based Access Control in ISCO. In Lgia Maria Ribeiro and Jos Marques dos Santos, editors, *The 8th International Conference of European University Information Systems*. FEUP Edies, June 2002. ISBN 972-752-051-0.
- AD03. Salvador Abreu and Daniel Diaz. Objective: In minimum context. In Catuscia Palamidessi, editor, *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 128–147. Springer, 2003.
- ADN04. Salvador Abreu, Daniel Diaz, and Vitor Nogueira. Organizational information systems design and implementation with contextual constraint logic programming. In *IT Innovation in a Changing World – The 10<sup>th</sup> International Conference of European University Information Systems*, Ljubljana, Slovenia, June 2004.
- Cor05. Oracle Corporation. Oracle database 10g workspace manager overview. Oracle White Paper, May 2005.
- Frü96. Thom W. Frühwirth. Temporal annotated constraint logic programming. *J. Symb. Comput.*, 22(5/6):555–583, 1996.
- MP93. Luís Monteiro and António Porto. A Language for Contextual Logic Programming. In K.R. Apt, J.W. de Bakker, and J.J.M.M. Rutten, editors, *Logic Programming Languages: Constraints, Functions and Objects*, pages 115–147. MIT Press, 1993.
- NA07a. Vitor Nogueira and Salvador Abreu. Temporal Annotations for a Contextual Logic Programming Language. In José Neves, Manuel Santos, and José Machado, editors, *Progress in Artificial Intelligence, 13th Portuguese Conference on Artificial Intelligence, EPIA 2007*, Universidade do Minho, 2007.
- NA07b. Vitor Nogueira and Salvador Abreu. Temporal contextual logic programming. *Electr. Notes Theor. Comput. Sci.*, 177:219–233, 2007.
- Por03. António Porto. An integrated information system powered by prolog. In Verónica Dahl and Philip Wadler, editors, *PADL*, volume 2562 of *Lecture Notes in Computer Science*, pages 92–109. Springer, 2003.
- TCG<sup>+</sup>93. Abdullah Uz Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass, editors. *Temporal databases: theory, design, and implementation*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1993.

---

# Knowledge Representation Using Logtalk Parametric Objects

Paulo Moura

Dep. of Computer Science, University of Beira Interior, Portugal  
Center for Research in Advanced Computing Systems, INESC–Porto, Portugal  
`pmoura@di.ubi.pt`

**Abstract.** This paper describes how Logtalk *parametric objects* can be used for knowledge representation by presenting a comprehensive set of useful programming patterns. A parametric object is an object whose identifier is a compound term containing logical variables. These variables play the role of object parameters. Object predicates can be coded to depend on the parameter values. Parametric objects are a common feature of some other object-oriented logic programming languages and Prolog object-oriented extensions. Logtalk extends the usefulness of parametric objects by introducing the concept of *object proxies*. An object proxy is a compound term that can be interpreted as a possible instantiation of the identifier of a parametric object. Object proxies, when represented as predicate facts, allow application memory footprint to be minimized while still taking full advantage of Logtalk object-oriented features for representing and reasoning with taxonomic knowledge.

**Keywords:** knowledge representation, logic-programming, parametric objects, object proxies, programming patterns.

## 1 Introduction

Logtalk [1–3] is an object-oriented logic programming language that can use most Prolog implementations as a back-end compiler. Logtalk extends Prolog with versatile code encapsulation and code reuse constructs based on an interpretation of object-oriented concepts in the context of logic programming. Logtalk features *objects* (both *prototypes* and *classes*), *static* and *dynamic binding* (with predicate lookup caching), *single* and *multiple inheritance*, *protocols* (aka interfaces; sets of predicate declarations that can be implemented by any object), *categories* (fine-grained units of code reuse that can be used as object building blocks), *event-driven programming*, and *high-level multi-threading programming* (and-parallelism and competitive or-parallelism). Objects, protocols, and categories can be either static or dynamic. The use of static entities and static binding results in performance similar to plain Prolog.

Logtalk includes support for *parametric objects*, a feature common to some object-oriented logic programming languages and Prolog object-oriented extensions. A parametric object is an object whose identifier is a compound term

---

containing logical variables. A simple example of a parametric object for representing two-dimensional geometric points could be:

```
:- object(point(_X, _Y)).  
  
:- end_object.
```

The variables in the object identifier play the role of object parameters. Object predicates can be coded to depend on the parameter values. For example, assuming that we needed to compute the distance of a point to the origin, we could write:

```
:- object(point(_X, _Y)).  
  
:- public(distance/1).  
  
distance(Distance) :-  
    parameter(1, X),  
    parameter(2, Y),  
    Distance is sqrt(X*X + Y*Y).  
  
:- end_object.
```

After compiling this parametric object we could try queries such as:

```
| ?- point(3.0, 4.0)::distance(Distance).  
  
Distance = 5.0  
yes
```

Thus, a parametric object can be regarded as a generic object from which specific *instantiations* can be derived by instantiating the object parameters at runtime. Note, however, that instantiating object parameters does not create new objects. For example, the object identifiers `point(2.0, 1.2)` and `point(3.0, 4.0)` refer to the same parametric object. Parameter instantiation usually takes place when a message is sent to the object. By using logical variables, parameter instantiation is undone on backtracking. Object parameters can be any valid term: logical variables, constraint variables, atomic terms, or compound terms.

Despite the simplicity of the concept, parametric objects proved a valuable asset in a diversity of applications. Parametric objects support several useful programming patterns, presented and illustrated in this paper. The remainder of the paper is organized as follows. Section 2 describes Logtalk built-in methods for accessing object parameters. Section 3 illustrates parameter passing within object hierarchies. Section 4 presents useful parametric object programming patterns, illustrated by examples. Section 5 presents and discusses the concept of *object proxies*, introducing additional programming patterns. Section 6 shows how to use both object proxies and regular objects to represent entities of the same *type* that differ on complexity or size. Section 7 discusses related work. Section 8 presents our conclusions.

---

## 2 Accessing Object Parameters

Logtalk provides a `parameter/2` built-in method for accessing object parameters. The first argument is the parameter position. The second argument is the object parameter. For example:

```
:- object(ellipse(_Rx, _Ry, _Color)).

    area(Area) :-
        parameter(1, Rx),
        parameter(2, Ry),
        Area is Rx*Ry*pi.
...

```

Logtalk compiles the calls to the method `parameter/2` in-line by unifying the second argument with the corresponding object identifier argument in the execution context argument in the translated clause head. A second built-in method, `this/1`, allows access to all object parameters at once:

```
area(Area) :-
    this(ellipse(Rx, Ry, _)),
    Area is Rx*Ry*pi.

```

As with the `parameter/2` method, calls to the `this/1` method are compiled in-line.

An alternative to the `parameter/2` and `this/1` methods for accessing object parameters would be to interpret parameters as logical variables with global scope within the object. This solution was discarded as it forces the programmer to always be aware of parameter names when coding object predicates and it collides with the local scope of Logtalk and Prolog variables in directives and predicate clauses.

## 3 Parameter Passing

Logtalk objects may *extend* (defining parent-prototype relations), *instantiate* (defining instance-class relations), or *specialize* (defining class-superclass relations) other objects. When using parametric objects, parameter passing is established through unification in the object opening directive. As an example, consider the following Logtalk version of a SICStus Objects [4] example:

```
:- object(ellipse(_RX, _RY, _Color)).

    :- public([color/1, rx/1, ry/1, area/1]).

    rx(Rx) :-
        parameter(1, Rx).

    ry(Ry) :-
        parameter(2, Ry).

```

---

```

    color(Color) :-
        parameter(3, Color).

    area(Area) :-
        this(ellipse(Rx, Ry, _)),
        Area is Rx*Ry*pi.

:- end_object.

:- object(circle(Radius, Color),          % circles are ellipses
    extends(ellipse(Radius, Radius, Color))). % where Rx = Ry

:- public(r/1).

r(Radius) :-
    parameter(1, Radius).

:- end_object.

:- object(circle1(Color),
    extends(circle(1, Color))).

:- end_object.

:- object(red_circle(Radius),
    extends(circle(Radius, red))).

:- end_object.

```

A query such as:

```

| ?- red_circle(3)::area(Area).

Area = 28.274334
yes

```

will result in the following instantiation chain of the parametric object identifiers:

```
red_circle(3) -> circle(3, red) -> ellipse(3, 3, red)
```

Note that the predicate `area/1` is declared and defined in the object representing ellipses.

## 4 Programming Patterns

This section presents a comprehensive set of useful parametric object programming patterns. These patterns build upon the basic idea that a parametric object encapsulates a set of predicates for working with compound terms that share the same functor and arity. Additional programming patterns will be described later



---

in this paper when presenting the Logtalk concept of parametric object proxies. The full source code of most examples is included in the current Logtalk distribution.

#### 4.1 Simplifying Object Interfaces

Parametric objects can simplify object interfaces by moving core object properties from predicate arguments to object parameters. This also makes the core properties visible without requiring the definition of *accessor* predicates to retrieve them. Consider the following parametric object representing rectangles:

```
:- object(rectangle(_Width, _Height)).

    :- public(area/1).

    area(Area) :-
        this(rectangle(Width, Height)),
        Area is Width*Height.

    :- public(perimeter/1).

    perimeter(Perimeter) :-
        this(rectangle(Width, Height)),
        Perimeter is 2*(Width + Height).

:- end_object.
```

The rectangle properties *width* and *height* are always accessible. The two rectangle predicates, *area/1* and *perimeter/1*, have a single argument returning the respective computed values.

Assume that our application would also need to compute areas and perimeters of circles, triangles, pentagons, and other (regular or non-regular) polygons. The alternative of using non parametric objects to encapsulate the same functionality would require adding extra arguments to both predicates that would depend on the type of shape. We could use instead a single *data* argument that would accept a compound term representing a shape but that would result in a awkward solution for encapsulating the knowledge about each type of shape in its own object. Another alternative would be to use predicates such as *area\_circle/1* and *perimeter\_triangle/1* but this solution is hard to extend to new shapes, to new predicates over shapes, and would hinder processing of heterogenous collections of shapes. Parametric objects provide us with a simple, clean, logical, and easily extensible knowledge representation solution.

#### 4.2 Data-Centric Programming

Parametric objects can be seen as enabling a more data-centric programming style where data is represented by *instantiations* of parametric object identifiers.

---

Instead of using a term as a predicate argument, predicates can be called by sending the corresponding message to the term itself.

To illustrate this idea we will use the well-known example of symbolic differentiation and simplification of arithmetic expressions, which can be found in [5]. The L&O [6] system also uses this example to illustrate parametric theories. The idea is to represent arithmetic expressions as parametric objects whose name is the expression operator with greater precedence, and whose parameters are the operator sub-expressions (that are, themselves, objects). In order to simplify this example, the object methods will be restricted to symbolic differentiation of polynomials with a single variable and integer coefficients. In addition, we will omit any error-checking code. The symbolic simplification of arithmetic expressions could easily be programmed in a similar way.

For an arithmetic expression reduced to a single variable,  $x$ , we will have the following object:

```
:- object(x,
    implements(diffp)). % protocol (interface) declaring a diff/1
                        % symbolic differentiation predicate
    diff(1).

:- end_object.
```

Arithmetic addition,  $x + y$ , can be represented by the parametric object '+'( $X$ ,  $Y$ ) or, using operator notation,  $X + Y$ . Taking into account that the operands can either be numbers or other arithmetic expressions, a possible definition will be:

```
:- object(_ + _,
    implements(diffp)).

diff(Diff) :-
    this(X + Y),
    diff(X, Y, Diff).

diff(I, J, 0) :-
    integer(I), integer(J), !.

diff(X, J, DX) :-
    integer(J), !, X::diff(DX).

diff(I, Y, DY) :-
    integer(I), !, Y::diff(DY).

diff(X, Y, DX + DY) :-
    X::diff(DX), Y::diff(DY).

:- end_object.
```

The object definitions for other simple arithmetic expressions, such as  $X - Y$  or  $X * Y$ , are similar. The expression  $x^n$  can be represented by the parametric object  $X ** N$  as follows:

---

```

:- object(_ ** _).

:- public(diff/1).

diff(N * X ** N2) :-
    this(X ** N),
    N2 is N - 1.

:- end_object.

```

By defining a suitable parametric object per arithmetic operator, any polynomial expression can be interpreted as an object identifier. For example, the polynomial  $2x^3 + x^2 - 4x$  ( $2x^3 + x^2 - 4x$ ) will be interpreted as  $(2x^3) + (x^2 - 4x)$ . Thus, this expression is an *instantiation* of the  $X + Y$  parametric object identifier, allowing us to write queries such as:

```

| ?- (2*x**3 + x**2 - 4*x)::diff(D).

D = 2* (3*x**2)+2*x**1-4*1
yes

```

The resulting expression could be symbolically simplified using a predicate defined in the same way as the `diff/1` differentiation predicate.

### 4.3 Restoring Shared Constraint Variables

Object parameters can be used to restore shared variables between sets of constraints that are encapsulated in different objects. We illustrate this idea using an example that is loosely based in the declarative process modeling research project ESProNa [7]. Each process can be described by a parametric object, implementing a common protocol (reduced here to a single predicate declaration for the sake of this example):

```

:- protocol(process_description).

:- public(domain/2).
:- mode(domain(-list(object_identifier), -callable), one).
:- info(domain/2, [
    comment is 'Returns the process dependencies and constraints.',
    argnames is ['Dependencies', 'Constraints']]).

% other process description predicates

:- end_protocol.

```

Part of the process description is a set of finite domain constraints describing how many times the process can or must be executed. An object parameter is used to provide access to the process constraint variable. For example, the following process, `a(_)`, can be executed two, three, or four times:

---

```

:- object(a(_),
    implements(process_description)).

    domain([], (A #>= 2, A #=< 4)) :-
        parameter(1, A).

:- end_object.

```

Processes may depend on other processes. These dependency relations can be described using a list of parametric object identifiers. For example:

```

:- object(b(_),
    implements(process_description)).

    domain([a(A)], (B #>= A, B #=< 3)) :-
        parameter(1, B).

:- end_object.

:- object(c(_),
    implements(process_description)).

    domain([a(A), b(B)], (C #= B + 1, C #= A + 1)) :-
        parameter(1, C).

:- end_object.

```

The object parameters allows us to restore shared constraint variables at runtime in order to model sets of inter-dependent processes. For example (using Logtalk with GNU Prolog, with its native finite domain solver, as the back-end compiler):

```

| ?- process_model::solve([c(C)], Dependencies).

C = _#59(3..4)
Dependencies = [b(_#21(2..3)),a(_#2(2..3)),c(_#59(3..4))]
yes

```

The predicate `solve/2` (the code of the `process_model` object is omitted due to the lack of page space) computes this answer by retrieving and solving the conjunction of the constraints of processes `a()`, `b()`, and `c()`.

#### 4.4 Logical Updates of Object State

Parametric objects provide an alternative to represent object dynamic state. This alternative supports logical updates, undone by backtracking, by representing object state using one or more object parameters. A similar solution was first used in the OL(P) system [8]. Logtalk supports this solution using a pure logical subset implementation of *assignable variables* [9], available in the library

---

`assignvars`.<sup>1</sup> This library defines *setter* and *getter* methods (`<=/2` and `=>/2`, respectively) and an initialization method (`assignable/2`) to work with assignable variables.

Consider the following simple example, where a parametric object is used to describe a geometric rectangle. The object state is comprised by its read-only dimensions, *width* and *height*, and by its dynamically updatable *position*:

```
:- object(rectangle(_Width, _Height, _Position),
    imports(private::assignvars)).

:- public([init/0, area/1, move/2, position/2]).

init :-
    parameter(3, Position),
    ::assignable(Position, (0, 0)).

area(Area) :-
    this(rectangle(Width, Height, _)),
    Area is Width*Height.

move(X, Y) :-
    parameter(3, Position),
    ::Position <= (X, Y).

position(X, Y) :-
    parameter(3, Position),
    ::Position => (X, Y).

:- end_object.
```

A sample query could be:

```
| ?- rectangle(2, 3, _)::(init, area(Area), position(X0, Y0),
    move(1, 1), position(X1, Y1), move(2, 2), position(X2, Y2)).

Area = 6
X0 = Y0 = 0
X1 = Y1 = 1
X2 = Y2 = 2
yes
```

Using object parameters for representing mutable state provides a solution that supports clean and logical application semantics, backtracking over state changes, and better performance than using `assert` and `retract` operations to update an object dynamic database.

---

<sup>1</sup> This library is implemented as a *category* [1], a fine-grained unit of code reuse that can be virtually imported by any object without code duplication.

---

## 5 Parametric Object Proxies

Compound terms with the same functor and arity as a parametric object identifier may act as *proxies* to a parametric object. Proxies may be stored on the database as Prolog facts and be used to represent different *instantiations* of a parametric object identifier. This representation can be ideal to minimize application memory requirements in the presence of a large number of data objects whose state is immutable. As a simple example, assume that our data represents geometric circles with attributes such as an identifier, the circle radius and the circle color:

```
% circle(Id, Radius, Color)
circle('#1', 1.23, blue).
circle('#2', 3.71, yellow).
circle('#3', 0.39, green).
circle('#4', 5.74, black).
```

We can define the following parametric object for representing circles:

```
:- object(circle(_Id, _Radius, _Color)).

    :- public([id/1, radius/1, color/1, area/1, perimeter/1]).

    id(Id) :-
        parameter(1, Id).

    radius(Radius) :-
        parameter(2, Radius).

    color(Color) :-
        parameter(3, Color).

    area(Area) :-
        parameter(2, Radius),
        Area is 3.1415927*Radius*Radius.

    perimeter(Perimeter) :-
        parameter(2, Radius),
        Perimeter is 2*3.1415927*Radius.

:- end_object.
```

The `circle/3` parametric object provides a simple solution for encapsulating a set of predicates that perform computations over circle properties, as illustrated by the `area/1` and `perimeter/1` predicates.

Logtalk provides a convenient notational shorthand for accessing proxies represented as Prolog facts when sending a message:

```
| ?- {Proxy}::Message.
```

---

In this case, Logtalk proves `Proxy` as a Prolog goal and sends the message to the resulting term, interpreted as the identifier of a parametric object.<sup>2</sup> This construct can either be used with a proxy argument that is sufficiently instantiated in order to unify with a single Prolog fact or with a proxy argument that unifies with several facts. Backtracking over the proxy goal is supported, allowing all matching object proxies to be processed by e.g. a simple failure-driven loop. For example, in order to construct a list with the areas of all the circles we can write:

```
| ?- findall(Area, {circle(_, _, _)}::area(Area), Areas).

Areas = [4.75291, 43.2412, 0.477836, 103.508]
yes
```

In non-trivial applications, data objects, represented as object proxies, and the corresponding parametric objects, are tied to large hierarchies representing taxonomic knowledge about the application domain. An example is the LgtSTEP application [10] used for validating STEP files, which is a format for sharing data models between CAD/CAM applications. A typical data model can contain hundreds of thousands of geometrical objects, which are described by a set of hierarchies representing geometric concepts, data types, and consistency rules. In its current version, the LgtSTEP application hierarchies define 252 geometric or related entities, 78 global consistency rules, 68 data types, and 69 support functions. STEP files use a simple syntax for geometrical and related objects. Consider the following fragment:

```
#1=CARTESIAN_POINT('', (0.E0,0.E0,-3.38E0));
#2=DIRECTION('', (0.E0,0.E0,1.E0));
#3=DIRECTION('', (1.E0,0.E0,0.E0));
#4=AXIS2_PLACEMENT_3D('', #1, #2, #3);
```

The first line above defines a cartesian point instance, whose identifier is `#1`, followed by an empty comment and the point coordinates in a 3D space. Similar for the other lines. This syntax is easily translated to Prolog predicate facts that are interpreted by Logtalk as object proxies:

```
cartesian_point('#1', '', (0.0, 0.0, -3.38)).
direction('#2', '', (0.0, 0.0, 1.0)).
direction('#3', '', (1.0, 0.0, 0.0)).
axis2_placement_3d('#4', '', '#1', '#2', '#3').
```

Complemented by the corresponding parametric objects, object proxies provide a source level representation solution whose space requirements are roughly equal to those of the original STEP file. An alternative representation using one Logtalk object per STEP object results in space requirements roughly equal to 2.2 times the size of the original STEP file in our experiments.<sup>3</sup> For example, the cartesian point above could be represented by the object:

---

<sup>2</sup> The `{}/1` functor was chosen as it is already used as a control construct to bypass the Logtalk compiler in order to call Prolog predicates.

<sup>3</sup> Logtalk object representation is currently optimized for execution time performance, not memory space requirements.

---

```

:- object('#1',
    instantiates(cartesian_point)).

    comment('').
    coordinates(0.0, 0.0, -3.38).

:- end_object.

```

For simple, immutable data objects, Logtalk object representation provides little benefit other than better readability. Object proxies and parametric objects allows us to avoid using a Logtalk object per data object, providing a bridge between the data objects and the hierarchies representing the knowledge used to reason about the data, while optimizing application memory requirements.

## 6 Using Both Object Proxies and Regular Objects

While object proxies provide a compact representation, complex domain objects are often better represented as regular objects. Moreover, while in object proxies the meaning of an argument is implicitly defined by its position in a compound term, predicates with meaningful names can be used in regular objects. In some applications it is desirable to be able to choose between object proxies and regular objects on a case-by-case basis. An example is the L-FLAT [11] application, a full rewrite in Logtalk of P-FLAT [12], a Prolog toolkit for teaching Formal Languages and Automata Theory. L-FLAT is a work in progress where one of the goals is to allow users to use both object proxies and regular objects when representing Turing machines, regular expressions, finite automata, context free grammars, and other concepts and mechanisms supported by L-FLAT. This flexibility allows users to represent e.g. a simple finite automaton with a small number of states and transitions as an object proxy:

```

% fa(Id, InitialState, Transitions, FinalStates)
fa(fa1, 1, [1/a/1, 1/a/2, 1/b/2, 2/b/2, 2/b/1], [2]).

```

It also allows users to represent e.g. a complex Turing machine with dozens of transitions as a regular object:

```

:- object(aibiciTM,
    instantiates(tm)).

    initial(q0).

    transitions([
        q0/'B'/'B'/'R'/q1,
        q1/'a'/'X'/'R'/q2,    q1/'Y'/'Y'/'R'/q5,    q1/'B'/'B'/'R'/q6,
        q2/'a'/'a'/'R'/q2,    q2/'Y'/'Y'/'R'/q2,    q2/'b'/'Y'/'R'/q3,
        q3/'b'/'b'/'R'/q3,    q3/'Z'/'Z'/'R'/q3,    q3/'c'/'Z'/'L'/q4,
        q4/'a'/'a'/'L'/q4,    q4/'b'/'b'/'L'/q4,    q4/'Y'/'Y'/'L'/q4,
        q4/'Z'/'Z'/'L'/q4,    q4/'X'/'X'/'R'/q1,

```



---

```

        q5/'Y'/'Y'/'R'/q5, q5/'Z'/'Z'/'R'/q5, q5/'B'/'B'/'R'/q6
    ]).

    finals([q6]).

:- end_object.

```

The transparent use of both object proxies and regular objects requires that all predicates expecting e.g. a regular expression object accept both an object identifier and an object proxy as argument. While this may sound as an additional hurdle, the solution is simple. Given that an object proxy is also an instantiation of the identifier of a parametric object, it suffices to define the corresponding parametric object. If, for example, we want to represent some finite automata as instances of a class `fa` and other finite automata as object proxies, we simply define a parametric object as an instance of the class `fa`. The object parameters will be the properties that might be unique for a specific finite automaton. The parametric object define the predicates that give access to these properties. These predicates would be the same used in class `fa` to define all predicates that must access to finite automaton properties:

```

:- object(fa(_Id, _Initial, _Transitions, _Finals),
    instantiates(fa)).

    initial(Initial) :-
        parameter(2, Initial).

    transitions(Transitions) :-
        parameter(3, Transitions).

    finals(Finals) :-
        parameter(4, Finals).

:- end_object.

```

Thus, using both object proxies and regular objects is simple, fully transparent, and just a matter of defining the necessary parametric objects.

## 7 Related Work

Logtalk parametric objects are based on L&O [6] parametric theories and SICStus Objects [4] parametric objects. The three systems provide similar functionality, with the exception of object proxies. One notable difference is parameter accessing. While Logtalk parameter values are accessed using the `parameter/1` and `this/1` built-in methods, the scope of the parameters in L&O and SICStus Objects is the whole object. Logtalk parametric objects are also partially based in the OL(P) system [8] representation of object instances.

---

## 7.1 L&O Parametric Theories

In L&O, parametric objects are known as parametric theories. A theory is identified by a *label*, a term that is either a constant or a compound term with variables. L&O uses as example a parametric theory describing trains:

```
train(S, Cl, Co):{
    colour(Cl).
    speed(S).
    country(Co).
    journey_time(Distance, T) :-
        T = Distance/S.
}
```

The label variables are universally quantified over the theory. A specific train can be described by instantiating the label variables:

```
train(120, green, britain)
```

Messages can be sent to labels, which act as object identifiers. For example, the following message:

```
train(120, green, britain):journey_time(1000, Time)
```

will calculate a journey time using the value of the label first parameter as the speed of the train.

## 7.2 SICStus Parametric Objects

SICStus parametric objects are similar to L&O parametric theories, with parameters acting as global variables for the parametric object. The SICStus Objects manual contains the following example, describing ellipses and circles:

```
ellipse(RX, RY, Color) :: {
    color(Color) &
    area(A) :-
        : (A is RX*RY*3.14159265)
}.

circle(R, Color) :: {
    super(ellipse(R, R, Color))
}.

red_circle(R) :: {
    super(circle(R, red))
}.
```

SICStus Objects uses the predicate `super/1` to declare the ancestors of an object. This example illustrates parameter-passing between related objects in a hierarchy, a feature common to both L&O and Logtalk.

---

### 7.3 OL(P) Object Instances

OL(P) is a Prolog object-oriented extension that represents object instances using a notation similar to parametric objects. An instance `I` of an object named `Object` is represented as `Object(I)`. The term `I` is a list of attributes and attribute-value pairs. Instance state changes can be accomplished by constructing a new list with the updated and unchanged attributes. The OL(P) system documentation offers the following example:

```
| ?- rect(I)::area(A), rect(I)::move(5, 5, J).
```

The method `move/3` will return, in its third argument, the attribute list `J` resulting from the update of the attribute list `I`. In addition, OL(P) provides a nice notation for accessing and updating attributes. This solution for object state changes implies the use of extra arguments for methods that update attributes. Nevertheless, it is an interesting technique, which preserves the declarative semantics found on pure Prolog programs. We can easily apply this solution to Logtalk programs by using parametric objects. Moreover, we are not restricted to using a list of attributes. If the number of attributes is small, an identifier with the format `Object(V1, V2, ..., Vn)` will provide a more efficient solution.

## 8 Conclusions

The main contributions of this paper are a survey of useful parametric object programming patterns and the concept of object proxies.

Parametric objects enable useful programming patterns that complement Logtalk encapsulation and reuse features. Parametric objects may be used to encapsulate a set of predicates for reasoning about compound terms sharing the same functor and arity, to simplify object interfaces, to restore shared variables between sets of constraints stored in different objects, to enable a more data-centric style of programming, and to provide a logical alternative to the use of an object dynamic database for representing mutable state. It is also possible to use the *instantiations* of a parametric object identifier to represent the history of object state changes.

Logtalk extends the usefulness of parametric objects by introducing the concept of object proxies, which provide a bridge between Logtalk objects and compact plain Prolog representations. Some applications need to represent a large number of immutable objects. These objects typically represent data that must be validated or used for data mining. This kind of data is often exported from databases and must be converted into objects for further processing. The application domain logic is usually represented by a set of hierarchies with the data objects at the bottom. Although data conversion is most of the time easily scriptable, the resulting objects take up more space than a straightforward representation as plain Prolog facts. By using object proxies, application memory footprint can be minimized while still taking full advantage of Logtalk object-oriented features for representing and reasoning with taxonomic knowledge. The concept of object proxies can be easily adopted and used in other languages supporting parametric objects.

---

**Acknowledgements.** This work is partially supported by the FCT research project MOGGY – PTDC/EIA/70830/2006. We are grateful to Paul Crocker for helpful suggestions in improving Logtalk support for object proxies. We thank also José Silva, Sara C. Madeira, and the anonymous reviewers for their comments and help in revising this paper.

## References

1. Moura, P.: Logtalk 2.6 Documentation. Technical Report DMI 2000/1, University of Beira Interior, Portugal (July 2000)
2. Moura, P.: Logtalk - Design of an Object-Oriented Logic Programming Language. PhD thesis, Department of Computer Science, University of Beira Interior, Portugal (September 2003)
3. Moura, P.: Logtalk 2.37.2 User and Reference Manuals. (June 2009)
4. Swedish Institute for Computer Science: SICStus Prolog 4.0 User Manual. (April 2009)
5. Clocksin, W.F., Mellish, C.S.: Programming in Prolog. Springer-Verlag, New York (1987)
6. McCabe, F.G.: Logic and Objects. Series in Computer Science. Prentice Hall (1992)
7. Igler, M., Joblonski, S.: ESProNa – Engine for Semantic Process Navigation. <http://www.ai4.uni-bayreuth.de/> (2009)
8. Fromherz, M.: OL(P): Object Layer for Prolog. <ftp://parcftp.xerox.com/ftp/pub/ol/> (1993)
9. Kino, N.: Logical assignment of Prolog terms. [http://www.kprolog.com/en/logical\\_assignment/](http://www.kprolog.com/en/logical_assignment/) (2005)
10. Moura, P., Marchetti, V.: Logtalk Processing of STEP Part 21 Files. In Etalle, S., Truszczyński, M., eds.: Proceedings of the 22nd International Conference on Logic Programming. Number 4079 in Lecture Notes in Computer Science, Berlin Heidelberg, Springer-Verlag (August 2006) 453–454
11. Moura, P., Dias, A.M.: L-FLAT: Logtalk Toolkit for Formal Languages and Automata. <http://code.google.com/p/lflat/> (2009)
12. Wermelinger, M., Dias, A.M.: A Prolog Toolkit for Formal Languages and Automata. In: ITiCSE '05: Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, New York, NY, USA, ACM (2005) 330–334

---

# Adaptive Reasoning for Cooperative Agents

Luís Moniz Pereira and Alexandre Miguel Pinto  
(lmp|amp)@di.fct.unl.pt

Centro de Inteligência Artificial - CENTRIA  
Universidade Nova de Lisboa

**Abstract.** Using explicit affirmation and explicit negation, whilst allowing for a third logic value of undefinedness, can be useful in situations where decisions have to be taken on the basis of scarce, ambiguous, or downright contradictory information. In a three-valued setting, we consider an agent that learns a definition for both the target concept and its opposite, considering positive and negative examples as instances of two disjoint classes. Explicit negation is used to represent the opposite concept, while default negation is used to ensure consistency and to handle exceptions to general rules. Exceptions are represented by examples covered by the definition for a concept that belong to the training set for the opposite concept.

One single agent exploring an environment may gather only so much information about it and that may not suffice to find the right explanations. In such case, a cooperative multi-agent strategy, where each agent explores a part of the environment and shares with the others its findings, might provide better results. We describe one such framework based on a distributed genetic algorithm enhanced by a Lamarckian operator for belief revision. The agents communicate their candidate explanations — coded as chromosomes of beliefs — by sharing them in a common pool. Another way of interpreting this communication is in the context of argumentation. In the process of taking all the arguments and trying to find a common ground or consensus we might have to change, or review, some of assumptions of each argument.

The resulting framework we present is a collaborative perspective of argumentation where arguments are put together at work in order to find the possible 2-valued consensus of opposing positions of learnt concepts in an evolutionary pool in order to find the “best” explanation to the observations.

## 1 Introduction

Expressing theories as Logic Programs has become more natural and common as the field of Computational Logic has grown mature and other fields started to use its tools and results. Theories are usually expressed as a set of ‘if-then’ rules and facts which allow for the derivation, through the use of logic inference, of non-obvious results. When writing such rules and facts, explicit negation, just like explicit affirmation, can be used to formalize sure knowledge which provides inevitable results.

Theories can be further refined by adding special rules taking the form of Integrity Constraints (ICs). These impose that, whatever the assumptions might be, some conditions must be met. One implicit constraint on every reasonable theory is overall consistency, i.e., it must not be possible to derive one conclusion and its opposition.

---

Since in the real world the most common situation is one where there is incomplete and updatable information, any system making a serious attempt at dealing with real situations must cope with such complexities. To deal with this issue, the field of Computational Logic has also formalized another form of negation, Default Negation, used to express uncertain knowledge and exceptions, and used to derive results in the absence of complete information. When new information updates the theory, old conclusions might no longer be available (because they were relying on assumptions that became false with the new information), and further new conclusions might now be derived (for analogous reasons).

The principle we use is thus the *Unknown World Assumption* (UWA) where everything is unknown or undefined until we have some solid evidence of its truthfulness or falseness. This principle differs from the more usual *Closed World Assumption* (CWA) where everything is assumed false until there is solid evidence of its truthfulness. We believe the UWA stance is more skeptical, cautious, and even more realistic than the CWA. We do not choose a fuzzy logic approach due to its necessity of specific threshold values. For such an approach we would need to compute those values *a priori*, possibly recurring to a probabilistic frequency-based calculation. Accordingly, we use a 3-valued logic (with the *undefined* truth value besides the *true* and *false* ones) instead of a more classical 2-valued logic.

We start by presenting the method for theory building from observations we use — a 3-valued logic rule learning method, — and in the following section we focus on a method to analyze observations and to provide explanations for them given the learned theory. We show how the possible alternative explanations can be viewed as arguments for and against some hypotheses, and how we can use these arguments in a collaborative way to find better consensual explanations. Conclusions and outlined future work close this paper.

## 2 Theory building and refinement

Complete information about the world is impossible to achieve and it is necessary to reason and act on the basis of the available partial information coming from distinct agents. In situations of incomplete knowledge, it is important to distinguish between what is true, what is false, and what is unknown or undefined.

Such a situation occurs, for example, when each agent incrementally gathers information from the surrounding world and has to select its own actions on the basis of acquired knowledge. If the agent learns in a two-valued setting, it can encounter the problems that have been highlighted in [10]. When learning in a specific to general way, it will learn a cautious definition for the target concept and it will not be able to distinguish what is false from what is not yet known. Supposing the target predicate represents the allowed actions, then the agent will not distinguish forbidden actions from actions with a still unknown outcome and this can restrict the agent's acting power. If the agent learns in a general to specific way (i.e., the agent starts with a most general concept and progressively restricts it by adding exceptions as he learns), instead, it will not know the difference between what is true and what is unknown and, therefore, it can try actions with an unknown outcome. Rather, by learning in a three-valued setting,

---

it will be able to distinguish between allowed actions, forbidden actions, and actions with an unknown outcome. In this way, the agent will know which part of the domain needs to be further explored and will not try actions with an unknown outcome unless it is trying to expand its knowledge.

In [25] the authors showed that various approaches and strategies can be adopted in Inductive Logic Programming (ILP, henceforth) for learning with Extended Logic Programs (ELP) — including explicit negation — under an extension of well-founded semantics. As in [18, 19], where answer-sets semantics is used, the learning process starts from a set of positive and negative examples plus some background knowledge in the form of an extended logic program. Positive and negative information in the training set are treated equally, by learning a definition for both a positive concept  $p$  and its (explicitly) negated concept  $\neg p$ .

Default negation is used in the learning process to handle *exceptions* to general rules. Exceptions are examples covered by the definition for the positive concept that belong to the training set for the negative concept or examples covered by the definition for the negative concept that belong to the training set for the positive concept.

In this work, we consider standard ILP techniques to learn a concept and its opposite. Indeed, separately learned positive and negative concepts may conflict and, in order to handle possible *contradiction*, contradictory learned rules are made defeatable by making the learned definition for a positive concept  $p$  depend on the default negation of the negative concept  $\neg p$ , and vice-versa, i.e., each definition, possibly arising from distinct agents, is introduced as an exception to the other.

The learned theory will contain rules of the form:

$$p(\mathbf{X}) \leftarrow \text{Body}^+(\mathbf{X}) \qquad \neg p(\mathbf{X}) \leftarrow \text{Body}^-(\mathbf{X})$$

for every target predicate  $p$ , where  $\mathbf{X}$  stands for a tuple of arguments. In order to satisfy the completeness requirement, the rules for  $p$  will entail all positive examples while the rules for  $\neg p$  will entail all (explicitly negated) negative examples. The consistency requirement is satisfied by ensuring that both sets of rules do not entail instances of the opposite element in either of the training sets.

The ILP techniques to be used depend on the level of generality that we want to have for the two definitions: we can look for the Least General Solution (LGS) or the Most General Solution (MGS) of the problem of learning each concept and its complement. In practice, LGS and MGS are not unique and real systems usually learn theories that are not the least nor most general, but closely approximate one of the two. In the following, these concepts will be used to signify approximations to the theoretical concepts.

## 2.1 Strategies for Combining Different Generalizations

The generality of concepts to be learned is an important issue when learning in a three-valued setting. In a two-valued setting, once the generality of the definition is chosen, the extension (i.e., the generality) of the set of false atoms is automatically decided, because it is the complement of the true atoms set. In a three-valued setting, rather, the extension of the set of false atoms depends on the generality of the definition learned for the negative concept. Therefore, the corresponding level of generality may be chosen

---

independently for the two definitions, thus affording four epistemological cases. The adoption of ELP allows case combination to be expressed in a declarative and smooth way.

When classifying an as yet unseen object as belonging to a concept, we may later discover that the object belongs to the opposite concept. The more we generalize a concept, the higher is the number of unseen atoms covered by the definition and the higher is the risk of an erroneous classification. Depending on the damage that may derive from such a mistake, we may decide to take a more cautious or a more confident approach. If the possible damage from an over extensive concept is high, then one should learn the LGS for that concept, if the possible damage is low then one can generalize the most and learn the MGS. The overall risk will depend also on the use of the learned concepts within other rules: we need to take into account the damage that may derive from mistakes made on concepts depending on the target one.

The problem of selecting a solution of an inductive problem according to the cost of misclassifying examples has been studied in a number of works. PREDICTOR [15] is able to select the cautiousness of its learning operators by means of meta-heuristics. These meta-heuristics make the selection based on a user-input penalty for prediction error. In [33] Provost provides a method to select classifiers given the cost of misclassifications and the prior distribution of positive and negative instances. The method is based on the Receiver Operating Characteristic (ROC) [16] graph from signal theory that depicts classifiers as points in a graph with the number of false positives on the X axis and the number of true positive on the Y axis. In [28] it is discussed how the different costs of misclassifying examples can be taken into account into a number of algorithms: decision tree learners, Bayesian classifiers and decision list learners.

As regards the confidence in the training set, we can prefer to learn the MGS for a concept if we are confident that examples for the opposite concept are correct and representative of the concept. In fact, in top-down methods, negative examples are used in order to delimit the generality of the solution. Otherwise, if we think that examples for the opposite concept are not reliable, then we should learn the LGS.

## 2.2 Strategies for Eliminating Learned Contradictions

The learned definitions of the positive and negative concepts may overlap. In this case, we have a contradictory classification for the objective literals<sup>1</sup> in the intersection. In order to resolve the conflict, we must distinguish two types of literals in the intersection: those that belong to the training set and those that do not, also dubbed *unseen* atoms.

In the following we discuss how to resolve the conflict in the case of unseen literals and of literals in the training set. We first consider the case in which the training sets are disjoint, and we later extend the scope to the case where there is a non-empty intersection of the training sets, when they are less than perfect. From now onwards,  $\mathbf{X}$  stands for a tuple of arguments.

For unseen literals, the conflict is resolved by classifying them as undefined, since the arguments supporting the two classifications are equally strong. Instead, for literals

---

<sup>1</sup> An ‘objective literal’ in a Logic Program is just an atom, possibly explicitly negated.



---

in the training set, the conflict is resolved by giving priority to the classification stipulated by the training set. In other words, literals in a training set that are covered by the opposite definition are considered as *exceptions* to that definition.

*Contradiction on Unseen Literals* For unseen literals in the intersection, the undefined classification is obtained by making opposite rules mutually defeasible, or “non-deterministic” (see [3, 6]). The target theory is consequently expressed in the following way:

$$p(\mathbf{X}) \leftarrow p^+(\mathbf{X}), \text{not } \neg p(\mathbf{X}) \qquad \neg p(\mathbf{X}) \leftarrow p^-(\mathbf{X}), \text{not } p(\mathbf{X})$$

where  $p^+(\mathbf{X})$  and  $p^-(\mathbf{X})$  are, respectively, the definitions learned for the positive and the negative concept, obtained by renaming the positive predicate by  $p^+$  and its explicit negation by  $p^-$ . From now onwards, we will indicate with these superscripts the definitions learned separately for the positive and negative concepts.

We want both  $p(\mathbf{X})$  and  $\neg p(\mathbf{X})$  to act as an exception to the other. In case of contradiction, this will introduce mutual circularity, and hence undefinedness according to *WFSX*. For each literal in the intersection of  $p^+$  and  $p^-$ , there are two stable models, one containing the literal, the other containing the opposite literal.

*Contradiction on Examples* Theories are tested for consistency on all the literals of the training set, so we should not have a conflict on them. However, in some cases, it is useful to relax the consistency requirement and learn clauses that cover a small amount of counterexamples. This is advantageous when it would be otherwise impossible to learn a definition for the concept, because no clause is contained in the language bias that is consistent, or when an overspecific definition would be learned, composed of many specific clauses instead of a few general ones. In such cases, the definitions of the positive and negative concepts may cover examples of the opposite training set. These must then be considered exceptions, which are then due to abnormalities in the opposite concept.

Let us start with the case where some literals covered by a definition belong to the opposite training set. We want of course to classify these according to the classification given by the training set, by making such literals *exceptions*. To handle exceptions to classification rules, we add a negative default literal of the form *not abnorm<sub>p</sub>(X)* (resp. *not abnorm<sub>¬p</sub>(X)*) to the rule for  $p(\mathbf{X})$  (resp.  $\neg p(\mathbf{X})$ ), to express possible abnormalities arising from exceptions. Then, for every exception  $p(\mathbf{t})$ , an individual fact of the form *abnorm<sub>p</sub>(t)* (resp. *abnorm<sub>¬p</sub>(t)*) is asserted so that the rule for  $p(\mathbf{X})$  (resp.  $\neg p(\mathbf{X})$ ) does not cover the exception, while the opposite definition still covers it. In this way, exceptions will figure in the model of the theory with the correct truth value. The learned theory thus takes the form:

$$p(\mathbf{X}) \leftarrow p^+(\mathbf{X}), \text{not } \text{abnorm}_p(\mathbf{X}), \text{not } \neg p(\mathbf{X}) \tag{1}$$

$$\neg p(\mathbf{X}) \leftarrow p^-(\mathbf{X}), \text{not } \text{abnorm}_{\neg p}(\mathbf{X}), \text{not } p(\mathbf{X}) \tag{2}$$

### 3 Explaining Observations and Meta-Learning

After a theory is built it can now be used to analyze observations and to provide explanations for them. Such explanations are sets of abductive hypotheses which, when

---

assumed true under the theory at hand, yield the observations as conclusions. There can be, of course, many different possible explanations. In the end, most of the times, we want to find the single “best” explanation for the observations, and hence we must have some mechanism to identify the “best” solution among the several alternative ones.

### 3.1 Abduction

Deduction and abduction differ in the direction in which a rule like “ $a$  entails  $b$ ” is used for inference. Deduction allows deriving  $b$  as a consequence of  $a$ ; i.e., deduction is the process of deriving the consequences of what is known. Abduction allows deriving  $a$  as a hypothetical explanation of  $b$ .

### 3.2 Finding alternative explanations for observations

Trying to find explanations for observations can be implemented by simply finding the alternative abductive models that satisfy both the theory’s rules and the observations. The latter can be coded as Integrity Constraints (ICs) which are added to the theory thereby imposing the truthfulness of the observations they describe.

*Example 1. Running example*

We will use this running example throughout the rest of the chapter.

Consider the following Logic Program consisting of four rules. According to this program a ‘professional’ is someone who is a regular employee or someone who is a boss in some company. Also, a non-employee is assumed to be a student as well as all those who are junior (all children should go to school!).

$$\begin{array}{ll} \text{professional}(X) \leftarrow \text{employee}(X) & \text{student}(X) \leftarrow \text{not employee}(X) \\ \text{professional}(X) \leftarrow \text{boss}(X) & \text{student}(X) \leftarrow \text{junior}(X) \end{array}$$

For now keep this example in mind as we will use it to illustrate the concepts and methods we are about to describe. Assume that ‘*employee/1*’, ‘*boss/1*’, and ‘*junior/1*’ are abducible hypotheses.

Adding one single IC to the theory might yield several alternative 2-valued models (sets of abductive hypotheses) satisfying it, let alone adding several ICs.

In the example above, adding just the Integrity Constraint ‘ $\perp \leftarrow \text{not professional}(\text{john})$ ’ — coding the fact that John is a professional — would yield two alternative abductive solutions:  $\{\text{employee}(\text{john})\}$  and  $\{\text{boss}(\text{john})\}$ .

When the information from several observations comes in at one single time, several ICs must be added to the theory in order to be possible to obtain the right explanations for the corresponding observations.

Our concern is with finding explanations to observations. In a nutshell, we split the set of observations into several smaller subsets; then we create several agents and give each agent the same base theory and a subset of the observations coded as ICs. We then allow each agent to come up with several alternative explanations to its ICs; the explanations need not be minimal sets of hypotheses.

---

Going back again to our running example, if we also know that John is a student, besides adding the ' $\perp \leftarrow \text{not professional}(\text{john})$ ' IC we must also add the ' $\perp \leftarrow \text{not student}(\text{john})$ ' IC.

Finding possible alternative explanations is one problem; finding which one(s) is(are) the “best” is another issue. In the next section we assume “best” means minimal set of hypotheses and we describe the method we use to find such best.

### 3.3 Choosing the best explanation

One well known method for solving complex problems widely used by creative teams is that of ‘brainstorming’. In a nutshell, every agent participating in the ‘brainstorm’ contributes by adding one of his/her ideas to the common idea-pool shared by all the agents. All the ideas, sometimes clashing and oppositional among each other, are then mixed, crossed and mutated. The solution to the problem arises from the pool after a few iterations of this evolutionary process.

The evolution of alternative ideas and arguments in order to find a collaborative solution to a group problem is the underlying inspiration of this work.

**Evolutionary Inspiration** Darwin’s theory is based on the concept of natural selection: only those individuals that are most fit for their environment survive, and are thus able to generate new individuals by means of reproduction. Moreover, during their lifetime, individuals may be subject to random mutations of their genes that they can transmit to offspring. Lamarck’s [21] theory, instead, states that evolution is due to the process of adaptation to the environment that an individual performs in his/her life. The results of this process are then automatically transmitted to his/her offspring, via its genes. In other words, the abilities learned during the life of an individual can modify his/her genes.

Experimental evidence in the biological kingdom has shown Darwin’s theory to be correct and Lamarck’s to be wrong. However, this does not mean that the process of adaptation (or learning) does not influence evolution. Baldwin [4] showed how learning could influence evolution: if the learned adaptations improve the organism’s chance of survival then the chances for reproduction are also improved. Therefore there is selective advantage for genetically determined traits that predisposes the learning of specific behaviors. Baldwin moreover suggests that selective pressure could result in new individuals to be born with the learned behavior already encoded in their genes. This is known as the Baldwin effect. Even if there is still debate about it, it is accepted by most evolutionary biologists.

Lamarckian evolution [22] has recently received a renewed attention because it can model cultural evolution. In this context, the concept of “meme” has been developed. A meme is the cognitive equivalent of a gene and it stores abilities learned by an individual during his lifetime, so that they can be transmitted to his offspring.

In the field of genetic programming [20], Lamarckian evolution has proven to be a powerful concept and various authors have investigated the combination of Darwinian and Lamarckian evolution.

In [24] the authors propose a genetic algorithm for belief revision that includes, besides Darwin’s operators of selection, mutation and crossover, a logic based Lamarckian

---

operator as well. This operator differs from Darwinian ones precisely because it modifies a chromosome coding beliefs so that its fitness is improved by experience rather than in a random way. There, the authors showed that the combination of Darwinian and Lamarckian operators are useful not only for standard belief revision problems, but especially for problems where different chromosomes may be exposed to different constraints, as in the case of a multi-agent system. In these cases, the Lamarckian and Darwinian operators play different roles: the Lamarckian one is employed to bring a given chromosome closer to a solution (or even find an exact one) to the current belief revision problem, whereas the Darwinian ones exert the role of randomly producing alternative belief chromosomes so as to deal with unencountered situations, by means of exchanging genes amongst them.

**Evolving Beliefs** Belief revision is an important functionality that agents must exhibit: agents should be able to modify their beliefs in order to model the outside world. What's more, as the world may be changing, a pool of separately and jointly evolved chromosomes may code for a variety of distinct belief evolution potentials that can respond to world changes as they occur. This dimension has been explored in [24] with specific experiments to that effect. Mark that it is not our purpose to propose here a competitor to extant classical belief revision methods, in particular as they apply to diagnosis. More ambitiously, we do propose a new and complementary methodology, which can empower belief revision — any assumption based belief revision — to deal with time/space distributed, and possibly intermittent or noisy laws about an albeit varying artifact or environment, possibly by a multiplicity of agents which exchange diversified genetically encoded experience. We consider a definition of the belief revision problem that consists in removing a contradiction from an extended logic program by modifying the truth value of a selected set of literals corresponding to the abducible hypotheses. The program contains as well clauses with *falsum* ( $\perp$ ) in the head, representing ICs. Any model of the program must ensure the body of ICs false for the program to be non-contradictory. Contradiction may also arise in an extended logic program when both a literal  $L$  and its opposite  $\neg L$  are obtainable in the model of the program. Such a problem has been widely studied in the literature, and various solutions have been proposed that are based on abductive logic proof procedures. The problem can be modeled by means of a genetic algorithm, by assigning to each abducible of a logic program a gene in a chromosome. In the simplest case of a two valued revision, the gene will have the value 1 if the corresponding abducible is *true* and the value 0 if the abducible is *false*. The fitness functions that can be used in this case are based on the percentage of ICs that are satisfied by a chromosome. This is, however, an over-simplistic approach since it assumes every abducible is a predicate with arity 0, otherwise a chromosome would have as many genes as the number of all possible combinations of ground values for variables in all abducibles.

**Specific Belief Evolution Method** In multi-agent joint belief revision problems, agents usually take advantage of each other's knowledge and experience by explicitly communicating messages to that effect. In our approach, however, we introduce a new and complementary method (and some variations of it), in which we allow knowledge and

---

experience to be coded as genes in an agent. These genes are exchanged with those of other agents, not by explicit message passing but through the crossover genetic operator. Crucial to this endeavor, a logic-based technique for modifying cultural genes, i.e. memes, on the basis of individual agent experience is used.

The technique amounts to a form of belief revision, where a meme codes for an agent's belief or assumptions about a piece of knowledge, and which is then diversely modified on the basis of how the present beliefs may be contradicted by laws (expressed as ICs). These mutations have the effect of attempting to reduce the number of unsatisfied constraints. Each agent possesses a pool of chromosomes containing such diversely modified memes, or alternative assumptions, which cross-fertilize Darwinianly amongst themselves. Such an experience in genetic evolution mechanism is aptly called Lamarckian.

Since we will subject the sets of beliefs to an evolutionary process (both Darwinian and Lamarckian) we will henceforth refer to this method as "Belief Evolution" (BE) instead of the classical "Belief Revision" (BR).

*General Description of the Belief Evolution Method* Each agent keeps a population of chromosomes and finds a solution to the BE problem by means of a genetic algorithm. We consider a formulation of the distributed BE problem where each agent has the same set of abducibles and the same program expressed theory, but is exposed to possibly different constraints. Constraints may vary over time, and can differ because agents may explore different regions of the world. The genetic algorithm we employ allows each agent to cross over its chromosomes with chromosomes from other agents. In this way, each agent can be prepared in advance for situations that it will encounter when moving from one place to another.

The algorithm proposed for BE extends the standard genetic algorithm in two ways:

- crossover is performed among chromosomes belonging to different agents,
- a Lamarckian operator called Learn is added in order to bring a chromosome closer to a correct revision by changing the value of abducibles

*The Structure of a Chromosome* In BR and BE, each individual hypothesis is described by the truth value of all the abducibles. In our present setting, however, each gene encodes a ground literal, i.e., all its variables are bound to fixed values.

So, we represent a chromosome as a list of genes and memes, and different chromosomes may contain information about different genes. This implies a major difference to traditional genetic algorithms where every chromosome refers exactly to the same genes and the crossover and mutation operations are somewhat straightforward.

The memes in a chromosome will be just like genes — representing abducibles — but they will have extra information. Each meme has associated with it a counter keeping record of how many times the meme has been confirmed or refuted. Each time a meme is confirmed this value is increased, and each time it is refuted the value decreases. This value provides thus a measure of confidence in the corresponding meme.

*Example 2. Running example (cont.)*

---

Continuing with our running example, let us assume that both *professional(john)* and *student(john)* have been observed. We can create two agents, each with the same rule-set theory, and split the observations among them. We would have thus

<p>Agent 1:</p> <p><math>\leftarrow \text{not professional}(\text{john})</math></p>	<p>Agent 2:</p> <p><math>\leftarrow \text{not student}(\text{john})</math></p>
<p>Agent 1 and Agent 2:</p> <p><math>\text{professional}(X) \leftarrow \text{employee}(X)</math></p> <p><math>\text{professional}(X) \leftarrow \text{boss}(X)</math></p> <p><math>\text{student}(X) \leftarrow \text{not employee}(X)</math></p> <p><math>\text{student}(X) \leftarrow \text{junior}(X)</math></p>	

In the simplest case where a gene encodes an abductive ground literal Agent 1 would come up with two alternative abductive solutions for its IC  $\perp \leftarrow \text{not professional}(\text{john})$ :  $\{\text{employee}(\text{john})\}$  and  $\{\text{boss}(\text{john})\}$ . Moreover, Agent 2 would come up with two other alternative abductive solutions for its IC  $\perp \leftarrow \text{not student}(\text{john})$ :  $\{\text{not employee}(\text{john})\}$  and  $\{\text{junior}(\text{john})\}$ .

*Crossover* Since each agent knows only some ICs the abductive answer the algorithm seeks should be a combination of the partial answers each agent comes up with. In principle, the overlap on abducibles among two chromosomes coming from different agents should be less than total — after all, each agent is taking care of its own ICs which, in principle, do not refer to the exact same abducibles. Therefore, crossing over such chromosomes can simply turn out to be the merging of the chromosomes, i.e., the concatenation of the lists of abducibles.

If several ICs refer to the exact same abducibles the chromosomes from different agents will contain either the same gene — in which case we can see this as an ‘agreement’ between the agents as far as the corresponding abducible is concerned — or genes stating contradictory information about the same abducible. In this last case if the resulting concatenated chromosome turns out to be inconsistent in itself the fitness function will filter it out by assigning it a very low value.

#### Example 3. Running example (cont.)

Continuing with our running example, recall that Agent 1 would come up with two alternative abductive solutions for its IC  $\perp \leftarrow \text{not professional}(\text{john})$ :  $\{\text{employee}(\text{john})\}$  and  $\{\text{boss}(\text{john})\}$ . Moreover, Agent 2 would come up with two other alternative abductive solutions for its IC  $\perp \leftarrow \text{not student}(\text{john})$ :  $\{\text{not employee}(\text{john})\}$  and  $\{\text{junior}(\text{john})\}$ .

The crossing over of these chromosomes will yield the four combinations  $\{\text{employee}(\text{john}), \text{not employee}(\text{john})\}$ ,  $\{\text{employee}(\text{john}), \text{junior}(\text{john})\}$ ,  $\{\text{boss}(\text{john}), \text{not employee}(\text{john})\}$ , and  $\{\text{boss}(\text{john}), \text{junior}(\text{john})\}$ .

The first resulting chromosome is contradictory so it will be filtered out by the fitness function. The second chromosome correspond to the situation where John is a junior employee who is still studying — a quite common situation, actually. The third chromosome corresponds to the situation where John is a senior member of a company — a ‘boss’ — who is taking some course (probably a post-graduation study). The last

---

chromosome could correspond to the situation of a young entrepreneur who, besides owning his/hers company, is also a student — this is probably an uncommon situation and, if necessary, the fitness function can reflect that “unprobability”.

*Mutation* When considering a list of abducible literals the mutation operation resembles the standard mutation of genetic algorithms by changing one gene to its opposite; in this case negating the truth value of the abducted literal.

*Example 4. Running example (cont.)*

In the example we have been using this could correspond to mutating the chromosome  $\{not\ employee(john)\}$  to  $\{employee(john)\}$ , or to mutating the chromosome  $\{junior(john)\}$  to  $\{not\ junior(john)\}$ .

*The Lamarckian Learn operator* The Lamarckian operator Learn can change the values of variables of an abducible in a chromosome  $c_i$  so that a bigger number of constraints is satisfied, thus bringing  $c_i$  closer to a solution. Learn differs from a normal belief revision operator because it does not assume that all abducibles are false by CWA before the revision but it starts from the truth values that are given by the chromosome  $c_i$ . Therefore, it has to revise the values of variables of some abducibles and, in the particular case of an abducible without variables, from *true* to *false* or from *false* to *true*. This Lamarckian Learn operator will introduce an extra degree of flexibility allowing for changes to a chromosome to induce the whole belief evolution algorithm to search a solution considering new values for variables.

In the running example this could correspond, for example, to changing the chromosome  $\{junior(john)\}$  to  $\{junior(mary)\}$ , where ‘mary’ is another value in the domain range of the variable for abducible *junior*/1.

*The Fitness Functions* Various fitness functions can be used in belief revision. The simplest fitness function is the following  $Fitness(c_i) = (n_i/n)/(1 + NC)$ , where  $n_i$  is the number of integrity constraints satisfied by chromosome  $c_i$ ,  $n$  is the total number of integrity constraints, and  $NC$  is the number of contradictions in chromosome  $c_i$ . We will call it an accuracy fitness function.

## 4 Argumentation

In [12], the author shows that preferred maximal scenarios (with maximum default negated literals — the hypotheses) are always guaranteed to exist for NLPs; and that when these yield 2-valued complete (total), consistent, admissible scenarios, they coincide with the Stable Models of the program. However, preferred maximal scenarios are, in general, 3-valued. The problem we address now is how to define 2-valued complete models based on preferred maximal scenarios. In [31] the authors took a step further from what was achieved in [12], extending its results. They did so by completing a preferred set of hypotheses rendering it approvable, ensuring whole model consistency and 2-valued completeness.

The resulting semantics thus defined, dubbed Approved Models [31], is a conservative extension to the widely known Stable Models semantics [14] in the sense that every

---

Stable Model is also an Approved Model. The Approved Models are guaranteed to exist for every Normal Logic Program, whereas Stable Models are not. Concrete examples in [31] show how NLPs with no Stable Models can usefully model knowledge, as well as produce additional models. Moreover, this guarantee is crucial in program composition (say, from knowledge originating in divers sources) so that the result has a semantics. It is important too to warrant the existence of semantics after external updating, or in Stable Models based self-updating [1].

For the formal presentation and details of the Approved Models semantics see [31].

#### 4.1 Intuition

Most of the ideas and notions of argumentation we are using here come from the Argumentation field — mainly from the foundational work of Phan Minh Dung in [12]. In [31] the *Reductio ad Absurdum* reasoning principle is also considered. This has been studied before in [29], [30], and [32]. In this paper we consider an *argument* (or set of hypotheses) as a set  $S$  of abducible literals of a NLP  $P$ .

We have seen before examples of Extended Logic Programs — with explicit negation. In [8] the authors show that a simple syntactical program transformation applied to an ELP produces a Normal Logic Program with Integrity Constraints which has the exact same semantics as the original ELP.

*Example 5.* Transforming an ELP into a NLP with ICs

Taking the program

$$\begin{aligned} dangerous\_neighborhood &\leftarrow not \neg dangerous\_neighborhood \\ \neg dangerous\_neighborhood &\leftarrow not dangerous\_neighborhood \end{aligned}$$

we just transform the explicitly negated literal  $\neg dangerous\_neighborhood$  into the positive literal  $dangerous\_neighborhood^-$ , and the original  $dangerous\_neighborhood$  literal is converted into  $dangerous\_neighborhood^+$

Now, in order to ensure consistency, we just need to add the IC  
 $\perp \leftarrow dangerous\_neighborhood^+, dangerous\_neighborhood^-$ . The resulting transformed program is

$$\begin{aligned} dangerous\_neighborhood^+ &\leftarrow not dangerous\_neighborhood^- \\ dangerous\_neighborhood^- &\leftarrow not dangerous\_neighborhood^+ \\ \perp &\leftarrow dangerous\_neighborhood^+, dangerous\_neighborhood^- \end{aligned}$$

Now know that we can just consider NLPs with ICs without loss of generality and so, henceforth, we will assume just that case. NLPs are in fact the kind of programs most Inductive Logic Programming learning systems produce.

#### 4.2 Assumptions and Argumentation

Previously, we have seen that assumptions can be coded as abducible literals in Logic Programs and that those abducibles can be packed together in chromosomes. The evolutionary operators of genetic and memetic crossover, mutation and fitness function



---

applied to the chromosomes provide a means to search for a consensus of the initial assumptions since it will be a consistent mixture of these.

Moreover, the 2-valued contradiction removal method presented in subsection 2.2 is a very superficial one. That method removes the contradiction between  $p(\mathbf{X})$  and  $\neg p(\mathbf{X})$  by forcing a 2-valued semantics for the ELP to choose either  $p(\mathbf{X})$  or  $\neg p(\mathbf{X})$  since they now are exceptions to one another. It is a superficial removal of the contradiction because the method does not look into the reasons why both  $p(\mathbf{X})$  and  $\neg p(\mathbf{X})$  hold simultaneously. The method does not go back to find the underlying assumptions supporting both  $p(\mathbf{X})$  and  $\neg p(\mathbf{X})$  to find out which assumptions should be revised in order to restore overall consistency. Any one such method must fall back into the principles of argumentation: to find the arguments supporting one conclusion in order to prevent it if it leads to contradiction.

### 4.3 Collaborative Opposition

In [12] the author shows that the Stable Models of a NLP coincide with the 2-valued complete Preferred Extensions which are self-corroborating arguments.

The more challenging environment of a Semantic Web is one possible ‘place’ where the future intelligent systems will live in. Learning in 2-values or in 3-values are open possibilities, but what is most important is that knowledge and reasoning will be shared and distributed. Different opposing concepts and arguments will come from different agents. It is necessary to know how to conciliate those opposing arguments, and how to find 2-valued consensus as much as possible instead of just keeping to the least-commitment 3-valued consensus. In [31] the authors describe another method for finding such 2-valued consensus in an incremental way. In a nutshell, we start by merging together all the opposing arguments into a single one. The conclusions from the theory plus the unique merged argument are drawn and, if there are contradictions against the argument or contradictions inside the argument we non-deterministically choose one contradicted assumption of the argument and revise its truth value. The iterative repetition of this step eventually ends up in a non-contradictory argument (and all possibilities are explored because there is a non-deterministic choice).

In a way, the evolutionary method we presented in subsection 3.3 implements a similar mechanism to find the consensus non-contradictory arguments.

## 5 Conclusions

The two-valued setting that has been adopted in most work on ILP and Inductive Concept Learning in general is not sufficient in many cases where we need to represent real world data.

Standard ILP techniques can be adopted for separate agents to learn the definitions for the concept and its opposite. Depending on the adopted technique, one can learn the most general or the least general definition and combine them in different ways.

We have also presented an evolution-inspired algorithm for performing belief revision in a multi-agent environment. The standard genetic algorithm is extended in two

ways: first the algorithm combines two different evolution strategies, one based on Darwin's and the other on Lamarck's evolutionary theory and, second, chromosomes from different agents can be crossed over with each other. The Lamarckian evolution strategy is obtained by means of an operator that changes the genes (or, better, the memes) of agents in order to improve their joint fitness.

We have presented too a new and productive way to deal with oppositional concepts in a cooperative perspective, in different degrees. We use the contradictions arising from opposing agents' arguments as hints for the possible collaborations. In so doing, we extend the classical conflictual argumentation giving a new treatment and new semantics to deal with the contradictions.

## References

1. Alferes, J. J., Brogi, A., Leite, J. A., and Pereira, L. M. Evolving logic programs. In S. Flesca et al., editor, *JELIA*, volume 2424 of *LNCS*, pages 50–61. Springer, 2002.
2. Alferes, J. J., Damásio, C. V., and Pereira, L. M. (1994). SLX - A top-down derivation procedure for programs with explicit negation. In Bruynooghe, M., editor, *Proc. Int. Symp. on Logic Programming*. The MIT Press.
3. Alferes, J. J. and Pereira, L. M. (1996). *Reasoning with Logic Programming*, volume 1111 of *LNAI*. Springer-Verlag.
4. <http://www.psych.utoronto.ca/museum/baldwin.htm>
5. Baral, C., Gelfond, M., and Rushton, J. Nelson. Probabilistic reasoning with answer sets. In Vladimir Lifschitz and Ilkka Niemelä, editors, *LPNMR*, volume 2923 of *Lecture Notes in Computer Science*, pages 21–33. Springer, 2004.
6. Baral, C. and Gelfond, M. (1994). Logic programming and knowledge representation. *Journal of Logic Programming*, 19/20:73–148.
7. Damásio, C. V., Nejdil, W., and Pereira, L. M. (1994). REVISE: An extended logic programming system for revising knowledge bases. In Doyle, J., Sandewall, E., and Torasso, P., editors, *Knowledge Representation and Reasoning*, pages 607–618. Morgan Kaufmann.
8. Damásio, C. V. and Pereira, L. M. Default Negated Conclusions: Why Not?. In *ELP'96*, pages 103–117. Springer, 1996.
9. De Raedt, L. and Bruynooghe, M. (1989). Towards friendly concept-learners. In *Procs. of the 11th Intl. Joint Conf. on Artificial Intelligence*, pages 849–856. Morgan Kaufmann.
10. De Raedt, L. and Bruynooghe, M. (1990). On negation and three-valued logic in interactive concept learning. In *Procs. of the 9th European Conf. on Artificial Intelligence*.
11. De Raedt, L. and Bruynooghe, M. (1992). Interactive concept learning and constructive induction by analogy. *Machine Learning*, 8(2):107–150.
12. Dung, P. M. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–358, 1995.
13. Esposito, F., Ferilli, S., Lamma, E., Mello, P., Milano, M., Riguzzi, F., and Semeraro, G. (1998). Cooperation of abduction and induction in logic programming. In Flach, P. A. and Kakas, A. C., editors, *Abductive and Inductive Reasoning*, Pure and Applied Logic. Kluwer.
14. Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. In Kowalski, R. and Bowen, K. A., editors, *Procs. of the 5th Int. Conf. on Logic Programming*, pages 1070–1080. MIT Press.
15. Gordon, D. and Perlis, D. (1989). Explicitly biased generalization. *Computational Intelligence*, 5(2):67–81.
16. Green, D.M., and Swets, J.M. (1966). Signal detection theory and psychophysics. New York: John Wiley and Sons Inc.. ISBN 0-471-32420-5.

- 
17. Greiner, R., Grove, A. J., and Roth, D. (1996). Learning active classifiers. In *Procs. of the Thirteenth Intl. Conf. on Machine Learning (ICML96)*.
  18. Inoue, K. (1998). Learning abductive and nonmonotonic logic programs. In Flach, P. A. and Kakas, A. C., editors, *Abductive and Inductive Reasoning*, Pure and Applied Logic. Kluwer.
  19. Inoue, K. and Kudoh, Y. (1997). Learning extended logic programs. In *Procs. of the 15th Intl. Joint Conf. on Artificial Intelligence*, pages 176–181. Morgan Kaufmann.
  20. Koza, J.R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection* MIT Press
  21. Jean Baptiste Lamarck: <http://www.ucmp.berkeley.edu/history/lamarck.html>
  22. Grefenstette, J. J., (1991). Lamarckian learning in multi-agent environments
  23. Lamma, E., Riguzzi, F., and Pereira, L. M. (1988). Learning in a three-valued setting. In *Procs. of the Fourth Intl. Workshop on Multistrategy Learning*.
  24. Lamma, E., Pereira, L. M., and Riguzzi, F. Belief revision via lamarckian evolution. *New Generation Computing*, 21(3):247–275, August 2003.
  25. Lamma, E., Riguzzi, F., and Pereira, L. M. Strategies in combined learning via logic programs. *Machine Learning*, 38(1-2):63–87, January 2000.
  26. Lapointe, S. and Matwin, S. (1992). Sub-unification: A tool for efficient induction of recursive programs. In Sleeman, D. and Edwards, P., editors, *Procs. of the 9th Intl. Workshop on Machine Learning*, pages 273–281. Morgan Kaufmann.
  27. Lavrač, N. and Džeroski, S. (1994). *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.
  28. Pazzani, M. J., Merz, C., Murphy, P., Ali, K., Hume, T., and Brunk, C. (1994). Reducing misclassification costs. In *Procs. of the Eleventh Intl. Conf. on Machine Learning (ML94)*, pages 217–225.
  29. Pereira, L. M. and Pinto, A. M. Revised stable models - a semantics for logic programs. In G. Dias et al., editor, *Progress in AI*, volume 3808 of *LNCS*, pages 29–42. Springer, 2005.
  30. Pereira, L. M. and Pinto, A. M. Reductio ad absurdum argumentation in normal logic programs. In *Argumentation and Non-monotonic Reasoning (ArgNMR'07) workshop at LP-NMR'07*, pages 96–113, 2007.
  31. Pereira, L. M. and Pinto, A. M. Approved Models for Normal Logic Programs. In *LPAR*, pages 454–468. Springer, 2007.
  32. Pinto, A. M. Explorations in revised stable models — a new semantics for logic programs. Master's thesis, Universidade Nova de Lisboa, February 2005.
  33. Provost, F. J. and Fawcett, T. (1997). Analysis and visualization of classifier performance: Comparison under imprecise class and cost distribution. In *Procs. of the Third Intl. Conf. on Knowledge Discovery and Data Mining (KDD97)*. AAAI Press.
  34. Quinlan, J. (1990). Learning logical definitions from relations. *Machine Learning*, 5:239–266.
  35. Vere, S. A. (1975). Induction of concepts in the predicate calculus. In *Procs. of the Fourth Intl. Joint Conf. on Artificial Intelligence (IJCAI75)*, pages 281–287.
  36. Whitley, D., Rana, S., and Heckendorn, R.B. (1998) The Island Model Genetic Algorithm: On Separability, Population Size and Convergence

---

---

# Runtime Verification of Agent Properties

Stefania Costantini<sup>1</sup>, Pierangelo Dell’Acqua<sup>2</sup>, Luís Moniz Pereira<sup>3</sup>, and  
Panagiota Tsintza<sup>1</sup>

<sup>1</sup> Dip. di Informatica, Università di L’Aquila, Coppito 67100, L’Aquila, Italy  
stefcost@di.univaq.it

<sup>2</sup> Dept. of Science and Technology - ITN, Linköping University, Norrköping, Sweden  
pierangelo.dellacqua@itn.liu.se

<sup>3</sup> Departamento de Informática, Centro de Inteligência Artificial (CENTRIA), Universidade  
Nova de Lisboa, 2829-516 Caparica, Portugal  
lmp@di.fct.unl.pt

**Abstract.** In previous work, we have proposed a multi-level agent model with at least a meta-level aimed at meta-reasoning and meta-control. In agents, these aspects are strongly related with time and therefore we retain that they can be expressed by means of temporal-logic-like rules. In this paper, we propose an interval temporal logic inspired by METATEM, that allows properties to be verified in specific time interval situated either in the past or in the future. We adopt this logic for definition and run-time verification of properties which can imply modifications to the agent’s knowledge base.

## 1 Introduction

Agents are by definition software entities which interact with an environment, and thus are subject to modify themselves and evolve according to both external and internal stimuli, the latter due to the proactive and deliberative capabilities of the agent themselves (whenever encompassed by the agent model at hand). In past work, we defined semantic frameworks for agent approaches based on logic programming that account for: (i) the kind of evolution of reactive and proactive agents due to directly dealing with stimuli, that are to be coped with, recorded and possibly removed [1]; and (ii) the kind of evolution related to adding/removing rules from the agent knowledge base [2]. These frameworks have been integrated into an overall framework for logical evolving agents (cf. [3, 4]) where every agent is seen as the composition of a base-level (or object-level) agent program and one or more meta-levels. In this model, updates related to recoding stimuli are performed in a standard way, while updates involving the addition/deletion of sets of rules, related to learning, belief revision, etc. are a consequence of meta-level decisions.

As agent systems are more widely used in real-world applications, the issue of verification is becoming increasingly important (see [5] and the many references therein). In computational logic, two common approaches to the verification of computational systems are model checking [6] and theorem proving. There are many attempts to adapt these techniques to agents (see again [5]). In this paper, we address the problem concerning the monitoring of agent behavior against desired properties, or with respect to a

---

certain specification, in a different way. We assume defined, possibly both at the object and at the meta-level, axioms that determine properties to be respected or enforced, or simply verified, whenever a property is desirable but not mandatory. We assume these properties to be verified at runtime. Upon verification of a property (which is evaluated within a context instantiated onto the present circumstances), suitable actions can be undertaken, that we call in general *improvement action*. Improvements can imply *revision* of the agent knowledge, or tentative *repair* of malfunctioning, or tentative improvement of future behavior, according to the situation at hand. Our approach is to some extent similar to that of [7] for evolving software.

As many of the properties to be defined and verified imply temporal aspects, we have considered to adopt the temporal logic METATEM [8, 9] as this logic is specially tailored to an agent context. Since properties should often be defined on certain intervals, we define a variant of METATEM, that we call I-METATEM, where connectives are defined over intervals. We do not adopt the full power of METATEM rules, where connectives are interpreted as modalities, and semantics provided accordingly. Instead, we remain within the realm of logic programming, and interpret the temporal axioms in the context of the above-mentioned semantic framework. Therefore, we should better call our axioms “temporal-logic-like” axioms. However, to simulate to some extent the power of modal logic, improvements can imply the removal/addition of new temporal-logic-like axioms. The addition of new ones determines their immediate operational use. In this way, we stay within our semantic framework, where we are able to provide a full declarative semantics and an efficient corresponding operational semantics, as demonstrated by the existing implementations ([2, 10]), though the whole proposed approach has not been fully implemented yet.

The plan of the paper is as follows. In Section 2 we summarize the features of the agent model our framework is based upon. This model is general, and many existing agent-oriented logic languages can be easily rephrased in terms of it. In Section 3 we shortly summarize the METATEM temporal logic, and then introduce the proposed extension. In Section 4 we show how we mean to use temporal-logic-like rules for defining properties, how these properties are meant to be verified, and we establish our notion of improvement. Then we conclude.

## 2 Layered Agent (Meta-)Model

We do not mean to restrict the proposed approach to one single agent model/language. Therefore, we refer to an abstract agent model (we might say, meta-model). In this way, the approach can be adapted to any specific agent formalism which can be seen as an instance of this meta-model. We therefore refer to the abstract multi-layer meta-framework for agents proposed in [3, 4]. In this framework, an agent is considered to be composed of two distinct interacting levels: the BA (standing for Base-Agent) and one or more meta-levels. BA is the base level, whereas MA (Meta-Agent) along with IEA (Information Exchange Agent) constitute the two meta-levels. Here we assume that BA is a logic program and make the additional assumption that its semantics may ascribe multiple models to BA in order to deal with “uncertainty”. For the semantics of logic

---

programs we can adopt one of those reported in the survey [11] and for the semantics dealing with “uncertainty” we can adopt the Answer Set Semantics [12].

The meta-level, by means of both components MA and IEA, performs various kinds of meta-reasoning and is responsible for supervising and coordinating the BA’s activities. MA is in charge of coordinating all activities and takes decisions over the BA. More precisely, the MA level will be the one up to decide which modifications have to be undertaken onto the BA level, in order to correct or improve inadequacies and unexpected behavior. The IEA level instead decides and evaluates when an interaction with the society is necessary in order to exchange knowledge: in fact, agents are in general not entities standing alone but, rather, are part of possibly several groups to form a society. Below we do not give any formal definition of BA, MA and IEA as their actual form depends upon the various possible concrete instances of the meta-framework. Rather, we specify the requirements they have to obey. We also define the overall architecture, and outline a possible semantic account.

## 2.1 Agent Model: operational behavior

To define the operational behavior of the agent meta-model we exploit our previous work reported in [3, 4]. Each agent is considered as a logic program that will evolve interacting with the environment. In fact, the interaction triggers many agent activities such as response, goals identification, decisions on recording and pruning the gathered information. Of course, these activities will be affected by the belief, desire and intention control that is part of MA. Note that this component will itself evolve and change in time as a result of the interaction with the society. In this paper, we consider the evolution of the initial agent into subsequent related versions of the agent itself. Therefore, we consider that each interaction will, eventually, determine the evolution of the initial agent in terms of successive transformations.

We start by providing a more formal view of agent evolution. We consider a generic instance of our agent meta-model that we refer to as  $\mathcal{M}$ . The agent model  $\mathcal{M}$  will have to provide an agent-oriented programming language and a control mechanism. For example, if  $\mathcal{M}$  provides a prolog-like programming language,  $\mathcal{C}$  may be a meta-interpreter, and  $\mathcal{CI}$  may be a set of directives to the interpreter. Below we describe the operational behavior of our meta-model thus providing a specification to which  $\mathcal{M}$  should conform, whatever its specific functioning, to be seen as an instance of our framework.

**Definition 1.** *An agent program is a tuple  $\langle BA, MA, \mathcal{C}, \mathcal{CI} \rangle$  of software components where  $BA$  and  $MA$  are logic programs,  $\mathcal{C}$  is the control component and  $\mathcal{CI}$  is the component containing control information.*

Specifically, the control component  $\mathcal{C}$  takes as input both the logic programs  $BA$  and  $MA$  and the control information  $\mathcal{CI}$ . The  $\mathcal{CI}$  component has the role of customizing the *run-time* behavior of the agent. For example,  $\mathcal{CI}$  may contain directives stating the priorities among different events/goals that the agent has to cope with.

In the following, we clarify how the control  $\mathcal{C}$  and control information  $\mathcal{CI}$  components are enabled to actually affect the operational behavior of agents. In fact, both components are taken as input by an underlying control mechanism  $\mathcal{U}$  that implements

---

the operational counterpart of the agent model. For example, if the agent model provides a prolog-like programming language the underlying control mechanism may be either an interpreter or a virtual machine related to a compiler. The underlying control mechanism  $\mathcal{U}$  that is able to put into operation the various components of an agent model  $\mathcal{M}$ , is a transformation function starting from  $A_0$  and transforming it step by step into  $A_1, A_2, \dots$ . The transition from a generic step  $A_i$  into the next step  $A_{i+1}$  is defined as follows.

**Definition 2.** Let  $A_i = \langle BA_i, MA_i, C_i, \mathcal{CT}_i \rangle$  be an agent program at step  $i$ . Then, the underlying control mechanism  $\mathcal{U}$  is a binary function defined as:

$$A_i \xrightarrow{\mathcal{U}(C_i, \mathcal{CT}_i, w_i)} \langle BA_{i+1}, MA_{i+1}, C_{i+1}, \mathcal{CT}_{i+1} \rangle.$$

It is important to notice that, given an initial step  $A_0$ , subsequent steps  $A_i$ s in general do not follow deterministically. The reason is that each step depends both on the interaction with the society  $w_i$  (external environment) and on the internal choices of each agent that are based on its previous knowledge and “experience”. The underlying control mechanism  $\mathcal{U}$  can operate in two different ways by providing: (i) either different parallel threads for BA and MA, or (ii) an interleaving of control between the two levels. In the former case, MA continuously monitors BA. In the latter case control must somehow pass between the two levels, for instance as follows. Control will shift up from BA to MA periodically (and/or upon certain conditions) by means of an act called *upward reflection*. When controls shifts up, MA will revise the BA’s activities, which may imply constraints and condition verification. Vice versa, control will shift down from MA to BA by performing an act called *downward reflection*. Forms of control based on reflection in computational logic are formally accounted for in [13]. The frequency as well as the conditions of each type of shift is defined in the control information component  $\mathcal{CT}_i$  and therefore can be encoded as a subset of directives included in this component. A declarative semantics for evolving agents that fulfills the above-proposed meta-model is presented in [1]. Dynamic changes that MA can operate on BA can be semantically modeled by means of the approach of Evolving Logic Programs [14]. In the following, we will assume these formalizations as the semantic bases of the approach proposed here.

The meta-model and its operational behavior are consistent at least with the KGP ([15–17]) and DALI ([10, 18, 19]) agent-oriented languages.

### 3 I-METATEM: Temporal Logic in the proposed framework

In the previous section, we discussed the non determinism of states that can be reached by agents during their evolution. For defining temporal-logic-like rules while keeping the complexity under control, we are going to adapt the approach of METATEM, a propositional Linear-time Temporal Logic (LTL), that implicitly quantifies universally upon all possible paths. LTL logics are called linear because, in contrast to branching time logics, they evaluate each formula with respect to a vertex-labeled infinite path  $s_0 s_1 \dots$  where each vertex  $s_i$  in the path corresponds to a point in time (or “time instant” or “state”). In order to model the dynamic behavior of agents, we propose an extension



---

to the well-established METATEM logic called I-METATEM, an acronym standing for Interval METATEM.

### 3.1 METATEM

In this subsection, we present the basic elements of propositional METATEM logic [8, 9]. Its language is based both on the classical propositional logic enriched by temporal connectives and on the direct execution of temporal logic statements. The symbols used by METATEM are: (i) a set  $A_C$  of propositions controlled by the component which is being defined, (ii) a set  $A_E$  of propositions controlled by the environment (where  $A_C \cap A_E = \emptyset$ ), (iii) the alphabet of propositional symbols  $A_P = A_C \cup A_E$ , (iv) a set of propositional connectives such as **true**, **false**,  $\neg$ ,  $\wedge$ ,  $\vee$ ,  $\Rightarrow$  and  $\Leftrightarrow$ , and (v) a set of temporal connectives. The set of temporal connectives is composed of a number of unary and binary connectives referring to future-time and past-time. A METATEM program is a set of temporal logic rules of the form:

$$\text{past time antecedent} \rightarrow \text{future time consequent}$$

where the *past time antecedent* is considered as a temporal formula concerning the past while the *future time consequent* is a temporal formula concerning the present and future time. Therefore, a temporal rule is the one determining how the process should progress through stages.

### 3.2 I-METATEM

The purpose of this extension is to allow properties and anomalous behavior in agent evolution to be checked at run-time. Since agent evolution can be considered as an infinite sequence of states, it is often not possible (and not suitable) to verify properties on the entire sequence. Sometimes it is not even desirable, since one needs properties to hold within a certain time interval. Thus we propose the extension I-METATEM to the METATEM logic. Specifically, we introduce the new connectives  $F_{m,n}$ ,  $G_{m,n}$ ,  $G_{\langle m,n \rangle}$ ,  $N_{m,n}$ ,  $E_{m,n}$ ,  $\hat{G}_{m,n}$  and  $\hat{G}_{\langle m,n \rangle}$ .

#### Future-time connectives of I-METATEM (Assume $m < n$ )

- $X$  (*next state*).  $X\varphi$  states that  $\varphi$  will be true at next state.
- $G$  (*always in future*).  $G\varphi$  means that  $\varphi$  will always be true in every future state.
- $F$  (*sometime in future*).  $F\varphi$  states that there is a future state where  $\varphi$  will be true.
- $W$  (*weak until*).  $\varphi W\psi$  is true in a state  $s$  if  $\psi$  is true in a state  $t$ , in the future of state  $s$ , and  $\varphi$  is true in every state in the time interval  $[s,t)$  where  $t$  is excluded.
- $U$  (*strong until*).  $\varphi U\psi$  is true in a state  $s$  if  $\psi$  is true in a state  $t$ , in the future of state  $s$ , and  $\varphi$  is true in every state in the time interval  $[s,t]$  where  $t$  is included.
- $N$  (*never*).  $N\varphi$  states that  $\varphi$  should not become true in any future state.
- $\tau$  (*current state*).  $\tau(i)$  is true if  $s_i$  is the current state.
- $X_m$  (*future m-state*).  $X_m\varphi$  states that  $\varphi$  will be true in the state  $s_{m+1}$ .

- 
- $F_m$  (*bounded eventually*).  $F_m\varphi$  states that  $\varphi$  eventually has to hold somewhere on the path from the current state to  $s_m$ .
  - $F_{m,n}$  (*bounded eventually in time interval*).  $F_{m,n}\varphi$  states that  $\varphi$  eventually has to hold somewhere on the path from state  $s_m$  to  $s_n$ .
  - $G_{m,n}$  (*always in time interval*).  $G_{m,n}\varphi$  states that  $\varphi$  should become true at most at state  $s_m$  and then hold at least until state  $s_n$ .
  - $G_{\langle m,n \rangle}$  (*strong always in time interval*).  $G_{\langle m,n \rangle}\varphi$  states that  $\varphi$  should become true just in  $s_m$  and then hold until state  $s_n$ , and not in  $s_{n+1}$ .
  - $N_{m,n}$  (*bounded never*).  $N_{m,n}\varphi$  states that  $\varphi$  should not be true in any state between  $s_m$  and  $s_n$ .
  - $E_{m,n}$  (*sometime in time interval*).  $E_{m,n}\varphi$  states that  $\varphi$  has to occur one or more times between  $s_m$  and  $s_n$ .

**Past-time connectives of I-METATEM** (Assume  $m < n$ )

- $\hat{X}$  (*last state*).  $\hat{X}\varphi$  states that if there is a last state, then  $\varphi$  was true in that state.
- $\hat{F}$  (*some time in the past*).  $\hat{F}\varphi$  states that  $\varphi$  was true in some past state.
- $\hat{G}$  (*always in the past*).  $\hat{G}\varphi$  states that  $\varphi$  was true in all past states.
- $\hat{Z}$  (*weak since*).  $\varphi\hat{Z}\psi$  is true in a state  $s$  if  $\psi$  was true in a state  $t$  (in the past of state  $s$ ), and  $\varphi$  was true in every state of the time interval  $[t,s]$ .
- $\hat{S}$  (*since*).  $\varphi\hat{S}\psi$  is true in a state  $s$  if  $\psi$  was true in a state  $t$  (in the past of state  $s$ ), and  $\varphi$  was true at every state in the time interval  $[t,s]$ .
- $\hat{X}_m$  (*last  $m$ -state*).  $\hat{X}_m\varphi$  states that  $\varphi$  was true in the past state  $s_{m-1}$ .
- $\hat{F}_m$  (*bounded some time in the past*).  $\hat{F}_m\varphi$  states that  $\varphi$  was true in some past state before  $s_m$  included.
- $\hat{G}_{m,n}$  (*always in time interval in the past*).  $\hat{G}_{m,n}\varphi$  states that  $\varphi$  was true in the past state  $s_m$  and then it remained true at least until the past state  $s_n$ .
- $\hat{G}_{\langle m,n \rangle}$  (*strong always in time interval in the past*).  $\hat{G}_{\langle m,n \rangle}\varphi$  states that  $\varphi$  became true just in the past state  $s_m$  and then remained true exactly until the past state  $s_n$ .

The syntax of I-METATEM temporal formulae is given by the following definition.

**Definition 3.** *Formulae of I-METATEM logic are defined inductively in the usual way:*

$$\begin{aligned}
\varphi &::= p \mid \text{true} \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \tau(i) \\
\varphi &::= X\varphi_1 \mid \varphi_1 U \varphi_2 \mid \hat{X}\varphi_1 \mid \varphi_1 \hat{S} \varphi_2 \\
\varphi &::= \varphi_1 U_{m,n} \varphi_2 \mid \varphi_1 \hat{S}_{m,n} \varphi_2 \\
\varphi &::= (\varphi_1)
\end{aligned}$$

where  $p \in A_P$ , and  $\varphi_1$  and  $\varphi_2$  are formulae of I-METATEM.

**Semantics of I-METATEM** After having introduced the syntax of I-METATEM, we present the semantics of I-METATEM formulae. To do so, we first recall the notion of model structures to be used in the interpretation of temporal formulae. In the following, let  $\sigma$  be a sequence of states  $s_0s_1\dots$  and  $i$  a time instant. A *structure* is a pair

---

$\langle \sigma, i \rangle \in (\mathbb{N} \rightarrow 2^{A_P}) \times \mathbb{N}$  where  $A_P$  is the alphabet of propositional symbols. Given some moment in time, represented by a natural number  $j$ ,  $\sigma(j)$  is the set of propositions drawn from the alphabet  $A_P$  and denotes all the propositions that are true at time  $j$ . The satisfaction relation  $\models$  gives the interpretation to temporal formulae in the given model structure.

**Definition 4.** (Semantics of I-METATEM temporal formulae) *Let  $\langle \sigma, i \rangle$  be a structure. The semantics of I-METATEM temporal logic is defined as follows.*

*Propositions and propositional connectives*

$$\begin{aligned} \langle \sigma, i \rangle \models p & \quad \text{iff } p \in \sigma(i) \\ \langle \sigma, i \rangle \models \text{true} & \\ \langle \sigma, i \rangle \models \neg \varphi & \quad \text{iff } \langle \sigma, i \rangle \not\models \varphi \\ \langle \sigma, i \rangle \models \varphi \wedge \psi & \quad \text{iff } \langle \sigma, i \rangle \models \varphi \text{ and } \langle \sigma, i \rangle \models \psi \\ \langle \sigma, i \rangle \models \tau(i) & \end{aligned}$$

*Temporal connectives*

$$\begin{aligned} \langle \sigma, i \rangle \models X\varphi & \quad \text{iff } \langle \sigma, i+1 \rangle \models \varphi \\ \langle \sigma, i \rangle \models \varphi U \psi & \quad \text{iff } \exists k \in \mathbb{N} \langle \sigma, i+k \rangle \models \psi \text{ and } \forall j (0 \leq j < k) \langle \sigma, i+j \rangle \models \varphi \\ \langle \sigma, i \rangle \models \widehat{X}\varphi & \quad \text{iff if } i > 0, \text{ then } \langle \sigma, i-1 \rangle \models \varphi \\ \langle \sigma, i \rangle \models \varphi \widehat{S} \psi & \quad \text{iff } \exists k (1 \leq k \leq i) \langle \sigma, i-k \rangle \models \psi \text{ and } \forall j (1 \leq j < k) \langle \sigma, i-j \rangle \models \varphi \end{aligned}$$

*Temporal connectives in time intervals*

$$\begin{aligned} \langle \sigma, i \rangle \models \varphi U_{m,n} \psi & \quad \text{iff } i \leq m \leq n, \exists k (m-i \leq k \leq n-i) \langle \sigma, i+k \rangle \models \psi \text{ and } \\ & \quad \forall j (0 \leq j < k-i) \langle \sigma, i+j \rangle \models \varphi \\ \langle \sigma, i \rangle \models \varphi \widehat{S}_{m,n} \psi & \quad \text{iff } m \leq n < i, \exists k (i-m \leq k \leq i-n) \langle \sigma, i-k \rangle \models \psi \text{ and } \\ & \quad \forall j (1 \leq j < k) \langle \sigma, i-j \rangle \models \varphi \end{aligned}$$

In addition, we use the following abbreviations:

$$\begin{aligned} \text{false} & \triangleq \neg \text{true} \\ \varphi \vee \psi & \triangleq \neg(\neg\varphi \wedge \neg\psi) \\ \varphi \Rightarrow \psi & \triangleq \neg\varphi \vee \psi \end{aligned}$$

$$\begin{aligned} F\varphi & \triangleq \text{true } U \varphi \\ G\varphi & \triangleq \neg F \neg \varphi \\ \varphi W \psi & \triangleq \varphi U \psi \vee G\varphi \\ N\varphi & \triangleq \neg F \varphi \end{aligned}$$

---


$$\begin{aligned}\widehat{F}\varphi &\hat{=} \text{true } \widehat{S}\varphi \\ \widehat{G}\varphi &\hat{=} \neg \widehat{F}\neg\varphi \\ \varphi \widehat{Z}\psi &\hat{=} \varphi \widehat{S}\psi \vee \widehat{G}\varphi\end{aligned}$$

$$\begin{aligned}X_m\varphi &\hat{=} G_{m+1,m+1}\varphi \\ F_m\varphi &\hat{=} \bigvee_{i=1}^m (\tau(i) \wedge \text{true } U_{i,m}\varphi) \\ G_{m,n}\varphi &\hat{=} \neg(\text{true } U_{m,n}\neg\varphi) \\ G_{\langle m,n \rangle}\varphi &\hat{=} \neg F_m\varphi \wedge G_{m,n}\varphi \wedge G_{n+1,n+1}\neg\varphi \\ E_{m,n}\varphi &\hat{=} \neg G_{m,n}\varphi \wedge \neg N_{m,n}\varphi\end{aligned}$$

$$\begin{aligned}\widehat{X}_m\varphi &\hat{=} \widehat{G}_{m-1,m-1}\varphi \\ \widehat{F}_m\varphi &\hat{=} \bigvee_{i=1}^{m-1} (\tau(i) \wedge \text{true } \widehat{S}_{m,i-1}\varphi) \\ \widehat{G}_{m,n}\varphi &\hat{=} \neg(\text{true } \widehat{S}_{m,n}\neg\varphi) \\ \widehat{G}_{\langle m,n \rangle}\varphi &\hat{=} \widehat{G}_{0,m}\neg\varphi \wedge \widehat{G}_{m,n}\varphi \wedge \widehat{G}_{n+1,n+1}\neg\varphi\end{aligned}$$

Let  $\sigma \models \varphi$  be an abbreviation for  $\langle \sigma, 0 \rangle \models \varphi$ . We call  $\sigma$  a *model* of  $\varphi$  iff  $\sigma \models \varphi$ .

#### 4 I-METATEM for defining and verifying properties in logical agents

In our framework, agents are supposed to live in an open society where they interact with each other and with the environment, and where they can learn either by observing other agents behavior or by imitation. Given the evolving nature of learning agents, their behavior has to be checked from time to time and not (only) “a priori”. Model checking and other “a priori” approaches are static, since the underlying techniques require to write an ad-hoc interpreter and this operation cannot be re-executed whenever the agent learns a new piece of information. Note that in case of re-execution this operation would in principle be required a huge number of times, adding a further cost to the system. Moreover, an a priori full validation of agent’s behavior would have to consider all possible scenarios that are not known in advance. These are the reasons why we propose (also) a run-time control on agent behavior and evolution, for checking correctness during agents activity, rather than a model checking control.

To do so, we add to the underlying logic programming agent-oriented language the possibility of specifying rules including I-METATEM connectives. These rules can be defined both at the object level and at the meta-level to determine properties to be respected. These rules will be attempted, and whenever verified they may determine suitable modifications to the program itself. In the rest of this section, we first define the syntax of I-METATEM connectives in the context of logic programs, and then we define I-METATEM basic rules, I-METATEM contextual rules, and I-METATEM rules with improvements. Along with the explanation we provide some examples.

---

In our framework, we consider I-METATEM rules to be applied upon universally quantified formulae. Note that the negation connective *not* is interpreted in our setting as negation by default. To increase readability in I-METATEM rules, we write any temporal connective of the form  $O_{m,n}$  as  $O(m,n)$ . Sometimes, we omit the connective arguments when implied from the context, and in these cases we write  $O$  instead of  $O(m,n)$ . Also, as a special case, when we do not care about the starting point of an interval, we introduce the special constant *start* where  $O(start,n)$  means that  $O$  is checked since the “beginning of time” up to  $n$ , where the beginning of time coincides with the agent’s activation time. We also introduce the shorthand *now* standing for the time  $t$  for which  $\tau(t)$  holds.

#### 4.1 I-METATEM rules

In this section we present rules with an associated remedial action to be performed any time the rule is violated. We start by introducing the notion of basic rule and contextual rule.

**I-METATEM basic rules** We start by first introducing the basic form of rules.

**Definition 5.** Any I-METATEM formula  $\varphi$  is a rule.

Whenever checked, an I-METATEM rule is verified (or *succeeds*) whenever  $\varphi$  holds, otherwise the rule is violated. In the case of I-METATEM connectives (or their negation), this means that the related property holds either in the specified interval (if elapsed) or up to now. According to the semantic framework of [1], where special formulas can be designated to be periodically executed, I-METATEM rules will be periodically attempted (we also say “checked”). We assume some default frequency whenever not explicitly defined. As a first example of an I-METATEM rule, consider the following:

$$N(\text{goal}(g) \wedge \text{deadline}(g, t) \wedge \text{now}(T) \wedge T \leq t \wedge \text{not achieved}(g) \wedge \text{dropped}(g))$$

We assume predicates *goal*, *achieved* and *dropped* to be suitably defined in the agent’s knowledge base. Informally: *goal*( $g$ ) means that  $g$  is the goal that has to be achieved; *achieved*( $g$ ) is deemed true when the plan for reaching the goal  $g$  has been successfully completed; *dropped*( $g$ ) means that agent has dropped any attempt to achieve  $g$ . The rule states that it cannot be the case that a given goal not accomplished up to now but not expired yet (the deadline  $t$  for this goal has not been met), is dropped by the agent. There are in principle different ways to exploit this rule: (i) as an “a priori” check to be performed whenever a *drop* action is attempted; if the check fails, then the action is not allowed and (ii) as an “a posteriori” check on the agent behavior; in case of violation, some repair action should presumably be undertaken, as discussed below.

Notice that for performing this kind of evaluation we have to consider ground rules. In the above rule in fact, the only variable is the present time  $T$ , which is however instantiated by the predefined connective *now*. Below we generalize to the non-ground case.

---

**I-METATEM contextual rules** For the sake of generality, and in view of a changing environment, we propose a further extension of rule syntax to include variables instantiated by an *evaluation context* associated to each rule.

**Definition 6.** A contextual I-METATEM rule is a rule of the form  $\chi \Rightarrow \varphi$  where:

- $\chi$  is the evaluation context of the rule, and consists of a conjunction of logic programming literals;
- every variable occurring in  $\varphi$  must occur in the context  $\chi$ .

From Definition 6 it follows that the evaluation of a contextual rule becomes feasible only when grounded from the context. In order to clarify the syntax of a *contextual I-METATEM rule*, consider the following example:

$$[ \text{goal}(\text{Goal}), \text{priority}(\text{Goal}, \text{Pr}), \text{timeout}(\text{Pr}, \text{T\_out}) ] \Rightarrow \\ F(\text{T\_out}) \text{ achieved}(\text{Goal})$$

In this rule, the goal *Goal* is established by the context, which also contains the atoms *priority* and *timeout*. Informally, the rule requires that a goal with timeout *T\_out* (set according to the priority *Pr* of the goal itself), should actually be accomplished before the timeout. This contextual rule can be verified whenever instantiated to a specific goal *g*. In general, the rule will be repeatedly checked until it either succeeds, if *g* will be achieved in time, or it fails because the timeout will have elapsed.

**I-METATEM rules with improvement** Whenever an instance of an I-METATEM rule succeeds, it either expresses a desirable property or not. In the former case, some kind of “positive” action may be undertaken; in the latter case, a repair action will in general be required. We call the corresponding modification of the program an *improvement* or *remedial action*. Program modification/evolution is accounted for by the EVOLP semantics [2, 14].

We now extend the definition of contextual I-METATEM rules to specify a corresponding improvement action.

**Definition 7.** Let *A* be an atom. An I-METATEM rule with improvement is a rule of the form  $\chi \Rightarrow \varphi : A$  where  $\chi \Rightarrow \varphi$  is a contextual rule, and *A* is its improvement action.

Given a rule  $\chi \Rightarrow \varphi : A$ , whenever its monitoring condition  $\chi \Rightarrow \varphi$  holds, then the rule is checked and the corresponding improvement action *A* is attempted. The improvement action is specified via an atom that is executed as an ordinary logic programming goal. Consider again the previous example which monitors the achievement of goals, but extended to specify that, in case of violation, the current level of commitment of the agent to its objectives has to be increased. This can be specified as:

$$[ \text{goal}(\text{Goal}), \text{deadline}(\text{Goal}, \text{T}), \text{now}(\text{T2}), \text{T2} \leq \text{T} ] \Rightarrow \\ G(\neg \text{achieved}(\text{Goal}) \wedge \text{dropped}(\text{Goal})) : \text{inc\_comt}(\text{T2}) \\ \text{incr\_comt}(\text{T2}) \leftarrow \text{commitment\_level}(\text{T}, \text{L}), \\ \text{increase\_level}(\text{L}, \text{L2}), \\ \text{assert}(\text{not commitment\_level}(\text{T}, \text{L})), \\ \text{assert}(\text{commitment\_level}(\text{T2}, \text{L2}))$$

---

Suppose that at a certain time  $t$  the monitoring condition

$$G (\neg \text{achieved}(\text{Goal}) \wedge \text{dropped}(\text{Goal}))$$

holds for some specific goal  $g$ . Upon detection, the system will attempt the improvement action consisting in executing the goal  $\text{inc\_comt}(t)$ . Its execution will allow the system to perform the specified run-time re-arrangement of the program that attempts to cope with the unwanted situation: in the example, the module defining the rules that specify the level of commitment to which the agent obeys is retracted and substituted by a new one corresponding to a higher level.

Semantically, in our agent meta-model the execution of the improvement action will determine the update of the current agent program  $\mathcal{P}_i$ , returning a new agent program  $\mathcal{P}_{i+1}$ . The I-METATEM rules with improvements are to some extent similar to METATEM rules, though here one does not state properties of the future but rather specifies actions to be undertaken.

Based on this definition, we are able for instance to define rules that aim at detecting various kinds of anomalous behavior of an agent (for a discussion of run-time anomalies see, e.g., [20]). For example, we can introduce a rule for checking an unexpected behavior such as **omission**, which occur whenever an agent fails to perform the desired action/goal. The rule:

$$[\text{goal}(\text{Goal}), \text{not achieved}(\text{Goal}), \text{dropped}(\text{Goal}, T3), \text{confidence}(G, T3, C3)] \Rightarrow G(\text{confidence}(G, \text{now}, C) \wedge C3 \leq C) : \text{re\_exec}(\text{Goal})$$

states how the agent has to behave in the case of a dropped goal. If, after dropping the goal (because it has not been achieved in a given interval), the agent's confidence in being able to achieve the goal has increased, then the goal will be re-attempted.

To detect an anomalous behavior consisting of **duplication** or **incoherence**, i.e., an agent performs more than once the same action/goal when not necessary, we introduce the following rule:

$$[\text{goal}(\text{Goal})] \Rightarrow \widehat{F}_0 \text{ times\_exec}(\text{Goal}) > k : \text{disable}(\text{Goal})$$

with the role of checking if a goal/plan has been executed more times than a given threshold: if so, further execution of the goal will be disabled.

The following example outlines the so-called anomaly of **intrusion**, i.e., the case of an unexpected behavior, or unwanted consequence, arisen from the execution of a goal. Whenever the constraint defined below succeeds, as a repair a new constraint is asserted establishing that  $G$  cannot be further pursued, at least until a certain time has elapsed. As soon as asserted, the new constraint will start being checked.

$$[\text{now}(T), \text{goal}(\text{Goal}), \text{executed}(\text{Goal}), \text{consequence}(\text{Goal}, C)] \Rightarrow \widehat{F}_{0,T} \neg \text{desired}(C) : \text{assert}([\text{now}(T), \text{threshold}(T1)] \Rightarrow N(T, T1) \text{ exec}(\text{Goal}))$$

I-METATEM connectives can be used to check the past behavior and knowledge of the agent but also to influence its future behavior. The agent evolution entails also an evolution of recorded information, which in turn may affect the evaluation of social factors such as trust and confidence. Consider for instance the following example, where the level of trust is increased for agents that have proved themselves reliable in communication during a test interval. The increase of the level of trust is modeled as an

---

improvement. Notice that the improvement is determined based on recorded information, that is, every agent that will pass the test will have its trust level increased as soon as the rule with improvement is executed.

$$\begin{aligned} [agent(Ag), now(T)] &\Rightarrow G(m, n) \text{ reliable}(Ag) : \text{assert}(\text{rel\_ag}(Ag, T)) \\ \text{rel\_ag}(Ag, T) &\Rightarrow \text{true} : \text{increase\_trust\_level}(Ag) \end{aligned}$$

## 5 Related Work

In this section we discuss other related approaches to verification starting by those using declarative programming.

Alberti et al [21] addressed the problem of verifying system's specifications by using abductive logic programming. In particular, they proposed the SCIFF framework, an abductive logic programming language and proof system for the specification and runtime verification of interaction protocols. The authors developed an abductive proof-procedure (called g-SCIFF) which is used to prove properties at design time and to generate counter-examples of properties that do not hold. g-SCIFF is proven to be sound and complete with respect to its declarative semantics [21]. In [22] the g-SCIFF proof procedure is proved to be a valid alternative to other verification methods by experimentally comparing its performance with the one of some well-known model checker. The SCIFF proof-procedure is based on the notion of events, for example, sending a message or starting an action. Events are associated with time points. SCIFF uses integrity constraints ICs to model relations among events and expectations about events. Expectations are abducibles identified by the functors  $E$  (for positive expectations) and  $EN$  (for negative expectations). Events and time variables are constrained by ICs taking the form  $body \Rightarrow head$ . The  $body$  is a conjunction of happened events, expectations and constraints, whereas the  $head$  is a disjunction of conjunctions of positive and negative expectations. For example, the rule  $H(a, T) \Rightarrow E(b, T2) \wedge T2 \leq T + 300$  states that if  $a$  occurs, then the event  $b$  should occur no later than 300 time units after  $a$ . The SCIFF approach is related to our proposed approach in a number of ways. Most of our interval temporal connectives can be simply expressed by SCIFF rules except for example the *until* connective. Moreover, in our logic (as well as in METATEM logic) temporal connectives can be composed in several ways with both temporal and logical connectives, while this is not possible with  $E$  and  $EN$  abducibles. For example,  $E(a \wedge (EN(b, T), T2))$  cannot be expressed.

McIlraith et al consider the problem of planning and monitoring in dynamic and uncertain domains (cf. [23–25]). To address real world planning problems, they extend classical planning to incorporate procedural control knowledge and temporally extended user preferences into the specification of the planning problem. These preferences are somewhat similar to the METATEM connectives. For example, if a preference establishes *eventually*( $\varphi$ ), they want the planner to choose actions that will lead to the satisfaction of  $\varphi$ . However, their approach substantially differs from ours since we consider intervals over METATEM formulae.

Bauer et al [26, 27] consider runtime verification for realtime systems emitting events at dedicated time points. A logic suitable for expressing properties of such a



system is *timed lineartime temporal logic* (TLTL), which is a timed variant of LTL (originally introduced by Raskin in [28]). TLTL allows one to express typical bounded response properties, for example requiring that an event  $a$  occurs within three time units. Note that such a property can be expressed in LTL with then formula  $\varphi \equiv XXX a$ . However, as discussed in [26], this formulation presumes a direct correspondence of discrete time delays with incoming events (that is, every time point corresponds to an incoming event). In contrast, what we want to express is that the event  $a$  occurs after three time units regardless of how many other events occur in between. Following [29] the authors consider an TLTL logic that is a timed variant of LTL, called  $LTLe_c$ . Its syntax includes the two new atomic formulae:  $\triangleleft_a \in I$  and  $\triangleright_a \in I$ . The first asserts that the time since the event  $a$  has occurred the last time lies within then interval  $I$ . Analogously,  $\triangleright_a \in I$  asserts that the time until  $a$  occurs again lies within  $I$ . For example, the formula:

$$G ( request \rightarrow \triangleright_{acknowledge} \in [0, 5] )$$

means that if a request event arrives, then it must be handled with an acknowledge event within 5 time units. These formulae can be expressed in I-METATEM by means of its temporal operators in time intervals. The formula above can be expressed as:

$$[now(T)] \Rightarrow G ( request \Rightarrow F_{0, T+5} acknowledge ).$$

Thus, I-METATEM logic is at least as expressive as the TLTL logic. Our conjecture is that the two logics have the same expressive power. For example, the I-METATEM connective  $U_{m,n}$  can be represented in TLTL as follows:

$$\begin{aligned} \langle \sigma, i \rangle \models \varphi U_{m,n} \psi & \quad (i \leq m \leq n) \\ & \equiv \\ \langle \sigma, i \rangle \models \bigvee_{x=m}^n (\triangleright_{\psi} \in [x, x] \wedge I) \end{aligned}$$

where:

$$I = \begin{cases} true & \text{if } x - 1 < i \\ \bigwedge_{y=i}^{x-1} (\triangleright_{\varphi} \in [y, y]) & \text{otherwise} \end{cases}$$

## 6 Concluding Remarks

We have introduced an approach to the definition and the run-time verification of properties of agent behavior that has elements of novelty: in fact, we adopt a temporal logic with connectives defined on intervals in order to define and verify the run-time behavior of agents evolution; we are able to undertake suitable repairing actions based on the verification of properties and, as the underlying abstract agent model includes meta-level(s), these actions may imply modifications to the agent's knowledge base.

At the moment, the implementation of the approach has not been completed yet. Thus, we do not make any claim on its performance and complexity. Future work includes a full implementation of the approach, the development of suitable case-studies in significant application realms such as, e.g., ambient intelligence, and theoretical developments aimed at coping with challenging contexts, e.g., learning.

---

## References

1. Costantini, S., Tocchio, A.: About declarative semantics of logic-based agent languages. In Baldoni, M., Torroni, P., eds.: *Declarative Agent Languages and Technologies*. LNAI 3904. 106–123
2. Alferes, J.J., Brogi, A., Leite, J.A., Pereira, L.M.: Evolving logic programs. In: *Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf., JELIA 2002*. LNAI 2424, Springer-Verlag, Berlin (2002) 50–61
3. Costantini, S., Tocchio, A., Toni, F., Tsintza, P.: A multi-layered general agent model. In: *AI\*IA 2007: Artificial Intelligence and Human-Oriented Computing, 10th Congress of the Italian Association for Artificial Intelligence*. LNCS 4733, Springer-Verlag, Berlin (2007)
4. Costantini, S., Dell’Acqua, P., Pereira, L.M.: A multi-layer framework for evolving and learning agents. In M. T. Cox, A.R., ed.: *Proceedings of Metareasoning: Thinking about thinking workshop at AAAI 2008*, Chicago, USA. (2008)
5. Fisher, M., Bordini, R.H., Hirsch, B., Torroni, P.: Computational logics and agents: a road map of current technologies and future trends. *Computational Intelligence Journal* **23**(1) (2007) 61–91
6. Clarke, E.M., Lerda, F.: Model checking: Software and beyond. *Journal of Universal Computer Science* **13**(5) (2007) 639–649
7. Barringer, H., Rydeheard, D., Gabbay, D.: A logical framework for monitoring and evolving software components. In: *TASE ’07: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, Washington, DC, USA, IEEE Computer Society (2007) 273–282
8. Barringer, H., Fisher, M., Gabbay, D., Gough, G., Owens, R.: MetateM: A framework for programming in temporal logic. In: *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*. LNCS 430, Springer-Verlag (1989)
9. Fisher, M.: Metatem: The story so far. In Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E., eds.: *PROMAS*. LNCS 3862, Springer (2005) 3–22
10. Costantini, S., Tocchio, A.: The DALI logic programming agent-oriented language. In: *Logics in Artificial Intelligence, Proc. of the 9th European Conference, Jelia 2004*. LNAI 3229, Springer-Verlag, Berlin (2004)
11. Apt, K.R., Bol, R.: Logic programming and negation: A survey. *The Journal of Logic Programming* **19-20** (1994) 9–71
12. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: *Logic Programming, Proc. of the Fifth Joint Int. Conf. and Symposium*, MIT Press (1988) 1070–1080
13. Barklund, J., Dell’Acqua, P., Costantini, S., Lanzarone, G.A.: Reflection principles in computational logic. *J. of Logic and Computation* **10**(6) (2000) 743–786
14. J.Alferes, J., Brogi, A., Leite, J.A., Pereira, L.M.: An evolvable rule-based e-mail agent. In: *Procs. of the 11th Portuguese Intl.Conf. on Artificial Intelligence (EPIA’03)*. LNAI 2902, Springer-Verlag, Berlin (2003) 394–408
15. Bracciali, A., Demetriou, N., Endriss, U., Kakas, A., Lu, W., Mancarella, P., Sadri, F., Stathis, K., Terreni, G., Toni, F.: The KGP model of agency: Computational model and prototype implementation. In: *Global Computing: IST/FET International Workshop, Revised Selected Papers*. LNAI 3267. Springer-Verlag, Berlin (2005) 340–367
16. Kakas, A.C., Mancarella, P., Sadri, F., Stathis, K., Toni, F.: The KGP model of agency. In: *Proc. ECAI-2004*. (2004)
17. Stathis, K., Toni, F.: Ambient Intelligence using KGP Agents. In Markopoulos, P., Eggen, B., Aarts, E.H.L., Crowley, J.L., eds.: *Proceedings of the 2nd European Symposium for Ambient Intelligence (EUSAI 2004)*. LNCS 3295, Springer Verlag (2004) 351–362

- 
18. Tocchio, A.: Multi-Agent systems in computational logic. PhD thesis, Dipartimento di Informatica, Università degli Studi di L'Aquila (2005)
  19. Costantini, S., Tocchio, A.: A logic programming language for multi-agent systems. In: Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf., JELIA 2002. LNAI 2424, Springer-Verlag, Berlin (2002)
  20. Costantini, S., Tocchio, A.: Memory-driven dynamic behavior checking in logical agents. In: Electr. Proc. of CILC'06, Italian Conference of Computational Logic. (2006) URL: <http://cilc2006.di.uniba.it/programma.html>.
  21. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verifiable agent interaction in abductive logic programming: The SCIFF framework. ACM Trans. Comput. Log. **9**(4) (2008)
  22. Montali, M., Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Verification from declarative specifications using logic programming. In Banda, M.G.D.L., Pontelli, E., eds.: 24th Int. Conf. on Logic Programming (ICLP). LNCS 5366, Udine, Italy, Springer Verlag (December 2008) 440–454
  23. Baier, J., Bacchus, F., McIlraith, S.: A heuristic search approach to planning with temporally extended preferences. In: Proc. 20th Int. J. Conf. on Artificial Intelligence (IJCAI-07). (2007) 1808–1815
  24. Baier, J.A., Fritz, C., Bienvenu, M., McIlraith, S.: Beyond classical planning: Procedural control knowledge and preferences in state-of-the-art planners. In: Proc. 23rd AAAI Conf. on Artificial Intelligence (AAAI), Nectar Track. (2008) 1509–1512
  25. Fritz, C., McIlraith, S.A.: Monitoring policy execution. In: Proc. 3rd Workshop on Planning and Plan Execution for Real-World Systems. (2007) (at ICAPS07).
  26. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. Technical Report TUM-I0724, Institut für Informatik, Technische Universität München (December 2007)
  27. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In Arun-Kumar, S., Garg, N., eds.: Proc. 26th Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS). LNCS 4337, Berlin, Heidelberg, Springer-Verlag (2006)
  28. Raskin, J.F.: Logics, Automata and Classical Theories for Deciding Real Time. PhD thesis, Institut d'Informatique, FUNDP, Namur, Belgium (1999)
  29. D'Souza, D.: A logical characterisation of event clock automata. Int. J. Found. Comput. Sci. **14**(4) (2003) 625–640