Software Engineering
for
Answer Set Programming

Second International Workshop
14 September 2009, Potsdam, Germany

Marina De Vos
Torsten Schaub (Eds.)

Preface

Over the last ten years, answer set programming (ASP) has grown from a pure theoretical knowledge representation and reasoning formalism to a computational approach with a very strong formal backing. At present, ASP is seen as the computational embodiment of non-monotonic reasoning incorporating techniques of databases, knowledge representation, logic and constraint programming. ASP has become an appealing tool for knowledge representation and reasoning and thanks to the increasing efficiency of the implementations of ASP solvers, the field has now started to tackle many industrially-relevant applications.

Writing complex programs in any language is not an easy task, with ASP being no exception. Most of the modern popular programming languages have an abundance of tools and development methodologies to facilitate and improve the coding process. Given the differences in for example language design, execution, and application domains for languages such as Java and C++, the existing methodologies and tools that are available are mostly not suitable for ASP. Therefore development tools and software engineering methodologies specifically designed for ASP are required.

The SEA'09 workshop provides an international forum to discuss all software engineering issues the field currently faces or in the future will experience.

SEA'09 is the second first event in hopefully a long series of workshops. The first workshop was held in Tempe, Arizona, USA as a co-located workshop of LPNMR'07, one of the leading conferences in the area of logic programming and in particular ASP. For the second edition we have again co-located with LPNMR, this time organised in Potsdam, Germany.

Apart from the regular paper presentations, the workshop also welcomes Tran Cao Son from the New Mexico State University, US as an invited speaker. In the previous edition we organised the "ASP Language Forum" as a starting point for a discussion on the requirements and specification of input, output and intermediate languages for answer set solvers and grounders. This edition we plan a panel on "How to program in Answer Set Programming: Towards a Software Engineering Methodology" as a starting point for a general programming methodology for ASP.

Within these proceedings you can find the five papers that were accepted for publication by our programme committee and the abstract of our invited talk together with a related paper for reference.

The programme committee and organisers wish to thank all the authors who submitted papers, the panel members, the reviewers, all participants and everyone who contributed to the success of this workshop.

May 2007

Marina De Vos
Torsten Schaub
Organisers
SEA'07

Organisation

Executive Committee

Workshop Chairs:	Marina De Vos (University of Bath, UK) Torsten Schaub (University of Potsdam, Germany)

Programme Committee

Marcello Balduccini	(Kodak Research Labs)
Martin Brain	(University of Bath, UK)
Wolfgang Faber	(University of Calabria, Italy)
Martin Gebser	(University of Potsdam, Germany)
Enrico Pontelli	(New Mexico State University, USA)
Alessandra Russo	(Imperial College London, UK)
Tran Cao Son	(New Mexico State University, USA)
Hans Tompits	(Vienna University of Technology, Austria)
Richard Watson	(Texas Tech University, USA)
Stefan Woltran	(Technical University of Vienna, Austria)

Additional Referees

Johannes Oetsch

Table of Contents

Part I

Invited Speaker

On Building a Competitive Comformant Planner

Tran Cao Son

Department of Computer Science
New Mexico State University
tson@cs.nmsu.edu

1 Invited Abstract

In this talk, I will detail the development of CpA(H), a competitive conformant planner, that won the Conformant Planning Category in the International Planning Competition 2006. Lessons learned and the influence of logic programming in our development of the planner will also be discussed.

Conformant Planning with Disjunctive Initial States: Design and Development of an Efficient Planner

D-V. Tran, H-K. Nguyen, E. Pontelli, T.C. Son

Department of Computer Science
New Mexico State University
vtran | knguyen | epontell | tson@cs.nmsu.edu

Abstract. The paper illustrates the design and development of a competitive conformant planner. The planner builds on the theoretical foundations of approximation-based planning to enable a compact representation of the possible states. The novelty of the proposed approach is the realization that the description of the (incomplete) initial state is often based on *constraints* (e.g., expressed through PDDL's `or` and `oneof` clauses); the paper illustrates how such constraints can be reasoned upon to reduce the size of the search space. The reasoning process is implemented in the form of transformations of the problem specification within the planner. This, along with approximations and the use of combined heuristics, leads to enhanced efficiency and scalability, outperforming state-of-the-art conformant planners on several benchmark suites.

1 Introduction and Motivation

Conformant planning is the problem of finding a sequence of actions that achieves the goal from every possible initial state of the world [14]. One of the main difficulties encountered in the process of determining a conformant plan is the high degree of uncertainty, due to the potentially large number of possible initial states of the problems.

The *Planning Domain Definition Language (PDDL)* introduces two constructs to express incomplete knowledge about the initial state of the world: *mutual-exclusion* statements (expressed using `oneof`-clauses) and *disjunctive* statements (expressed using `or`-clauses). Frequently, `oneof`-clauses are used to specify the possible initial states and `or`-clauses are used to eliminate infeasible states. Because of this, the number of possible initial states depends mainly on the number and the size of the `oneof`-clauses—and these are often exponential in the number of constants present in the problem instances. For example, three out of six domains in the 2006 planning competition have this property (Table 1).

Instance	# Cons/States	Instance	# Cons/States
comm-15	$35/2^{16}$	coins-20	$17/9 \times 8^6$
comm-20	$85/2^{21}$	coins-25	$39/10^{20}$
comm-25	1402^{26}	coins-30	$45/10^{25}$
sortnet-10	$11/2^{11}$	sortnet-15	$2^{16}/16$

Table 1. Number of Constants/Possible Initial States

Effective methodologies and data structures are required to deal with the large number of possible initial states. Some conformant planners, such as POND [6] and KACMBP [7], employ a BDD representation of belief states, while others, such as CFF [4], adopt a CNF representation. These types of encodings avoid dealing directly with the exponential number of states, but they require extra work in determining the truth value of certain fluents after the execution of a sequence of actions in the initial belief state. For instance, CFF needs to make a call to a SAT-solver with the initial state and the sequence of actions; other planners need to recompute the BDD representation, which could also be an expensive operation. Observe that the problem of determining the truth value of a proposition after the execution of a single action in a belief state is co-NP complete [1].

An alternative approach to deal with the large number of possible initial states is used by the planners `cf2cs(ff)` and CPA [12, 17], and further investigated in their successors `t0` and CPA+ [13, 16]. This approach relies on an *approximation semantics* in reasoning with incomplete information [15]. The planners `cf2cs(ff)` and `t0` reduce the number of possible initial states to one by introducing additional propositions, transforming the original problem to a classical planning problem, and using FF, a classical planner [8], to find solutions. On the other hand, CPA and CPA+ reduce this number by dividing them into groups and using the intersection of each group as its representative during planning.

CPA+ and `t0` implement the idea of approximations differently. While CPA+ could be seen as a standard heuristic search forward planner, `t0` follows a translational approach. The performance of CPA+ depends on its heuristic function and its ability to approximate the initial belief state to a manageable set of partial states. On the other hand, the performance of `t0` largely depends on the performance of FF. `t0` was the winner of the 2006 planning competition.

In this paper, we describe the design and implementation of a competitive conformant planner. The proposed planner[1] expands the idea of approximation-based conformant planning, introducing novel techniques to significantly enhance efficiency and scalability. We explore the problem of engineering an approximation-based planner that can avail of modern data structures and heuristic functions. A cornerstone of our approach is viewing the description of the initial state not just as a passive characterization of a collection of states, but as a collection of constraints; by reasoning on such constraints, we discover ways to transforms the problem specification, enabling drastic reductions in the size of the search space. The resulting planner outperforms the state-of-the-art in conformant planning on large pool of benchmarks, including the problems from the latest planning competition.

2 Problem Representation

Following the notation in [12], we describe a *problem specification* as a tuple $P = \langle F, O, I, G \rangle$, where F is a set of propositions, O is a set of actions, I and O describe the initial state of the world and the goal respectively. A *literal* is either a proposition $p \in F$ or its negation $\neg p$. $\bar{\ell}$ denotes the complement of a literal ℓ and is defined by

[1] Named CPA(H) to recognize its roots in the CPA+ system.

$\bar{\ell} = \neg\ell$ where $\neg\neg p{=}p$ for $p{\in}F$. For a set of literals L, $\overline{L} = \{\bar{\ell} \mid \ell \in L\}$. A conjunction of literals is often represented by a set.

A set of literals X is *consistent* if there exists no $p \in F$ such that $\{p, \neg p\} \subseteq X$. A *state* s is a consistent and *complete* set of literals, i.e., s is consistent, and for each $p \in F$, either $p \in s$ or $\neg p \in s$. A *belief state* is a set of states. A *partial state* is a consistent set of literals. A *cs-state* is a set of partial states. A set of literals X satisfies a literal ℓ (resp. a set of literals Y) iff $\ell \in X$ (resp. $Y \subseteq X$).

Each action a in O is associated with a precondition ϕ (denoted by $pre(a)$) and a set of conditional effects of the form $\psi \rightarrow \ell$ (also denoted by $a : \psi \rightarrow \ell$), where ϕ and ψ are sets of literals and ℓ is a literal.

The initial state of the world is described by $I = I^d \cup I^o \cup I^r$ where I^d is a set of literals, I^o is a set of `oneof`-clauses—of the form $\texttt{oneof}(\phi_1, \ldots, \phi_n)$—and I^r is a set of `or`-clauses of the form $\texttt{or}(\phi_1, \ldots, \phi_n)$, where each ϕ_i is a set of literals. A `oneof`-clause indicates that the ϕ_i's are mutually exclusive, while an `or`-clause is a disjunctive normal form (DNF) representation of a formula. A set of literals X satisfies $\texttt{oneof}(\phi_1, \ldots, \phi_n)$ if there exists some $1{\leq}i{\leq}n$ s.t. $\phi_i \subseteq X$ and for every $j \neq i$, $1{\leq}j{\leq}n$, $\overline{\phi_j} \cap X{\neq}\emptyset$. X satisfies $\texttt{or}(\phi_1, \ldots, \phi_n)$ if there exists some $1{\leq}i{\leq}n$ s.t. $\phi_i \subseteq X$. $ext(I)$ denotes the set of all states satisfying I^d, every `oneof`-clause in I^o, and every `or`-clause in I^r. E.g., if $F{=}\{g, f\}$ and $I{=}\{g, \texttt{oneof}(f, \neg f)\}$ then $ext(I) = \{\{g, f\}, \{g, \neg f\}\}$.

G can contain literals or `or`-clauses. Given a `oneof`-clause or an `or`-clause o, we write $L \in o$ to denote that L is an element of o and $lit(o) = \bigcup_{L \in o}(L \cup \bar{L})$.

3 A Competitive Conformant Planner: Design

The planner, called CPA(H), is composed of two modules. The first module (*Preprocessor*) is a static analyzer that performs a number of transformations of the problem specification. Along with a grounder (which also applies standard simplifications, such as forward reachability), the preprocessor applies some novel transformations (`oneof`-clause combination and goal splitting) aimed at drastically reducing the size of the search space. The second module (*Planning engine*) is a heuristic search engine implementing forward planning.

3.1 Design of the Planning Engine

Theoretical Foundations Given a state s and an action a, a is executable in s if $pre(a) \subseteq s$. The set of effects of a in s, denoted by $e_a(s)$, is defined by: $e_a(s) = \{l \mid \psi \rightarrow l \text{ is an effect of } a, \psi \subseteq s\}$. The execution of a in a state s results in a successor state $succ(a, s)$ which is defined by: $succ(a, s) = s \cup e_a(s) \setminus \overline{e_a(s)}$ if a is executable in s; and $succ(a, s) = failed$, otherwise.

$succ$ is extended to define $succ^*$, which computes the result of the execution of an action in a belief state, as follows.

$$succ^*(a, S) = \begin{cases} \{succ(a, s) \mid s \in S\} \\ \quad \text{if } a \text{ is executable in every } s \in S \\ failed \quad \text{otherwise} \end{cases} \tag{1}$$

Finally, we can define the function $\widehat{succ}$ to compute the final belief state resulting from the execution of a plan:

- $\widehat{succ}([a_1, \ldots, a_n], S) = S$ if $n = 0$, and
- $\widehat{succ}([a_1, \ldots, a_n], S) = succ^*(a_n, \widehat{succ}([a_1, \ldots, a_{n-1}], S))$ if $n > 0$.

Several heuristic search-based conformant planners (e.g. CFF, POND), employ $\widehat{succ}$ in plan computation, using $S_0 = ext(I)$ as the initial belief state. An action sequence α is a *solution* of P iff $\widehat{succ}(\alpha, S_0) \neq failed$ and G is satisfied in every state belonging to $\widehat{succ}(\alpha, S_0)$.

The notion of approximation used in CPA+, `cf2cs(ff)`, and `t0` has been originally proposed in [15]. The original intuition behind approximation is to approximate sets of possible states by a single partial state—thus, reducing the complexity of reasoning w.r.t. using all possible states. It is characterized by a function ($succ_A$) that maps an action and a partial state to a partial state. The *possible effects* of a in a partial state δ are given by

$$pc_a(\delta) = \{l \mid \psi \rightarrow l \text{ is an effect of } a, \overline{\psi} \cap \delta = \emptyset\}. \tag{2}$$

The successor partial state from the execution of a in δ is defined by $succ_A(a, \delta) = (\delta \cup e_a(\delta)) \setminus \overline{pc_a(\delta)}$ if a is executable in δ; and $succ_A(a, \delta) = failed$, otherwise.

Similarly to $succ^*$ and $\widehat{succ}$, $succ_A$ can be extended to define $succ^*_A$ (mapping cs-states to cs-states) and $\widehat{succ_A}$ for computing the result of the execution of an action sequence starting from a cs-state. The notion of a solution of P w.r.t. the approximation is extended accordingly. Our planner uses $\widehat{succ^*_A}$ in its search for plans.

Observe that, in general, reasoning using approximations is incomplete. Completeness can be gained by identifying appropriate partitions $\{\Delta_1, \ldots, \Delta_k\}$ of $ext(I)$ such that $\bigcup_{i=1}^{k} \Delta_i = ext(I)$, $\Delta_i \cap \Delta_j = \emptyset$ for each $i \neq j$, and we have that for each formula φ and sequence of actions α, $\widehat{succ_A}(\alpha, \{\delta_1, \ldots, \delta_k\})$ entails φ iff $\widehat{succ}(\alpha, ext(I))$ entails φ, where δ_i is the intersection of the states in Δ_i. Research has been conducted to provide sufficient syntactical conditions to identify valid partitions—based on the identification of fluents that should be explicitly distinguished in different partitions (see, e.g., [16]).

Heuristic Search The $succ^*_A$ function is used in the context of a planning algorithm which implements forward planning using a traditional best-first heuristic search.

The proposed planner enables the user to choose among different heuristics; as discussed in the experimental section, we observed that sustained better performance can be achieved by using combinations of heuristics, improving the ability of discriminating between states in the priority queue (as employed in other systems as well [6]). The basic heuristics employed are:

- *The cardinality heuristic:* we prefer belief states that have a smaller cardinality. In other words, $h_{card}(\Sigma) = |\Sigma|$ where Σ is a belief state. Note that we use this heuristic in a forward fashion, and this is different from its use in [2, 5]. The intuition behind this is that planning with complete information is "easier" than planning with incomplete information and a lower cardinality implies a lower degree of uncertainty.

- *The relaxed graphplan heuristic:* for a belief state Σ, we define $h_{rgp}(\Sigma) = \sum_{\delta \in \Sigma} d(\delta)$, where $d(\delta)$ is the well-known sum heuristic value given that the initial state is $\delta \cup \{\neg p \mid p \in F,\ p \notin \delta,\ \neg p \notin \delta\}$ [11].
- *The number of satisfied subgoals:* denoted by $h_{gc}(\Sigma)$.

We investigate the following combination of these heuristics: $h_{css}(\Sigma) = (h_{card}(\Sigma), h_{gc}(\Sigma), h_{rgp}(\Sigma))$; heuristic measures are compared according to their lexicographic order.

3.2 Design of the Preprocessor

Standard Transformations Key to our analysis is the notion of *dependence* between actions and propositions—similar to the notion of dependence between actions and literals explored in [16]. We denote with P a planning problem.

Definition 1. *An action a* depends *on a literal ℓ if*

1. *$\ell \in pre(a)$, or*
2. *there exists an effect $a : \phi \rightarrow h$ in P and $\ell \in \phi$, or*
3. *there exits an action b that depends on ℓ and a depends on some of the effects of b, i.e., b depends on ℓ and there exists $b : \phi \rightarrow h$ such that a depends on h.*

By $preact(\ell)$ we denote the set of actions depending on ℓ.

Intuitively, the fact that a depends on ℓ indicates that the truth value of ℓ could influence the result of the execution of a. For a set of literals L, $preact(L) = \bigcup_{\ell \in L} preact(\ell)$.

Definition 2. *Two literals ℓ and ℓ' are* distinguishable *if $\ell \neq \ell'$ and there is no action that depends on both ℓ and ℓ', i.e., $preact(\ell) \cap preact(\ell') = \emptyset$.*

Obviously, the distinguishable relation is symmetric and irreflexive. Two set of literals L_1 and L_2 are distinguishable if $preact(L_1) \cap preact(L_2) = \emptyset$.

The dependence between a literal and an action, often used in *reachability analysis*, is defined next.

Definition 3. *A literal ℓ* depends *on an action a if* (1) *$a : \psi \rightarrow \ell$ is in P; or* (2) *there exists an action b such that $b : \psi \rightarrow \ell$ is in P and there exists some ℓ' in ψ or in $pre(b)$ s.t. ℓ' depends on a. We denote with $deps(a)$ the set of literals that depend on a.*

Intuitively, ℓ depends on a implies that ℓ may be achieved by executing a. $postact(\ell)$ denotes the set of actions which ℓ depends on, i.e., $postact(\ell)$=$\{a \mid \ell \in deps(a)\}$.

Definition 4. *Two literals ℓ and ℓ' are* independent *if $\ell \neq \ell'$ and there exists no action that both ℓ and ℓ' depend on, i.e., $postact(\ell) \cap postact(\ell') = \emptyset$.*

The preprocessor starts its operations with a number of basic normalization steps, aimed at reducing the number of propositions and the number of actions present in the problem specification. In particular, it implements a traditional *forward reachability* simplification (to detect propositions whose value cannot be changed and actions that cannot be triggered w.r.t. the initial state) and a symmetrical *goal relevance* (removing actions that cannot contribute to the goal).

Combination of `oneof`-clauses The idea of this technique is based on the *non*-interaction between actions and propositions in different sub-problems of a conformant planning problem. We illustrate this idea in the next example.

Example 1. Let $P = \langle \{f, g, h, p, i, j\}, O, I, G\rangle$ where $I = \{\texttt{oneof}(f, g), \texttt{oneof}(h, p), \neg i, \neg j\}$, $G = i \wedge j$, and $O = \{\, a : f \rightarrow i \; c : h \rightarrow j \; b : g \rightarrow i \; d : p \rightarrow j \,\}$. It is easy to see that the sequence $\alpha = [a, b, c, d]$ is a solution of P. Furthermore, the search should start from the initial belief state consisting of the four states:

$$\{f, \neg g, h, \neg p, \neg i, \neg j\} \; \{\neg f, g, h, \neg p, \neg i, \neg j\}$$
$$\{f, \neg g, \neg h, p, \neg i, \neg j\} \; \{\neg f, g, \neg h, p, \neg i, \neg j\}$$

Let P' be the problem obtained from P by replacing I with I', where $I' = \{\texttt{oneof}(f \wedge h, g \wedge p), \neg i, \neg j\}$.

We can see that α is also a solution of P'. Furthermore, each solution of P' is also a solution of P. This transformation in interesting since the initial belief state now consists only of two states: $\{f, \neg g, h, \neg p, \neg i, \neg j\}$ and $\{\neg f, g, \neg h, p, \neg i, \neg j\}$. I.e., the number of states in the initial belief state that a conformant planner has to consider in P' is 2, while it is 4 in P. This transformation is possible because the set of actions that are "activated" by f and g is disjoint from the set of actions that are "activated" by h and p, i.e., $preact(\{f, g\}) \cap preact(\{h, p\}) = \emptyset$.

The above example shows that different `oneof`-clause can be combined into a single `oneof`-clause, which effectively reduces the size of the initial state that a planner needs to consider in its search for a solution. Theoretically, if the size of the two `oneof`-clauses in consideration is m and n, then it is possible to achieve a reduction in the number of possible partial states from $m \times n$ to $\max(m, n)$. Since in many problems the size of the `oneof`-clauses increases with the number of objects, being able to combine the `oneof`-clauses could provide a significant advantage for the planner.

Definition 5. *Let $P = \langle F, O, I, G\rangle$ be a planning problem. Two* `oneof`*-clauses o_1 and o_2 are* combinable *if (*i*) $lit(o_1) \cap lit(o_2) = \emptyset$; and (*ii*) $lit(o_1)$ is distinguishable from $lit(o_2)$. where $lit(o)$ denote the union of the set of literals occurring in o and its complements*

For example, the `oneof`-clauses in Ex. 1 are combinable. Let $o_1 = \texttt{oneof}(L_1, \ldots, L_n)ando_2 = \texttt{oneof}(S_1, \ldots, S_m)$. Assume that $n \geq m$. A *combination* of o_1 and o_2, denoted by $o_1 \oplus o_2$ (or $o_2 \oplus o_1$) is the clause

$$\texttt{oneof}(L_1 \wedge S_1, \ldots, L_m \wedge S_m, L_{m+1} \wedge S_1, \ldots, L_n \wedge S_1)$$

Intuitively, a combination of o_1 and o_2 is a `oneof`-clause whose elements are pairs obtained by composing one element of o_1 with exactly one element of o_2.

Proposition 1. *Let $P = \langle F, O, I, G\rangle$ be a planning problem, where G is a conjunction of literals and o_1 and o_2 are two combinable* `oneof`*-clauses in P. Let $P' = \langle F, O, I', G\rangle$, where I' is obtained from I by replacing o_1 and o_2 by $o_1 \oplus o_2$. Every solution of P' is also a solution of P and vice versa.*

Observe that the above proposition may not hold if P contains disjunctive goals, as shown next.

Example 2. Let $P = \langle \{q, g, h, p, i, j\}, O, I, G\rangle$ where

$$I = \{\texttt{oneof}(h, g), \texttt{oneof}(p, q), \neg i, \neg j\} \text{ and } G = \texttt{or}(i, j)$$

and O consists of $a : p, \neg q \rightarrow i$, $c : p, q \rightarrow i$, $b : g, \neg h \rightarrow j$, and $d : g, \neg h \rightarrow j$.

It is easy to check that $\texttt{oneof}(h, g)$ and $\texttt{oneof}(q, p)$ are combinable. Let P' be the problem obtained from P by replacing I with $I' = \{\texttt{oneof}(g \wedge q, h \wedge p), \neg i, \neg j\}$. Then, $[a, b]$ is a solution of P' but not a solution of P.

The *combinable* notion can be generalized as follows.

Definition 6. *A set of* `oneof`*-clauses* $\{o_1, \ldots, o_k\}$ *is combinable if* o_i *and* o_j *are combinable for each* $1 \leq i \neq j \leq k$.

Let $\oplus(o_1, \ldots, o_k)$ be the shorthand for $(((o_1 \oplus o_2) \oplus \ldots) \oplus o_k)$. Proposition 1 can be generalized as follows.

Proposition 2. *Let* $P = \langle F, O, I, G\rangle$ *be a planning problem, where* G *is a conjunction of literals. Let* $\{o_1, \ldots, o_k\}$ *be a combinable set of* `oneof`*-clauses in* P. *Let* $P' = \langle F, O, I', G\rangle$, *where* I' *is obtained from* I *by replacing* $\{o_1, \ldots, o_k\}$ *with* $\oplus(o_1, \ldots, o_k)$. *We have that each solution of* P' *is a solution of* P *and vice versa.*

We implemented a greedy algorithm, whose running time is polynomial in the size of P, for detecting sets of combinable `oneof`-clauses and replacing them with their corresponding combination. This is possible since testing if ℓ and ℓ' are distinguishable can be done in polynomial time in the size of P, and the number of pairs that need this test is quadratic in the number of propositions.

Goal Splitting Reducing the size of the initial state only helps the planner to start the search. It does not necessarily imply that the planner can find a solution. In this section, we present another technique, called *goal-splitting*, which can be used in conjunction with the combination of `oneof` to deal with large planning problems. This technique can be seen as a variation of the goal ordering technique in [9] and it relies on the notion of dependence proposed in Def. 4. The key idea is that if a problem P contains a subgoal whose truth value cannot be negated by the actions used to reach the other goals, then the problem can be decomposed into smaller problems with different goals, whose solutions can be combined to create a solution of the original problem. This is illustrated in the following example.

Example 3. Consider the problem P of Example 1. It is easy to see that the two goals i and j are independent and P can be decomposed into two sub-problems $P_1 = \langle F, O_1, I, i\rangle$ and $P_2 = \langle F, O_2, I_2, j\rangle$ where $O_1 = \{a : f \rightarrow i,\ \ b : g \rightarrow i\}$ and $O_2 = \{c : h \rightarrow j,\ \ d : p \rightarrow j\}$ with the following property: if α is a solution of P_1 and β is a solution of P_2 where $I_2 = \widehat{succ_A}(\alpha, I_1)$, then $\alpha; \beta$ is a solution of P.[2]

Let us start with a definition capturing the condition that allows the splitting of goals.

[2] $\alpha; \beta$ denotes the concatenation of two sequences of actions.

Definition 7. *Let $P = \langle F, O, I, G\rangle$ be a planning problem and let $\ell \in G$. We say that ℓ is G-separable if, for each $\ell' \in G \setminus \{\ell\}$ we have that $\bar{\ell}$ and ℓ' are independent.*

Proposition 3. *Let $P = \langle F, O, I, G\rangle$ be a planning problem and let ℓ be G-separable. Let $P_\ell = \langle F, postact(\ell), I, \ell\rangle$ and α be a solution of P_ℓ. Let $P_{G\setminus\{\ell\}} = \langle F, postact(G \setminus \{\ell\}), I', G \setminus \{\ell\}\rangle$, where $I' = \widehat{succ_A}(\alpha, I)$, and β be a solution of $P_{G\setminus\{\ell\}}$. Then, $\alpha;\beta$ is a solution of P.*

The proof is trivial, since $postact(G \setminus \{\ell\})$ does not contain any action that can make $\bar{\ell}$ true.

On the other hand, it is easy to see that not every plan of P can be split into two parts α and β such that α is a solution of P_ℓ and β is a solution of $P_{G\setminus\{\ell\}}$. We can prove, however, that for each plan γ of P, there is a plan α for P_ℓ and a plan β for $P_{G\setminus\{\ell\}}$ such that γ is a permutation of $\alpha;\beta$. This provides a weak form of completeness.

We note that the splitting proposed in Prop. 3 can be improved by also splitting the propositions and initial states into different theories. We have implemented a generalized version of Prop. 3 to split a problem into a sequence of problems. This implementation runs in polynomial time in the size of P.

4 Implementation Considerations

The preprocessor has been implemented as a Prolog program. The program maps the input PDDL theory to a collection of Prolog clauses. This mapping nicely avoids the need of explicitly grounding the problem specification a priori. The transformations are implemented as fixpoint computations on the Prolog clauses representing the problem specification.

The planning engine has been implemented as a C++ program, running on a Linux (Athlon 64, 4Ghz), gcc 4.2.1 version, with STL library. A partial state is implemented as a set (a basic data structure in STL) of literals.

The engine implements a best first search over the search space of cs-states. Each cs-state is a data structure consisting of a set of partial states, a plan to reach that cs-state, and the heuristic values: h_{card}, h_{gc}, and h_{rpg}. A modified version of the algorithm presented in [10] is implemented to compute h_{rpg}.

$succ^*_A$ is used to compute the next cs-state. A hash table (resp. priority queue) is used to store the visited (resp. unvisited) cs-states. A special module is developed to compute the initial cs-state, which consists of the set of initial partial states. Each initial partial state δ satisfies the following conditions: a) $\{p, \neg p\} \cap \delta \neq \emptyset$ for each proposition p appears in I; b) $I^d \subseteq \delta$; c) for each $\texttt{oneof}(\phi_1, \ldots, \phi_n) \in I^o$, there exists an i such that $\phi_i \subseteq \delta$ and for all $j \neq i$, $\overline{\phi_j} \cap \delta \neq \emptyset$; d) for each $\texttt{or}(\phi_1, \ldots, \phi_n) \in I^r$, there exists an i such that $\phi_i \subseteq \delta$; e) δ is consistent. Choosing to implement the initial cs-state as a set (of the set of initial partial states) makes the computation of the successor cs-state (the result of $succ^*_A$) easier. The main disadvantage of this choice is that the size of the initial cs-state can be exponential in the size of the number of object constants in the problem. This is the reason why reducing the size of the initial cs-state is critical to our planner.

5 Experimental Evaluation

The experimental evaluation has been performed using several benchmark suites—i.e., problems from the IPC-5 (or I5) and the IPC-6 (I6) planning competitions, challenging (C) problems proposed in [13], and several other domains from previous planning competitions. The benchmark suite for each domain is listed in Table 5. Due to lack of space, we omit the detailed description of the actual benchmarks, that have been drawn from the existing literature. We also report only a subset of the complete experimental results due to limited space (full results will be made available through a linked technical report). Time is in seconds, TO denotes time-out (30 min), AB denotes out of memory, and BM denotes benchmark suite.

Table 2 summarizes some results aimed at evaluating the impact of the transformations; the three columns indicate execution times and the length of the first plan found; we can observe that the improvement in performance is often significant; it occasionally comes at the price of a longer plan.

Problem	NoTransf.	`oneof`	goal-splitting
coins-05	0.02/13	0.024/19	0.03/14
coins-10	1.33/35	0.66/129	1.52/43
coins-15	/AB	13.54/391	/AB
coins-20	/AB	33.39/621	/AB
comm-05	0.90/48	0.31/60	0.20/35
comm-10	61.14/87	3.98/190	22.40/65
comm-15	/AB	15.82/327	/AB
uts-05	34.43/115	34.43/115	1.06/43
uts-10	36.14/130	36.14/130	21.21/87
uts-20	53.88/286	53.88/286	35.91/138
uts-30	152.07/177	152.07/177	18.06/74
dispose-4-2	1.30/72	0.395/59	0.78/76
dispose-4-3	43.00/93	0.36/78	19.14/111
dispose-8-2	412.70/272	10./234	368.03/284
dispose-8-3	/AB	487.28/1187	/AB
push-4-2	1.11/58	1.00/133	1.84/96
push-4-3	34.66/265	2.04/251	46.68/141
push-8-2	282.55/444	128.21/979	/AB
push-8-3	/AB	454.80/1811	/AB

Table 2. Impact of `oneof`-combination and goal-splitting

Table 3 reports the execution times of the planner using different heuristics. The column t_S indicates the time for preprocessing. Although the results are mixed for small instances, $h_{css(\Sigma)}$ comes ahead for large instances.

Table 4 compares the execution times and plan lengths for the proposed planner and other three state-of-the-art systems for conformant planning (`t0`, CFF, and POND, all run with default parameters according to their documentation). Table 5 reports the number of instances each planner can solve.

6 Discussion

We now discuss the question of whether the proposed techniques can be applied to other planning systems.

Observe that a combination of several `oneof`-clauses is a `oneof`-clause, whose elements are conjunctions of literals, which can be represented by a set of `oneof`-clauses

Problem	t_S	CPA(H) h_{card}	CPA(H) h_{gc}	CPA(H) h_{rgp}	CPA(H) $h_{css(\Sigma)}$
bwl-02	0.13	0.262/26	/TO	0.064/33	0.217/41
bwl-03	0.18	7.668/198	/TO	2.219/145	73.188/312
coins-15	0.49	7.449/423	15.874/551	0.387/191	5.920/329
coins-20	0.66	25.51/756	24.413/722	0.902/195	20.239/481
comm-15	3.64	0.496/96	0.148/95	0.094/95	0.124/95
comm-20	141	2.739/240	0.993/239	0.994/239	0.976/239
comm-25	1081	18.74/389	3.355/389	3.674/389	3.249/389
sortnet-05	0.11	0.054/13	16.35/13	0.036/12	0.023/12
sortnet-10	0.24	12.537/39	/TO	3.270/39	4.205/39
sortnet-15	0.52	/TO	/TO	/TO	313.044/65
uts-10	0.93	21.21/87	/TO	17.567/89	9.602/80
uts-20	0.89	35.91/138	/TO	12.596/150	36.314 /125
uts-30	0.86	18.06/74	/TO	346.467/103	31.609/94
d-4-3	.61	. 3.94/288	4.83/369	0.95/314	3.80/288
d-8-1	47.3	1.95/143	124.71/1229	24.12/725	1.86/137
d-8-2/C	53.6	386.68/1328	600.77/2298	135.8/1494	391.44/1328
d-10-1	285	7.65/213	/AB	148.7/1489	7.69/213
push-4-3	.65	40.45/1176	340.23/959	1.0/225	288.21/847
push-8-1/C	52.4	3.824/184	/AB	27.25/468	3.97/184
push-10/C	304	23.04/414	/AB	/AB	23.48/414
1-d-4-3	.69	24.8/64	/AB	81.57/108	25.12/64
1-d-8-1	47.9	9.91/340	/AB	/AB	9.22/340
1-d-10-1	288	36/568	/AB	/AB	33.81/568
lng-8-1-1	50.4	2.45/94	284.73/5547	/AB	1.78/94
lng-8-2-1	57.8	0.97/94	31.79/563	68.45/287	0.97/94
lng-8-1-2	59.5	73.14/125	/AB	/AB	72.65/125

Table 3. Comparison between heuristics

and a set of disjunctions eliminating the unwanted combinations. We tested the effectiveness of the `oneof`-simplification on POND and CFF. Table 6 shows the results of our experiment with the planner POND in the `comm` and `coins` domains where the `oneof`-combination is applicable. As we can see, the performance of POND improves in these problems, and the improvement is more significant when the size of the problem is large. This technique helps POND to scale up but its impact is not as great as in CPA(H): POND can solve more problems in the `comm` domain. The problems `comm-16.0` and `comm-16.1` have more objects than `comm-16` but less than `comm-17`. For CFF, we did not observe improvements using the modified problems.

We also experimented with using the preprocessor to perform goal splitting and produce modified PDDL files that can be processed by other planners. For example, CFF is unable to solve the problems from p21 to p30 of the `coins` domain. The difficulty in this domain lies in the large number of elevators and coins. The goal-splitting technique divides the problem into a sequence of sub-problems, each dealing with one coin but still has all elevators, enabling CFF to solve these problems. We observed that CFF spent most of the time finding the solution for the first problem. This is reasonable, since the location of the elevators is initially unknown, and some locations will be known at the end of the first solution. The planning time of CFF for coins p21, p25 and p30 is 24.48 2.13 and 68.09 (secs) accordingly.

These initial experiments show that the proposed techniques could be useful for other planners.

7 Conclusions

In this paper, we presented the complete design and implementation of an efficient conformant planner, called CPA(H). The planner builds on recently developed techniques for conformant planning using approximations; it introduces several novelties, including a preprocessing module to transform the problem specification, leading to significantly reduced search spaces, and the ability to explore the search space with different

Problem	t_S	CPA(H) $h_{css(\Sigma)}$	t0	CFF	POND
block-03	0.13	0.22/41	/NA	/AB	1.02/26
block-04	0.18	73.19/312	/NA	/AB	1379/111
coins-10	0.18	0.14/81	0.09/26	1.02/38	5.26/28
coins-15	0.49	5.92/329	0.26/81	7.35/79	/TO
coins-20	0.66	20.24/481	0.32/108	38.19/143	/TO
comm-15	3.64	0.12/95	0.19/110	0.22/95	1662/110
comm-20	141	0.98/239	0.86/278	13.33/239	/ TO
comm-25	1081	3.25/389	3.99/453	109.49/389	/ TO
sortnet-10	0.24	4.21/39	NA	NA	/TO
sortnet-15	0.52	313.04/65	NA	NA	/TO
adder-01	8.2	1.29/3	/NA	/AB	/AB
UTS-cycle-03	0.17	1.08/3	/NA	/NA	1.99/3
UTS-cycle-04	0.29	23.88/6	/NA	/NA	48.19/6
forest-02	2.4	23/84	0.37/12	0.03/17	1.33/15
forest-04	7.7	/TO	1.58/60	/TO	71.17/62
Rao-keys-02	0.22	0.04/32	/NA	0.08/33	0.29/21
Rao-keys-03	0.37	3.84/152	/NA	25.09/101	4.51/68
dispose-8-1	47.3	1.86/137	27.85/291	423.25/226	/AB
dispose-8-2	53.6	391/1328	118.46/422	/AB	/AB
dispose-10-1	285	7.69/213	275.08/474	/AB	/AB

Table 4. Comparison between systems (NA: not applicable)

Domain/BM	# of instances	CPA(H) $h_{css(\Sigma)}$	t0	CFF	POND
block/I6	4	3	0	1	3
adder/I5	4	1	0	0	0
coins/I5	30	20	20	20	10
comm/I5	25	25	25	25	16
sortnet/I5	15	15	0	0	6
uts/I5	30	30	30	30	24
UTS-cycle/I6	27	2	0	0	2
forest/I6	9	1	8	1	2
Rao-keys/I6	29	2	0	2	2
dispose/C	90	62	50	41	8
push/C	90	29	33	27	12
1-dispose/C	90	24	7	1	8
look-n-grab/C	81	63	20	21	9

Table 5. Number of problems solved in different domains

heuristic functions. The result is a conformant planner that has been shown to be highly competitive with the state-of-the-art in the field. In particular, CPA(H) outperforms all existing systems on the problems from the latest International Planning Competition.

The results presented in this paper confirm the strength of using approximations for conformant planning, the possibility of implementing approximation-based planning in an efficient and scalable system, and the scope for improvement that can be achieved via transformation of problem specifications.

Problem	Orig/Modified	Problem	Orig/Modified
comm-15	1662/4.89	coins-5	0.52/0.51
comm-16	TO/57.93	coins-10	5.54/1.63
comm-16.0	TO/124.05	coins-15	17.13/17.95
comm-16.1	TO/267.39	coins-10	143/126

Table 6. Impact of `oneof`-Combination on POND (Orig/ Modified: Time for solving the original/modified problem)

The future developments of this project include exploring whether alternative methods for the internal implementation of cs-states (e.g., OBDD) can further enhance performance. We also plan to expand the reasoning component of the preprocessor, to obtain additional simplifications of the problem specifications (e.g., detecting symmetries between fluents).

Acknowledgement

The authors are partially supported by the NSF grants IIS-0812267, CBET-0754525, CNS-0220590, and CREST-0420407.

References

1. C. Baral et al. Computational complexity of planning and approximate planning in the presence of incompleteness. *AIJ*, 122:241–267, 2000.
2. P. Bertoli et al. Heuristic search + symbolic model checking = efficient conformant planning. *IJCAI*, pages 467–472, 2001.
3. B. Bonet and B. Givan. Results of the conformant track of the 5th planning competition, 2006. `http://www.ldc.usb.ve/~bonet/`.
4. R. Brafman and J. Hoffmann. Conformant planning via heuristic forward search: A new approach. *ICAPS-04*, pages 355–364, 2004.
5. D. Bryce and S. Kambhampati. Heuristic Guidance Measures for Conformant Planning. *ICAPS-04*, pages 365–375, 2004.
6. D. Bryce et al. Planning Graph Heuristics for Belief Space Search. *JAIR*, 26:35–99, 2006.
7. A. Cimatti et al. Conformant Planning via Symbolic Model Checking and Heuristic Search. *Artificial Intelligence Journal*, 159:127–206, 2004.
8. J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *JAIR*, 14:253–302, 2001.
9. J. Hoffmann et al. Ordered landmarks in planning. *JAIR*, 22:215–278, 2004.
10. D. Long and M. Fox. . Efficient Implementation of the Plan Graph in STAN. *JAIR*, 10, 1999.
11. X.L Nguyen et al. Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search. *AIJ*, 135:73–123, 2002.
12. H. Palacios and H. Geffner. Compiling Uncertainty Away: Solving Conformant Planning Problems Using a Classical Planner. *AAAI*, 2006.
13. H. Palacios and H. Geffner. From Conformant into Classical Planning: Efficient Translations that may be Complete Too. *ICAPS-07*, 2007.
14. D.E. Smith and D.S. Weld. Conformant graphplan. *AAAI*, pages 889–896, 1998.
15. T.C. Son and C. Baral. Formalizing sensing actions - a transition function based approach. *Artificial Intelligence*, 125(1-2):19–91, January 2001.
16. T.C. Son and P.H. Tu. On the Completeness of Approximation Based Reasoning and Planning in Action Theories with Incomplete Information. *KRR*, 2006.
17. T.C. Son, P.H. Tu, M. Gelfond, and R. Morales. Conformant Planning for Domains with Constraints. *AAAI*, pages 1211–1216, 2005.

Part II

Research Papers

A Preference Meta-Model for Logic Programs with Possibilistic Ordered Disjunction

Roberto Confalonieri, Juan Carlos Nieves, and Javier Vázquez-Salceda

Universitat Politècnica de Catalunya
Dept. Llenguatges i Sistemes Informàtics
C/ Jordi Girona Salgado 1-3
E - 08034 Barcelona
{confalonieri,jcnieves,jvazquez}@lsi.upc.edu

Abstract This paper presents an approach for specifying user preferences related to services by means of a preference meta-model, which is mapped to logic programs with possibilistic ordered disjunction following a Model-Driven Methodology (MDM). MDM allows to specify problem domains by means of meta-models which can be converted to instance models or other meta-models through transformation functions. In particular we propose a preference meta-model that defines an abstract preference specification language allowing users to specify preferences in a more friendly way using models. We also present a meta-model for logic programs with possibilistic order disjunction. Then we show how we conceptually map the preference meta-model to logic programs with possibilistic ordered disjunction by means of a mapping function.

1 Introduction

Recently with the adoption of both Service-Oriented Architectures (SOA) and Web services as growing trend for building distributed applications, services can be world-widely advertised and accessed. In this context, service discovery and selection play an important role *w.r.t.* the search and selection of the most suitable services users are looking for. With the rapidly growing number of services that are becoming available, users will be offered in fact with a choice of functionally similar services, which increase the need of enhancing traditional discovery and selection processes with the possibility for the users to express preferences about and relevant to certain services.

On the other hand, expressing and reasoning about user preferences is a complex and challenging task, as preferences cannot be generally explicitly expressed because of the large number of possible alternatives. Nonmonotonic logics have shown to be a potent knowledge representation formalism to reason about preferences [4]. Several extensions of the basic formalism of Answer Set Programming (ASP) have been proposed to model preferences [6], showing how nonmonotonic logics constitute an effective way of resolving indeterminate solutions, reasoning in terms of preferred answer sets of a logic program. Unfortunately, nonmonotonic logics by themselves are not flexible enough and not well designed for mod-

eling orderings on belief sets specified in terms of preferences on their elements [4].

Logic programs with ordered disjunction (or LPODs) offer one way to overcome this problem as they permit to explicitly represent preference information directly into head rules [2]. In this way, the language can capture user qualitative preferences by means of disjunction rules, represent choices among different alternatives and specify a preference order between the answer sets through some comparison criteria. However, in some scenarios the preference information can be subject to uncertainty and preference-aware reasoning methods that can handle uncertainty are needed [5]. For this reason in [5] the authors have proposed an extension of the semantics of logic programs with ordered disjunction in order to cope with the degree of uncertainty in the reasoning process. In particular, they have defined a possibilistic semantics for capturing *logic programs with possibilistic ordered disjunction* (or LPPODs) which is close to the proof theory of possibilistic logic and answer set semantics.

ASP has become quite popular in knowledge representation problems as it is based on solid theoretical foundation, it is expressively rich, and its semantics and computational properties well understood today. Moreover many efficient ASP solvers such as *dlv* [8] and *smodels* [14] are available. As such ASP has called the attention from the industry, and points out to be a promising knowledge representation method in many application areas. Nevertheless, ASP has not been applied to Service-Oriented Architectures yet. One of the main obstacles towards the adoption of nonmonotonic preference representation methods is determined by the lack of an ASP programming environment. Writing correct and efficient ASP programs is in fact a difficult task and users are required to have the sufficient expertise to encode real problems in ASP. For this reason one of the open issues of ASP is the development of ASP programming environments and friendly interfaces [10].

In this paper we propose an approach for specifying user preferences related to services by means of a preference meta-model which we map to LPPODs following a Model-Driven Methodology (MDM). MDM allows to specify problem domains by means of meta-models which can be converted to instance models or other meta-models through transformation functions. We propose a preference meta-model that defines an abstract preference specification language allowing the users to specify preferences about services in a more friendly way using models. We also present a meta-model for logic programs with possibilistic order disjunction showing how we conceptually map the preference meta-model to the model of LPPODs by means of a mapping function.

The paper is organised as follows. In Section 2 the syntax and semantics of logic programs with possibilistic ordered disjunction are presented and the main characteristics of the MDM approach are described. In Section 3 we propose a model-driven framework for capturing preferences in SOA and a preference meta-model is described. In Section 4 we present the meta-model for logic programs with possibilistic ordered disjunction and a conceptual mapping. In Section 5 we describe how MDM can be applied in the modeling of a simple user preference

scenario. Finally in Section 6 we discuss our approach, draw some conclusions and outline future work.

2 Background

In this section we introduce the reader with some basic concepts *w.r.t.* the syntax and semantics of logic programs with possibilistic ordered disjunction and Model-Driven Methodology.

2.1 Logic Programs with Possibilistic Ordered Disjunction

Logic programs with possibilistic ordered disjunction (LPPODs) are logic programs with ordered disjunction with possibilistic values added to each rule [5]. The syntax of a logic program with possibilistic ordered disjunction is based on the syntax of ordered disjunction rules [2] and of possibilistic logic [7].

LPPODs Syntax: A signature $\mathcal{L}$ is a finite set of elements called atoms. Atoms negated by $\neg$ will be called extended atoms. The concept of atom will be used without paying attention if it is an extended atom or not. A *possibilistic atom* is a pair $p = (a, q) \in \mathcal{A} \times \mathcal{Q}$ where $\mathcal{A}$ is a set of atoms and $(\mathcal{Q},\leq)$ a finite lattice[1]. The projection $*$ to any possibilistic atom p is defined as follows: $p^* = a$. Given a set of possibilistic atoms M, the generalization of $*$ over M is defined as: $M^* = \{p^* \mid p \in M\}$. Given a lattice $(\mathcal{Q},\leq)$, a possibilistic ordered disjunction rule r is of the form:

$$\alpha : c_1 \times \ldots \times c_n \leftarrow b_1, \ldots, b_m, \ \mathit{not}\ b_{m+1} \ldots, \ \mathit{not}\ b_{m+k}$$

where $\alpha \in \mathcal{Q}$ and $c_i (1 \leq i \leq n)$, $b_j (1 \leq j \leq m+k)$ are atoms. Sometimes a possibilistic ordered disjunction clause is denoted as: $\alpha : c_1 \times \ldots \times c_n \leftarrow \mathcal{B}^+, \ \mathit{not}\ \mathcal{B}^-$ where $\mathcal{B}^+ = \{b_1, \ldots, b_m\}$ and $\mathcal{B}^- = \{b_{m+1}, \ldots, b_{m+k}\}$. The projection $*$ for a possibilistic ordered disjunction rule r, is $r^* = c_1 \times \ldots \times c_n \leftarrow \mathcal{B}^+, \ \mathit{not}\ \mathcal{B}^-$. It can be observed that the ordered disjunction clause r^* is an ordered disjunction clause as was defined in [2]. $n(r) = \alpha$ is a necessity degree representing the certainty level of the information described by r. A possibilistic constraint C is of the form $\mathcal{TOP}_{\mathcal{Q}} :\leftarrow \mathcal{B}^+, \ \mathit{not}\ \mathcal{B}^-$, where $\mathcal{TOP}_{\mathcal{Q}}$ is the top of the lattice $(\mathcal{Q},\leq)$ and $\leftarrow \mathcal{B}^+, \ \mathit{not}\ \mathcal{B}^-$ is a constraint as in standard ASP [1]. Please notice that any possibilistic constraint must have the top of the lattice $(\mathcal{Q},\leq)$. This restriction is motivated by the fact that, like constraints in standard Answer Set Programming, the purpose of the possibilistic constraint is to eliminate possibilistic models. Hence, it is assumed that there is no uncertainty about the information captured by a possibilistic constraint. As in possibilistic ordered disjunction rules, the projection $*$ for a possibilistic constraint C is $C^* =\leftarrow \mathcal{B}^+, \ \mathit{not}\ \mathcal{B}^-$.

A *logic program with possibilistic ordered disjunction* (LPPOD) is a tuple of the form $P := \langle (Q, \leq), N \rangle$ such that N is a finite set of possibilistic ordered

[1] Only finite lattices are considered.

disjunction rules and possibilistic constraints. The generalization of $*$ over P is defined as follows: $P^* := \{r^* \mid r \in N\}$. Notice that P^* is an ordered disjunction logic program.

LPPODs Semantics: Before defining the possibilistic semantics for capturing LPPODs, basic operations between sets of possibilistic atoms and a relation of order between them are introduced.

Definition 1. *Given $\mathcal{A}$ a finite set of atoms and ($\mathcal{Q}$,$\leq$) a lattice, $\mathcal{PS} = 2^{\mathcal{A}\times\mathcal{Q}}$ is considered as the finite set of all the possibilistic atom sets induced by $\mathcal{A}$ and $\mathcal{Q}$. Let $A, B \in \mathcal{PS}$, the operators $\sqcap$, $\sqcup$ and $\sqsubseteq$ can be defined as follows:*

$$\begin{aligned} A \sqcap B &= \{(x, GLB\{q_1, q_2\}) | (x, q_1) \in A \ \wedge (x, q_2) \in B\} \\ A \sqcup B &= \{(x, q) | (x, q) \in A \text{ and } x \notin B^*\} \cup \{(x, q) | x \notin A^* \text{ and } (x, q) \in B\} \cup \\ &\quad \{(x, LUB\{q_1, q_2\}) | (x, q_1) \in A \text{ and } (x, q_2) \in B\}. \\ A \sqsubseteq B &\iff A^* \subseteq B^*, \text{ and } \forall x, q_1, q_2, (x, q_1) \in A \wedge (x, q_2) \in B \text{ then } q_1 \leq q_2. \end{aligned}$$

The semantics of LPPODs is close to the proof theory of possibilistic logic and answer set semantics. As in answer set semantics definition, the possibilistic semantics is defined based on a syntactic reduction.

Definition 2 (Reduction $r_\times^M$). *Let $r = \alpha : c_1 \times \ldots \times c_n \leftarrow \mathcal{B}^+,\ not\ \mathcal{B}^-$ be a possibilistic ordered disjunction clause and M be a set of atoms. The $\times$-possibilistic reduct $r_\times^M$ is defined as follows:*

$$r_\times^M := \{\alpha : c_i \leftarrow \mathcal{B}^+ | c_i \in M \text{ and } M \cap (\{c_1, \ldots, c_{i-1}\} \cup \mathcal{B}^-) = \emptyset\}$$

Definition 3 (Reduction $P_\times^M$). *Let $P = \langle (Q, \leq), N\rangle$ be a LPPOD and M be a set of atoms. The $\times$-possibilistic reduct $P_\times^M$ is defined as follows:*

$$P_\times^M = \bigcup_{r \in N} r_\times^M$$

Observe that the program $P_\times^M$ is a possibilistic positive extended logic program.[2] Once a LPPOD P has been reduced by a set of possibilistic atoms M, it is possible to test whether M is a possibilistic answer set of the program P by considering the following definition.[3]

Definition 4 (Possibilistic answer set). *Let $P = \langle (Q, \leq), N\rangle$ be a LPPOD and M be a set of possibilistic atoms such that M^* is an answer set of $(P_\times^{M^*})^*$. M is a possibilistic answer set of P if and only if $P_\times^{M^*} \vdash_{PL} M$ and $\nexists M' \in \mathcal{PS}$ such that $M' \neq M$, $P_\times^{(M')^*} \vdash_{PL} M'$ and $M \sqsubseteq M'$.*

By the original (no possibilistic) ordered disjunction rule definition, it is possible to represent preferences among possibilistic answer sets by considering degrees of satisfaction denoted as $deg_M(r)$ and defined by the following definition.

[2] A positive program is a program without negation as failure atoms.

[3] $\vdash_{PL}$ denotes the inference under possibilistic logic.

Definition 5 (Rule Satisfaction Degree). *Let M be a possibilistic answer set of a LPPOD P. Then M satisfies the rule r*

$$\alpha : c_1 \times \ldots \times c_n \leftarrow b_1, \ldots, b_m, \textit{ not } b_{m+1} \ldots, \textit{ not } b_{m+k}$$

- *to degree 1 if $b_j \notin M^*$ for some j $(1 \leq j \leq m)$, or $b_i \in M^*$ for some i $(m+1 \leq i \leq m+k)$,*
- *to degree j $(1 \leq j \leq n)$ if all $b_l \in M^*$ $(1 \leq l \leq m)$, $b_i \notin M^*$ $(m+1 \leq i \leq m+k)$, and $j = min\{r \mid c_r \in M^*, 1 \leq r \leq n\}$.*

To distinguish between preferred possibilistic answer sets, the satisfaction degree of a possibilistic answer set M *w.r.t.* a rule, denoted by $deg_M(r)$, provides a ranking of the possibilistic answer sets of a LPPOD, and a preference order on the possibilistic answer sets can be obtained by means of a comparison criteria. In [5] the authors have proposed three criteria for comparing possibilistic answer sets, respectively *possibilistic cardinality*, *possibilistic inclusion* and *possibilistic Pareto*, which are the possibilistic version of the original criteria of [2].
The set of possibilistic atoms M satisfying a degree i is defined as follows:

Definition 6. *Let M be a set of possibilistic atoms and P be a LPPOD. Then $M^{i,\alpha}(P) = \{r \in P \mid deg_M(r) = i \textit{ and } n(r) \geq \alpha\}$.*

Given a set of possibilistic atoms M, $n(M)$ is defined as $min\{\alpha \mid (a, \alpha) \in M\}$. Three preference relations can be defined. The possibilistic version of cardinality-based preference can be defined as follows:

Definition 7. *Let M_1 and M_2 be possibilistic answer sets of a LPPOD P. M_1 is possibilistic cardinality-preferred to M_2, $(M_1 >_{pc} M_2)$ iff $\exists$ i such that $\mid M_1^{i,\alpha}(P) \mid > \mid M_2^{i,\alpha}(P) \mid$ and $\forall j < i$, $\mid M_1^{j,\alpha}(P) \mid = \mid M_2^{j,\alpha}(P) \mid$, where $\alpha = min\{n(M_1), n(M_2)\}$.*

The inclusion-based preference is defined as:

Definition 8. *Let M_1 and M_2 be possibilistic answer sets of a LPPOD P. M_1 is possibilistic inclusion-preferred to M_2, $(M_1 >_{pi} M_2)$ iff $\exists$ k such that $M_2^{k,\alpha}(P) \subset M_1^{k,\alpha}(P)$ and $\forall$ $j < k$, $M_1^{j,\alpha}(P) = M_2^{j,\alpha}(P)$, where $\alpha = min\{n(M_1), n(M_2)\}$.*

Lastly, the possibilistic Pareto-based preference is:

Definition 9. *Let M_1 and M_2 be possibilistic answer sets of a LPPOD P. M_1 is possibilistic pareto-preferred to M_2, $(M_1 >_{pp} M_2)$ iff $\exists$ $r \in P$ such that $deg_{M_1}(r) < deg_{M_2}(r)$, and $\nexists r' \in P$ such that $deg_{M_1}(r') > deg_{M_2}(r')$, and $n(r) \geq min\{n(M_1), n(M_2)\}$.*

One interesting characteristic of LPPODs is that they provide a mean to represent preferences among problem solutions and allow to represent preferences which can depend on incomplete knowledge. As LPPODs are based on extended nonmonotonic logic, incomplete information can be expressed by means of default negation.

2.2 Model-Driven Methodology

Model-Driven Engineering (MDE) refers to the systematic use of models as primary artefacts throughout the Software Engineering (SE) development process. The defining characteristic of MDE is the use of models to represent the important artefacts in a system [13]. Each of the models in a MDE system is constructed from a language specified in a meta-model, which captures the concepts and relationships of the language in a structured and regular form. In relation to these meta-models, the models can then be stored, manipulated and transformed to other models, and to implementation artefacts.

A Model-Driven Methodology (MDM) to development is generally based on the Model Driven Architecture (MDA) [19], an initiative by the Object Management Group (OMG)[4] which specifies a framework of open standards and related technologies. The framework is built upon the metamodel foundation in order to enable a standard specification and interoperability mechanism for tools. So systems and applications are formalized with metamodel descriptions and are visualized by models as metamodel instantiations. Actual code implementations are created automatically by applying predefined transformations from source models to target models and implementation languages.

In the context of this paper, MDD specifies the user preference meta-model (Section 3.3) upon which a *user preference editor* can be created, allowing the modeling and instantiation of corresponding models. By the specification of a meta-model for logic programs with possibilistic ordered disjunction (Section 4.1) and a transformations function it is possible to (semi-)automatically translate the user preference abstract representation to the formalism of logic programs with possibilistic ordered disjunction (Section 4.2).

3 Preferences in Service-Oriented Architectures

The use of nonmonotonic reasoning about preferences in Service-Oriented Architectures is a new and unexplored field. We believe that the applicability of preference and reasoning methods to the services' domain can enhance the service discovery and selection processes and assist the user in services' searches. In order to reuse existing works about service oriented technologies and nonmonotonic preference handling methods for preference representation and reasoning we propose a model-driven framework that attempts to glue these approaches together.

3.1 Model-Driven Framework

Figure 1 shows the model-driven framework we are proposing. The diagram depicts the main components of the framework, and shows the relations between user preferences and services. The meta-model provides a domain independent and technology independent representation of user preferences about services.

[4] `http://www.omg.org/`

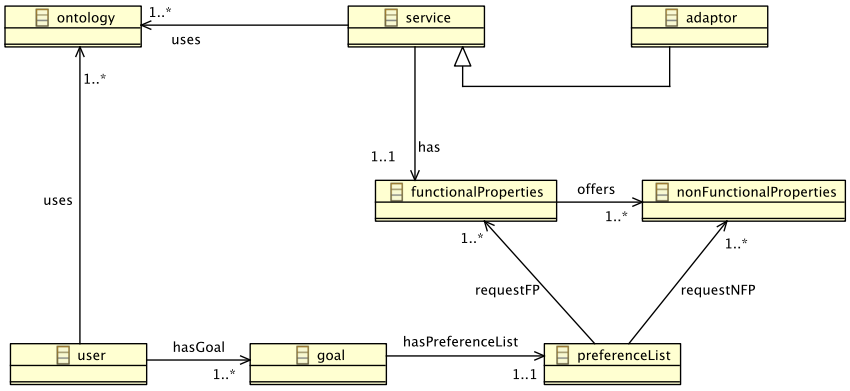

Figure 1. Model-Driven Framework for User Preferences in SOA

The framework is inspired by the Web Service Modeling Framework [9] which has been recently adopted in WSMO [15], although we introduce new concepts covering the relation between user preferences and services. The main component of the framework are: *goals user preferences*, *services*, *ontologies*, and *adaptors*:

- **Services** offer specific functionalities to users, and are described by *functional* and *non-functional properties*. Functional descriptions of services consist in service *input* and *output* and *pre* and *post-conditions*. Non-functional properties are usually related to service usage or domain dependent properties. Services are semantically described through an *ontology*.
- **Goals and User Preferences**: a *goal* is the user-centered view of a service usage. Normally users have specific tasks they want to accomplish and goals are specifications of a desired state of outcomes. Goals are accompanied by a *preference list* which represents the user preferences for the service. Such preferences *request functional* and *non-functional properties*. For instance let us imagine a user is interested in getting a map of restaurants, and she has a set of preferences about the type of restaurants, the map and the cost. Her goal could be to get a service that takes as input an object of type Restaurant and as output type Map, while preferences could be *"I prefer a high resolution map than a low resolution"* or *"I prefer to spend 1 euro than 2 euros, if the service response is fast"*. A goal usually consists of a functional description of the objectives users want to achieve using a Web service. In this sense a user goal is a *hard-constraint w.r.t.* a service functionality. User preferences are requirements the user wants over achieving a goal. They may include Quality of Services (QoS) metrics or domain dependent properties. From this point of view preferences are *soft-constraints w.r.t.* the properties ot the service that fulfills a certain goal.

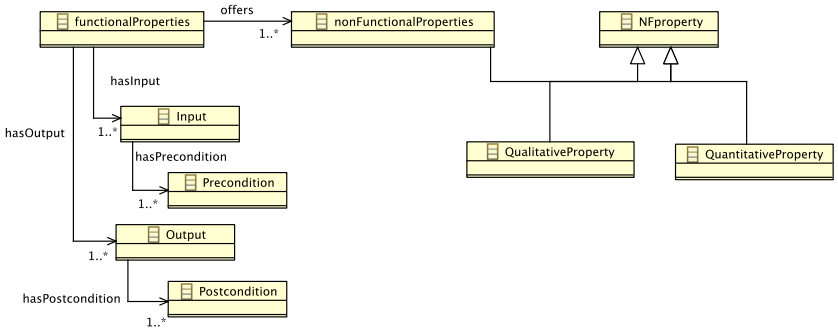

Figure 2. Service Properties Meta-model

- **Service Adaptors** address the handling of heterogeneities occurring between elements that shall interoperate by resolving mismatches between different used terminologies (data level), on communicative behavior between services (protocol level), and on the business process level.
- **Ontologies** provide the formal semantics for the terminology used within all other framework components. We expect to have a framework ontology which will consist in a *service ontology* and *domain ontologies* specific for domain applications.

In order to express preferences about and relevant to services, we need to consider the properties *requested* by the user and the properties *offered* by the service.

3.2 Service Properties Meta-Model

From a provider perspective a service can expose different offered properties associated with the same functionality to address different business requirements (*e.g.* speedy and slow service at different price). We assume that functional descriptions are provided in terms of *input* and *output*. *Pre-* and *post-condition* are constraints about input and output respectively. Offered properties consist in a set of *non-functional properties* which can be *qualitative* and *quantitative* whose values are defined in a *domain ontology* (Figure 2). Non-functional properties can be seen as a kind of service configuration which the provider *offers* for the service usage.

3.3 User Preference Meta-Model

Figure 3 shows the user preference meta-model. From a user perspective, required properties can be considered as a set of constraints on the requested services. To

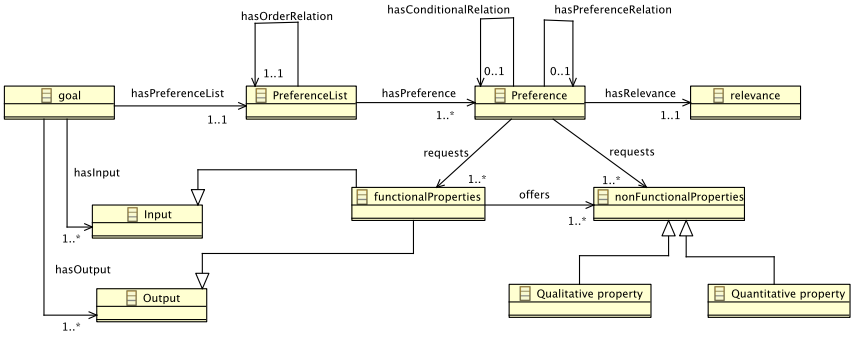

Figure 3. User Preference Meta-model

collect sets of preferences we consider a *preference list* to associate a goal with a set of *preferences*. A *goal* is described by *input* and *output* which specify strong constraints *w.r.t.* a service that fullfills that goal. A preference list is an ordered list of preferences specified by the user, where such order represents a *preference order* between sets of desired properties. Desired properties can be viewed along two dimensions: *functional* and *non-functional*. Functional properties are preferences about input and output of the goal, while non-functional preferences are related to service usage.

To manage qualitative and quantitative properties (*e.g.* Restaurant and Cost), two classes of properties are introduced, *quantitative* and *qualitative properties* respectively. All the properties' values are concepts of the *domain ontology* (*e.g.* Restaurant, Map) or data types (String, Integer *etc*). Preferences may be associated with a degree of *relevance w.r.t.* the preference rules or non-functional properties of a service. A *Preference* has a *preference relation* to be able to specify a preference order (*e.g. "I prefer a map with higher resolution to a lower one"*), and a conditional preference relation to be able to capture *conditional preferences* between properties *e.g.* (*"if the cost of the service is not high I prefer a high map resolution"*).

4 Model Transformations

As we do not want to stick to a particular formalism, language or technology during the solution specification, we have defined a meta-model describing how user preferences and service properties should be specified in a general way. The advantage to have a meta-model is to have a specification general enough which can be then translated to other meta-models and models (representing target languages or formalism) through transformations [18].

Generally speaking, a model transformation takes as input a model conforming to a given meta-model and produces as output another model conforming to

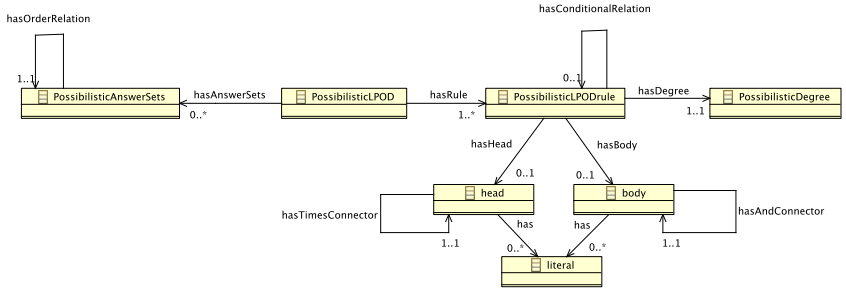

Figure 4. Meta-Model for Logic Programs with Possibilistic Ordered Disjunction

a given meta-model. The process of translating a model to another is specified by a mapping algorithm. The main advantage of this approach is that, from the same meta-model specification, several mappings to different models can be done through different mapping functions [12].

We define a transformation t of a preference model M_1 according to the preference meta-model *P_MM* to a LPPOD model M_2 according to the LPPODs meta-model *LPPOD_MM* as

$$t : M_{1_{P\_MM}} \rightarrow M_{2_{LPPOD\_MM}}$$

To be able to (informally) define a the transformation function t we first need a meta-model which describes the formalism of LPPODs.

4.1 Meta-Model for LPPODs

A proposal of a meta-model for this class of logic programs is shown in Figure 4. A *possibilistic LPOD program* consists of a finite set of *possibilistic LPOD rules* associated with a *possibilistic degree*. Each rule *has* a *head* and a *body*, where in the body a preference relation is specified by means of the logical connector $\times$ (times). Outcomes of LPPOD are *possibilistic answer sets* and a *relation order* between them can be obtained applying the comparison criteria defined in Section 2.1.

4.2 A Mapping to LPPODs

According to MDD the preference meta-model *P_MM* and LPPOD meta-model *LPPOD_MM* dependencies are formulated with model driven mappings (relations). The mappings are specified with a transformation language, among the corresponding elements of the *P_MM* and *LPPOD_MM* meta-models shown in Figure 3 and Figure 4 respectively. As illustrative example the mappings `preferenceListToLPPOD` and `preferenceToPossibilisticLPODRule` have been specified.

The transformation process can be initiated from the `preferenceListToLPPOD` mapping that converts user preferences in a LPPOD. The rule in turn applies a mapping between preference and preference rules of a LPPOD.

```
mapping preferenceListToLPPOD(in pl:P_MM::PreferenceList,
  inout lppod:LPPOD_MM::PossibilisticLPOD) {
 var preferenceRules := pl.preference -> collect(p|map
  preferenceToPossibilisticLPODRule(p,pl,preferenceRules));
 lppod.possibilisticLPODrule := preferenceRules;
}
mapping preferenceToPossibilisticLPODRule(in p:P_MM::Preference,
  in: pl:P_MM::PreferenceList,
  inout: pr:LPPOD_MM::PossibilisticLPODrule) {
 pr.possibilisticDegree := p.relevance;
 pr.head := p.functionalProperties -> collect(fp|map
  functionalPropertiesToHead(fp,p.preferenceOrder));
 pr.body := p.NFproperties -> collect(nfp|map
  nfPropertiesToBody(nfp));
}
mapping functionalPropertiesToHead(in fp:P_MM::functionalProperty,
  in po:P_MM:PreferenceRelation) {
 ...
}
mapping nfPropertiesToBody(in nfp:P_MM::NFPropertyPreference) {
 ....
}
```

5 Applying Model-Driven Methodology to User Preference Modeling

The preference meta-model presented in Section 3.3 can be instantiated to different domain problems and used for expressing user preferences about services in an abstract way. In particular it can be generated by means of a *preference editor*. The meta-model in fact can be specified by the Eclipse Ecore specification. Once the meta-model is available, a preference editor can be implemented using the EMF tools of the Eclipse Platform.[5] The model generated by the editor can be mapped to an instantiated model of a LPPOD by means of the transformation function t (Figure 5).

For example let us consider a user looking for a recommendation service, that takes as input *restaurants* and returns a *map*. She can have a goal where Restaurant is the Input and Map is the Output. She can have preferred values about the input and output of the goal, be undecided about the type of restaurants, and be looking for specific non-functional properties such as the cost of the service

[5] `http://www.eclipse.org/modeling/emf`

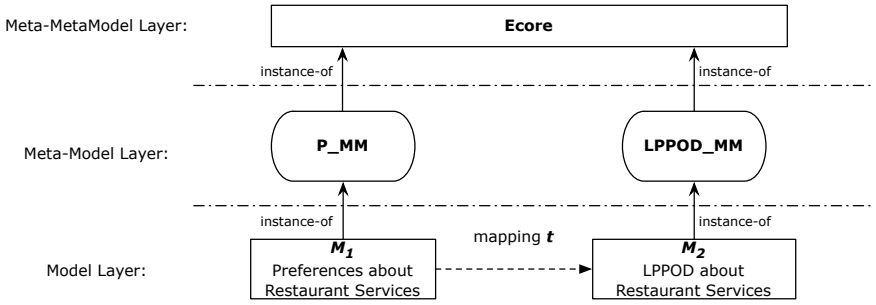

Figure 5. MDD applied to User Preference Modeling

and the map resolution. The request of a personalized service according to her preferences can be expressed as a list of preference such as:

- she prefers Italian to Mexican restaurants
- she prefers a higher map resolution to a lower one if the cost of the service is not > 2 euros (high) and the time of response is not ≥ 0.5 sec (slow).
- she prefers a lower map resolution if the service cost is > 2 euros (high) and time of response is ≥ 0.5 sec (slow).

Let us imagine we have the model M_1 for the scenario described above. By the transformation function t we can then generate the model of a LPPOD M_2. A further step (not shown here) is represented by a code generation function that converts M_2 to the syntax of LPPODs. The code generation results in the LPPOD code shown in the following example.

Example 1. Let $P = \langle (Q, \leq), N \rangle$ be a LPPOD expressing the user preferences about restaurant maps. First of all we define the lattice $(\mathcal{Q},\leq)$ to specify an ordered set of relevance degrees. We consider the lattice $\mathcal{Q} = (\{0, 0.1, 0.2, \ldots, 1\}, \leq)$, and $\leq$ the standard relation between rational numbers. Let N be the set of possibilistic ordered disjunction rules expressing the user preferences about restaurant maps generated by the transformation function t. One possible way to encode the user preferences in the syntax of a LPPOD is:[6]

$r_1 = \mathbf{1} : rest(italian) \times rest(mexican).$
$r_2 = \mathbf{1} : map(high) \times map(low) \leftarrow not\ cost(\_, high), not\ response(\_, slow).$
$r_3 = \mathbf{1} : map(low) \times map(high) \leftarrow cost(\_, high).$
$r_4 = \mathbf{1} : map(low) \times map(high) \leftarrow response(\_, slow).$
$r_5 = \mathbf{1} : \ \leftarrow map(low), map(high).$

Please notice that whenever the α values are equal to 1, the LPPOD behaves like an ordered disjunction program where the possibilistic values of the atoms

[6] The predicates cost(_,high) and response(_,slow) are qualitative predicates for the quantitative value *w.r.t.* the cost and the response time. Such predicates can be built by specific rules in the logic program.

in the answer sets are the top of the lattice. Therefore we can see that there are four possibilistic answer sets satisfying the program,

$M_1 = \{(rest(italian), 1), (map(high), 1)\}$
$M_2 = \{(rest(italian), 1), (map(low), 1)\}$
$M_3 = \{(rest(mexican), 1), (map(high), 1)\}$
$M_4 = \{(rest(mexican), 1), (map(low), 1)\}$

and according to their satisfaction degrees they have the following preference order: $M_1 >_{pp} M_2$, $M_1 >_{pp} M_3$, M_2 and M_3 are equally preferred and $M_3 >_{pp} M_4$. Such a order can be exploited to select the best service that accomplish the user goal. Each of the possibilistic answer set in fact can be used to build a service search where the atoms are used as input parameters for a matchmaker.[7] The matchmaking process returns a list of candidate services *w.r.t.* the possibilistic answer sets used. Non-functional service properties can be converted as a set of possibilistic facts $\mathcal{PF}_i$, where the necessity-value degrees corrispond to normalized non-functional properties values (*e.g.* $0.5 : cost(s_1, high)$) and added to a new LPPOD P_i, such that $P_i = P \cup \mathcal{PF}_i \cup C_i$.[8] The possibilistic answer sets of the new generated LPPODs P_i (one for each original M_i) are candidate service solutions where the possibilistic values drawn the selection of the best service *w.r.t.* the user preferences.

6 Conclusions

In this paper we have presented an approach for assisting the user in expressing preferences about services' searches following the Model-Driven Methodology. We have defined a preference meta-model which allows to represent user preferences about services without being bound to a specific implementation technology. We have conceptually shown how it is possibile to translate preference models to the model of LPPODs. The advantage of having a model for LPPODs is that LPPODs' code can be (semi-)automatically generated by means of a transformation function t.

The proposed preference meta-model provides in fact a flexible way to capture user preferences and represents the basic artefact through which different problem domains can be instantiated. The preference meta-model representation can ease the development of a preference editor which allows users to express preferences relevant to services in a friendly way. This abstract representation can be mapped to LPPODs which, based on ordered disjunction programs, are a flexible way to represent and reason about user preferences.

Concerning related works on user preference representation about services, in most of the existing approaches preferences have been studied in the context of

[7] The matchmaker is a component which is able to perform a syntactic or semantic matching of a user goal against service descriptions.

[8] C_i is a possibilistic constraint that forces the solutions of each P_i to be relevant only *w.r.t.* M_i.

Web services composition [11,17,16]. For instance, in [16] user preferences specified using Conditional Preference Networks (CP-Nets) are used to improve the quality of generated compositions. In [11,17] an augmented version of the logic programming language Golog is used to specify and to integrate user preferences into Web service composition. Although the use of user preferences related to services is not new, our proposed work differs with the cited works on at least two aspects: a) we use a preference-aware and uncertainty-aware preference representation language as part of the user preference selection process; b) we incorporate the use of this language as part of a MDD methodology according to which implementation details will be trasparent to the user. The meta-models we are proposing in fact can improve the time of development of LPPODs and the preference meta-model represents the first step towards a preference editor implementation. Our approach can be generalised to be used in other ASP-based formalisms in order to ease the knowledge modeling and the reuse of existing knowledge. Moreover relations between the original formalism of logic programs with ordered disjunction (LPODs) and other preference handling methods such as CP-Nets have been already explored in [3] showing that a further mapping is feasible.

Interesting issues for future work are the refinement of the preference and LPPODs meta-models and the specification of the transformation function t in a more formal way using the ATL transformation language.[9] Currently we are implementing our approach using the Eclipse Platform and the tools of the Ecore framework. As soon as we have the Ecore meta-models we can start the implementation of the preference editor to model practical domain problems. Although not related to the paper itself, it is worthy to mention that we are also studying the implementation of the solver for LPPODs.

Acknowledgements This work has been funded mainly by the European Commission Framework 7 funded project ALIVE (FP7-215890). Javier Vázquez-Salceda's work has been also partially funded by the Ramón y Cajal program of the Spanish Ministry of Education and Science. The opinions of the authors do not reflect the opinions of the European Commission. We thank anonymous referees for the valuable comments.

References

1. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
2. G. Brewka, I. Niemelä, and T. Syrjänen. Logic Programs with Ordered Disjunction. *Computational Intelligence*, 20(2):333–357, 2004.
3. G. Brewka, I. Niemelä, and M. Truszczyński. Answer Set Optimization. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 867–872. Morgan Kaufmann Publishers, 2003.

[9] `http://www.eclipse.org/m2m/atl/`

4. G. Brewka, I. Niemelä, and M. Truszczyński. Preferences and Nonmonotonic Reasoning. *AI Magazine*, 29(4):69–78, 2008.
5. R. Confalonieri, J. C. Nieves, and J. Vázquez-Salceda. Logic Programs with Possibilistic Ordered Disjunction. Technical Report LSI-09-19-R, Universitat Politècnica de Catalunya - LSI, 2009.
6. J. Delgrande, T. Schaub, H. Tompits, and K. Wang. A classification and Survey of Preference Handling Approaches in Nonmonotonic Reasoning. *Computational Intelligence*, 20(2):308–334, 2004.
7. D. Dubois, J. Lang, and H. Prade. Possibilistic Logic. In D. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming Volume 3: Nonmonotonic Reasoning and Uncertain Reasoning*, pages 439–513. Oxford University Press, Oxford, 1994.
8. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System dlv: Progress Report, Comparisons and Benchmarks. In L. S. A.G. Cohn and S. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning*, pages 406–417. Morgan Kaufmann, 1998.
9. D. Fensel and C. Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137, 2002.
10. N. Leone. Logic Programming and Nonmonotonic Reasoning: From Theory to Systems and Applications. In *Proceedings of 9th International Conference on Logic Programming and Nonmonotonic Reasoning*, page 1, 2007.
11. S. Mcilraith and T. C. Son. Adapting Golog for Programming the Semantic Web. In *In Fifth International Symposium on Logical Formalizations of Commonsense Reasoning*, pages 195–202, 2001.
12. A. Metzger. *Model-Driven Software Development*, chapter A Systematic Look at Model Transformations, pages 19–33. Computer Science. Springer Berlin Heidelberg, 2005.
13. J. B. Nicolas, N. Farcet, J. M. Jézéquel, B. Langlois, and D. Pollet. Reflective Model Driven Engineering. In *Proceedings of the 6th International Conference on the Unified Modeling Language*, LNCS, pages 175–189. Springer, 2003.
14. I. Niemelä and P. Simons. Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal LP. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR97)*, pages 421–430. Springer-Verlag, 1997.
15. D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1), 2005.
16. G. Santhanam, S. Basu, and V. Honavar. On Utilizing Qualitative Preferences in Web Service Composition: A CP-net Based Approach. In *IEEE Congress on Services - Part I*, pages 538–544, July 2008.
17. S. Sohrabi, N. Prokoshyna, and S. A. McIlraith. Web Service Composition Via Generic Procedures and Customizing User Preferences. In *International Semantic Web Conference*, volume 4273 of *LNCS*, pages 597–611. Springer, 2006.
18. A. U. Stephen Mellor, Kendall Scott and D. Weise. *MDA Distilled: Principles of Model-Driven Architecture*. Addison Wesley, 2004.
19. T. Weis, A. Ulbrich, and K. Geihs. Model Metamorphosis. *IEEE Software*, 20(5):46–51, 2003.

A Framework for Programming with Module Consequences*

Wolfgang Faber[1] and Stefan Woltran[2]

[1] University of Calabria, Italy
wf@wfaber.com
[2] Vienna University of Technology, Austria
woltran@dbai.tuwien.ac.at

Abstract. We present a framework which allows to combine answer-set programs in a way that consequences (rather than answer sets themselves) of programs can be used as input to other programs. Situations in which such a composition of programs is required appear in many practical application problems. So far, to deal with such problems, multiple calls to answer-set solvers were usually indispensable, as a direct ASP encoding is often much less obvious. In addition, we provide a technique for compiling such frameworks into a single ASP program which consequently can be evaluated by a single call to an answer-set solver. Our approach relies on the recently introduced concept of manifold programs which make use of weak constraints to identify consequences of programs.

1 Introduction

In the last decade, *Answer-Set Programming* (ASP) [1, 2], also known as A-Prolog [3, 4], has emerged as a declarative programming paradigm. ASP is well suited for modelling and solving problems which involve common-sense reasoning, and has been fruitfully applied to a wide variety of applications including diagnosis, data integration, configuration, and many others. This development was fueled by the efficiency of the latest tools for processing ASP programs (so-called ASP solvers) which reached a state that makes ASP applicable for problems of practical importance [5]. The most frequent use of ASP is to compute answer sets (usually stable models) of a logic program from which the solutions of the problem encoded by the program can be obtained.

A somewhat neglected aspect of ASP are its capabilities in terms of consequence relations (or, more general, using queries over answer sets), which are firmly rooted in the tradition of nonmonotonic reasoning. Different to classical settings, in nonmonotonic reasoning there is no canonical consequence relation—the two most studied ones are brave and cautious consequence (sometimes also termed as brave and cautious reasoning); the former is also known as credulous or possible reasoning, the latter is also referred to as skeptical or certain reasoning. In the context of ASP, one is usually interested in a subset of the atomic brave or cautious consequences, which corresponds to a

* This work was supported by the Vienna Science and Technology Fund (WWTF), grant ICT08-028, and by M.I.U.R. within the Italia-Austria internazionalization project "Sistemi basati sulla logica per la rappresentazione di conoscenza: estensioni e tecniche di ottimizzazione".

generalization of query answering for databases. In this sense, ASP can also be seen as an evolution of Datalog, a logical database query language.

As an example scenario, let us consider a problem stemming from database systems. A database is inconsistent, if a given database instance does not satisfy some of the constraints imposed. One could argue that the creation of inconsistent databases should be inhibited, but it is also obvious that this is not always possible: For instance, when integrating data, that is, whenever only partitions of the data are maintained in a locally consistent state (for example due to permissions or physical distribution), the merged data is not guaranteed to be consistent. Still, one would like to work with such data.

An attractive approach to dealing with inconsistent data is to consider minimal repairs, that is, considering minimal modifications of the data that establish a consistent state. ASP has been successfully employed for specifying and computing minimal repairs (see, e.g., [6]). In general, there is no unique minimal repair, and the usual workaround is to take a conservative approach and consider those parts of the database which hold in each minimal repair. In the ASP setting, this neatly corresponds to considering the cautious consequences of the program encoding the database repairs.

However, as mentioned earlier, support for consequence relations is somewhat limited in current answer-set programming tools: Not all ASP systems support computing atomic consequences directly, and even if they do, it is usually done as a final processing step, in the sense that it is not possible to use the atomic consequences in the same run in order to do further reasoning. One could try to simulate this kind of reasoning by adding additional rules to the program over which the consequences are computed. However, the following simple example demonstrates the problems of this approach: The program $\{a \text{ :- not } b \ ; \ b \text{ :- not } a\}$ has two answer sets, $\{a\}$ and $\{b\}$, and so its brave atomic consequences are a and b, while there is no cautious atomic consequence. In order to represent the question whether at least one of a or b is a consequence, one could try to add $\{q \text{ :- } a \ ; \ q \text{ :- } b\}$ to the program and check whether q is an atomic consequence. While this works correctly for brave consequences (a positive answer), it does not for cautious consequences. The reason is that q is indeed a cautious consequence of the modified program thus yielding a positive answer to the query, while neither a nor b is a cautious atomic consequence.

Actually, one would hope to be able to use as many language features that ASP provides in order to reason with atomic consequences of a program, but as seen in the simple example above, existing query interfaces are insufficient for this task. Indeed, if one wants to employ recursion, a hypothetical method of endowing the original program by additional rules is quite obviously inadequate in most cases.

In this work, we introduce a framework that overcomes the limitations outlined above. In particular, we propose a language that encapsulates computing brave, cautious or definite[3] consequences of a program, which can then be utilized in a larger ASP program.

We discuss properties and limitations of the language and describe techniques for implementing a system supporting the language. In particular, we propose an extension of *manifold programs*, that we have recently proposed as a method for compiling query answering into ASP in [8]. In particular, a manifold program M_P of an ASP program

[3] An alternative notion proposed in [7].

P allows for identifying all consequences of a certain type (variants exist for brave, cautious, and definite consequences) within a single answer set. The framework we present here goes beyond the concept of a single manifold program which facilitates query answering wrt. a single program. Our framework permits that the results (i.e., consequences of a certain type) of modules can serve as input for further modules which compile different queries of their own, and so forth. A so-called base program finally collects the result necessary for the overall task and computes its own answer sets. These sets are identified as the answer sets of the entire framework.

However, there is a price to be paid for identifying program consequences by manifolding: While M_P contains optimization constructs (in [8] weak constraints were used, cf. [9]), P should not contain any optimization constructs. There is also a reason for this: While deciding whether one ground atom is a brave (respectively cautious) consequence is NP-complete for normal ground programs and Σ_2^P-complete for disjunctive ground programs (respectively co-NP- and Π_2^P-complete), enumerating brave (or cautious) consequences is complete for the complexity class $\mathrm{FP}_{||}^{\mathrm{NP}}$ for normal ground programs and for $\mathrm{FP}_{||}^{\Sigma_2^P}$ for disjunctive ground programs. It follows that unless the polynomial hierarchy collapses, a program enumerating brave or cautious consequences without optimization constructs does not exist. Moreover, in [9] the relevant decision problems for programs with weak constraints (without different levels) have been shown to be complete for the complexity class Θ_2^P (Θ_3^P in the presence of disjunction), from which the function complexity $\mathrm{FP}_{||}^{\mathrm{NP}}$ ($\mathrm{FP}_{||}^{\Sigma_2^P}$) can be obtained. It follows that the presence of weak constraints is necessary given our current knowledge on $NP \stackrel{?}{=} P$ and also not excessive.

2 Preliminaries

In this section, we review the basic syntax and semantics of ASP with weak constraints, following [10], to which we refer for a more detailed definition.

An *atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity $\alpha(p) = n \geq 0$ and each t_i is either a variable or a constant. A *literal* is either an atom a or its negation $\mathrm{not}\ a$. A *(disjunctive) rule* r is of the form

$$a_1 \vee \cdots \vee a_n \text{ :- } b_1, \ldots, b_k, \mathrm{not}\ b_{k+1}, \ldots, \mathrm{not}\ b_m$$

with $n \geq 0$, $m \geq k \geq 0$, $n + m > 0$, and where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms.

The *head* of r is the set $H(r) = \{a_1, \ldots, a_n\}$, and the *body* of r is the set $B(r) = \{b_1, \ldots, b_k, \mathrm{not}\ b_{k+1}, \ldots, \mathrm{not}\ b_m\}$. Furthermore, $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$. We will sometimes denote a rule r as $H(r)$:- $B(r)$.

A *weak constraint* [9] is an expression wc of the form

$$:\sim\ b_1, \ldots, b_k, \mathrm{not}\ b_{k+1}, \ldots, \mathrm{not}\ b_m.\ [w : l]$$

where $m \geq k \geq 0$ and $b_1, \ldots, b_m$ are literals, while $weight(wc) = w$ (the *weight*) and l (the *level*) are positive integer constants or variables. For convenience, w and/or l may be omitted and are set to 1 in this case. The sets $B(wc)$, $B^+(wc)$, and $B^-(wc)$ are defined as for rules. We will sometimes denote a weak constraint wc as $:\sim B(wc)$.

A *program* P is a finite set of rules and weak constraints. $Rules(P)$ denotes the set of rules and $WC(P)$ the set of weak constraints in P. w^P_{max} and l^P_{max} denote the maximum weight and maximum level over $WC(P)$, respectively. A program (rule, atom) is *propositional* or *ground* if it does not contain variables. A program is called *strong* if $WC(P) = \emptyset$, and *weak* otherwise.

For any program P, let U_P be the set of all constants appearing in P (if no constant appears in P, an arbitrary constant is added to U_P); let HB_P be the set of all ground literals constructible from the predicate symbols appearing in P and the constants of U_P; and let $Ground(P)$ be the set of rules and weak constraints obtained by applying, to each rule and weak constraint in P all possible substitutions from the variables in P to elements of U_P. U_P is usually called the *Herbrand Universe* of P and HB_P the *Herbrand Base* of P.

A ground rule r is *satisfied* by a set I of ground atoms iff $H(r) \cap I \neq \emptyset$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. I satisfies a ground program P, if each $r \in P$ is satisfied by I. For non-ground P, I satisfies P iff I satisfies $Rules(Ground(P))$. A ground weak constraint wc is *violated* by I, iff $B^+(wc) \subseteq I$ and $B^-(wc) \cap I = \emptyset$; it is satisfied otherwise.

Following [11], a set $I \subseteq HB_P$ of atoms is an *answer set* for a strong program P iff it is a subset-minimal set that satisfies the *reduct*

$$P^I = \{H(r) \text{ :- } B^+(r) \mid I \cap B^-(r) = \emptyset, r \in Ground(P)\}.$$

A set $I \subseteq HB_P$ of atoms is an *answer set* for a weak program P iff I is an answer set of $Rules(P)$ and $H^{Ground(P)}(I)$ is minimal among all the answer sets of $Rules(P)$, where the penalization function $H^P(I)$ for weak constraint violation of a ground program P is defined as follows:

$$\begin{array}{l} H^P(I) = \sum_{i=1}^{l^P_{max}} \left(f_P(i) \cdot \sum_{w \in N_i^P(I)} weight(w) \right) \\ f_P(1) \ = 1, \ \text{and} \\ f_P(n) \ = f_P(n-1) \cdot |WC(P)| \cdot w^P_{max} + 1 \ \text{ for } n > 1. \end{array}$$

where $N_i^P(I)$ denotes the set of weak constraints of P in level i violated by I. For any program P, we denote the set of its answer sets by $AS(P)$. Note that for programs having weak constraints only of weight and level 1, $H^{Ground(P)}(I)$ amounts to the number of weak constraints violated in I.

A ground atom a is a *brave* (sometimes also called credulous or possible) consequence of a program P, denoted $P \models_b a$, if $a \in A$ holds for at least one $A \in AS(P)$. A ground atom a is a *cautious* (sometimes also called skeptical or certain) consequence of a program P, denoted $P \models_c a$, if $a \in A$ holds for all $A \in AS(P)$. A ground atom a is a *definite* consequence [7] of a program P, denoted $P \models_d a$, if $AS(P) \neq \emptyset$ and $a \in A$ holds for all $A \in AS(P)$. The sets of all brave, cautious, definite consequences of a program P are denoted as $BC(P)$, $CC(P)$, $DC(P)$, respectively.

3 Consequence Modules

A module essentially consists of a program, a collection of partially instantiated atoms, and a reasoning mode. It can also receive some predicates as input. The idea is that this

module represents those consequences of the program under the specified reasoning mode which match one of the atoms.

Definition 1. *A* consequence module *(or module, for short) is a quadruple $\langle P, I, O, m\rangle$, where P (the module program) is a strong program, I (the input predicates) is a set of predicates, O (the output atoms) is a set of atoms (possibly containing variables), and m (the reasoning mode) is one of* brave, cautious, definite*.*

A consequence module framework *(or consequence module program) $F = \langle B, \mathcal{M}\rangle$ consists of a strong program B (called the base program) and a set $\mathcal{M}$ of consequence modules.*

Although the realization of a module looks very similar to known concepts (e.g. modules as defined in [12] or the signature of module atoms in [13]), we remark that the concept of a reasoning mode clearly separates our approach from previous ones. In particular, the output of a module in our approach is just a set of facts (depending on the chosen reasoning mode, this set is obtained from the answer sets of the modules) which serves as input to further modules, while in other approaches the output is usually a collection of answer sets, which have to combined with answer sets of other modules.

We define the universe U_F of a consequence module framework F as the set of all constants appearing in F (if no constant appears in F, an arbitrary constant is added), and the base HB_F of F as the set of all ground literals constructible from the predicate symbols appearing in F and the constants of U_F.

A consequence module framework is stratified on modules if there are no circular dependencies through consequence modules. In the following, let $Pred(\Sigma)$ denote the set of predicates in a syntactic element Σ.

Definition 2 (Stratification on Modules). *A consequence module framework $F = \langle B, \mathcal{M}\rangle$ is* stratified on modules *if there exists a level mapping λ (a stratification) from the set of predicates in F to $\mathbb{N}$, such that for each rule r in B, $\lambda(b) \leq \lambda(h)$ holds for each $b \in Pred(B(r))$ and $h \in Pred(H(r))$, and for each module $\langle P, I, O, m\rangle \in \mathcal{M}$, $\lambda(i) < \lambda(o)$ holds for each $i \in I$ and $o \in Pred(O)$.*

In the following, we will consider only consequence module programs, which are stratified on modules.

The semantics of a stratified consequence module program F is given by an evaluation along one of its level mappings. In other words, the answer sets of F are obtained by simply running the modules in an order of stratification and applying the modules query on the result of each.

Definition 3. *Given a stratified consequence module framework $F = \langle B, \mathcal{M}\rangle$ and λ one of its stratifications, let, for each $i \in \mathbb{N}$, $B_i = \{r \in B \mid i = max\{\lambda(h) \mid h \in H(r)\}\}$ and $\mathcal{M}_i = \{\langle P, I, O, m\rangle \in \mathcal{M} \mid i = max\{\lambda(o) \mid o \in Pred(O)\}\}$.*

Definition 4. *For a module $M = \langle P, I, O, m\rangle$ and a set A of ground atoms, $AS(A \triangleright M) = \{o\sigma \mid o \in O, o\sigma \in X\}$, where σ is a substitution, $X = BC(P \cup A)$ if $m =$ brave, $X = CC(P \cup A)$ if $m =$ cautious, $X = DC(P \cup A)$ if $m =$ definite. For a set $\mathcal{M}$ of modules, let $AS(A \triangleright \mathcal{M}) = \bigcup_{M \in \mathcal{M}} AS(A \triangleright M)$.*

Given a stratified consequence module framework $F = \langle B, \mathcal{M} \rangle$, we then define the following sequence

$$AS_0(F) = AS(B_0) \cup AS(\emptyset \triangleright \mathcal{M}_0)$$
$$AS_i(F) = AS(AS_{i-1}(F) \cup B_i) \cup AS(AS_{i-1}(F) \triangleright \mathcal{M}_i), \text{ for } i > 0$$

in order to obtain $AS(F) = AS_n(F)$ where $n = max\{\lambda(p) \mid p \in Pred(F)\}$.

It is not hard to see that any stratification will lead to the same answer sets.

Note that the semantics of unstratified consequence module frameworks cannot be defined in this way because of circular dependencies. In this paper we refrain from studying unstratified settings, as their intended semantics is not obvious and possibly gives rise to complexity issues. In a similar way, we do not consider nested modules (that is, occurrences of modules inside module programs). While the intended semantics for these would be more obvious, they would hamper the considerations in Section 5 and possibly also give rise to complexity issues. We believe that the framework in this paper is sufficiently rich to describe many problems, which incur reasoning subtasks, in a natural way. We conjecture that considering an unrestricted language not requiring stratification and allowing for nested consequence modules would result in a relatively high complexity, while the language considered here essentially stays inside ASP complexity bounds.

4 Applications

In this section, we study some example encodings using consequence modules. The first one is a well-known problem from propositional logic, which we will describe in detail, the second one is from planning. Another example would be computing the ideal extension for abstract argumentation frameworks, which was studied in [8], and which we omit here for space reasons.

4.1 The Unique Minimal Model Problem

As a first example, we show how to encode the problem of deciding whether a given propositional formula φ has a unique minimal model. This problem is known to be in Θ_2^P and to be co-NP-hard (the exact complexity is an open problem). Our encodings will rely on the following observation which is obvious if one considers models as sets of those atoms which are assigned to true: Let I be the intersection of all models of φ, then φ has a unique minimal model iff I is also a model of φ.

We will use a simple consequence module framework for this task consisting of a single module which will take care of computing the intersection of all models of a propositional CNF formula φ, and a simple base program which, on the one hand, contains a suitable representation of the formula φ (and passes this to the module), and, on the other hand, checks whether the result of the module yields a model of φ.

Let us make this idea more precise. To start with, we fix the representation of CNFs. Let φ (over atoms A) be of the form $\bigwedge_{i=1}^{n} c_i$. Then,

$$D_\varphi = \{\mathrm{at}(a) \mid a \in A\} \cup \{\mathrm{cl}(i) \mid 1 \leq i \leq n\} \cup$$

$$\{\mathrm{pos}(a,i) \mid \text{atom } a \text{ occurs positively in } c_i\} \cup$$
$$\{\mathrm{neg}(a,i) \mid \text{atom } a \text{ occurs negatively in } c_i\}.$$

For the module, we require a program whose answer sets are in a one-to-one correspondence to models of formulas. For this purpose, consider the program SAT as the set of the following rules.

$$\begin{aligned} \mathit{true}(X) &\text{ :- } \text{not } \mathit{false}(X), \mathrm{at}(X);\\ \mathit{false}(X) &\text{ :- } \text{not } \mathit{true}(X), \mathrm{at}(X);\\ \mathit{ok}(C) &\text{ :- } \mathit{true}(X), \mathrm{pos}(X,C);\\ \mathit{ok}(C) &\text{ :- } \mathit{false}(X), \mathrm{neg}(X,C);\\ &\text{ :- } \text{not } \mathit{ok}(C), \mathrm{cl}(C). \end{aligned}$$

It can be checked that the answer sets of $\mathrm{SAT} \cup D_\varphi$ are in a one-to-one correspondence to the models (over A) of φ. In particular, for any model $I \subseteq A$ of φ there exists an answer set M of $\mathrm{SAT} \cup D_\varphi$ such that $I = \{a \mid \mathit{true}(a) \in M\}$.

Our consequence module will now be given by

$$\mathrm{SAT}_{\mathrm{cautious}} = \langle \mathrm{SAT}, \{\mathrm{at}, \mathrm{cl}, \mathrm{pos}, \mathrm{neg}\}, \{\mathit{true}(X)\}, \mathsf{cautious} \rangle.$$

In fact, using D_φ as input to $\mathrm{SAT}_{\mathrm{cautious}}$, we obtain a result which characterizes those atoms in φ which are true in all models of φ.

For the base program, let us now define the program $\mathrm{MODELCHECK}$ as the set of the following rules

$$\begin{aligned} &\mathit{ok}(C) \text{ :- } \mathit{true}(X), \mathrm{pos}(X,C);\\ &\mathit{ok}(C) \text{ :- } \text{not } \mathit{true}(X), \mathrm{neg}(X,C);\\ &\text{ :- } \text{not } \mathit{ok}(C), \mathrm{cl}(C). \end{aligned}$$

We immediately obtain the following result.

Theorem 1. *For any CNF formula φ, it holds that φ has a unique minimal model, if and only if the framework $\langle D_\varphi \cup \mathrm{MODELCHECK}, \{\mathrm{SAT}_{\mathrm{cautious}}\}\rangle$ has an answer set.*

A slight adaption of this encoding allows us to formalize reasoning under the closed-world assumption (CWA), cf. [14], over a propositional knowledge base φ, since the atoms a in φ, for which the corresponding atoms $\mathit{true}(a)$ are *not* contained in any answer set of the program SAT on input D_φ, i.e. those atoms $\mathit{true}(a)$ *not* contained in the output of the module $\mathrm{SAT}_{\mathrm{cautious}}$ on input D_φ, are exactly those which are added negated to φ for CWA-reasoning. In other words, the framework,

$$\langle D_\varphi \cup \{\mathit{false}(X) \text{ :- } \text{not } \mathit{true}(X), \mathrm{at}(X)\}, \{\mathrm{SAT}_{\mathrm{cautious}}\}\rangle$$

represents the closed-world closure of φ. Further extensions of the base program can now be used to formulate CWA-reasoning problems.

4.2 Planning

Let us consider secure planning, which is also known as conformant or certain planning [15–18]. Given a description of a nondeterministic transition system involving states (composed of fluents) and actions (occurring between states), a secure plan is a sequence of actions (a plan), which allows for reaching a goal state in any possible outcome of action execution. This means that, starting from a set of initial states, executing a secure plan must not get "stuck" during execution (the subsequent action must always be executable), and must eventually reach the goal state.

Let us consider the problem of deciding whether a given plan is secure. For the language $\mathcal{K}$ of [17], some ASP encodings have been defined in [19]. Let us assume the availability of a program TRAJ (for a given transition system described in $\mathcal{K}$) that has one answer set for each trajectory for a sequence of actions given in the input, where a trajectory is a sequence of states along a path in the given transition system that is labeled by the sequence of actions in the input. For so-called proper transition systems (cf. [17]) it is sufficient to check whether all trajectories end in a goal state.

We can then define a consequence module

$$\mathrm{TRAJ}_{\mathrm{cautious}} = \langle \mathrm{TRAJ}, \{a_1, \ldots, a_n\}, \{f_1(\overline{t_1}), \ldots, f_n(\overline{t_n})\}, \mathsf{cautious} \rangle.$$

where $a_1, \ldots, a_n$ are predicates representing actions (and thus plans), and $f_1(\overline{t_1}), \ldots, f_n(\overline{t_n})$ are atoms representing a goal state. Secure plan checking can then be captured by a framework

$$\langle A_P \cup G = \{ \text{:- not } g \ ; \ g \text{:- } f_1(\overline{t_1}), \ldots, f_n(\overline{t_n}) \}, \{\mathrm{TRAJ}_{\mathrm{cautious}}\} \rangle$$

where A_P is an encoding of the plan to be checked. If there is an answer set, the plan is secure.

Now assume that INITEX is a program that computes initial states of a given transition system and which moreover derives an atom i with each initial state (cf. [19]). We define the consequence module

$$\mathrm{INITEX}_{\mathrm{brave}} = \langle \mathrm{INITEX}, \emptyset, \{i\}, \mathsf{brave} \rangle.$$

Moreover, assume the existence of a program $\mathrm{ENUMPLANS}$, which whenever i holds, generates as answer sets all possible plans of a specified length (cf. [19]). Finding secure plans for proper planning domains can then be accomplished by the following framework.

$$\langle \mathrm{ENUMPLANS} \cup G, \{\mathrm{INITEX}_{\mathrm{brave}}, \mathrm{TRAJ}_{\mathrm{cautious}}\} \rangle$$

5 Transforming Consequence Modules to ASP

While one could implement the suggested language using oracles formed of ASP systems (see [20] for a recent realization of such an approach), and so lifting the framework on a metalevel, we propose an alternative method which allows for an implementation using ASP itself. We make use of manifold programs that we have recently proposed in [8], and elaborate on them.

5.1 Manifold Programs

The main idea of manifold programs is to obtain a translation which creates a copy of a given program for each element of a subset of its Herbrand base. Let us first consider the simpler case of propositional programs.

We create a copy of a given program P for each atom a in a given set S, whereby the transformation guarantees the existence of an answer set by enabling the copies conditionally.

Definition 5. *For a strong propositional program P and $S \subseteq HB_P$, define its* manifold *as*

$$P_S^{tr} = \bigcup_{r \in P} \{H(r)^a \text{ :- } \{c\} \cup B(r)^a \mid a \in S\} \cup \{c \text{ :- not } i \ ; \ i \text{ :- not } c\}.$$

where $I^a = \{p^a \mid \text{atom } p \in I\} \cup \{\text{not } p^a \mid \text{not } p \in I\}$ for a set I of atoms and an atom a. We assume $HB_P \cap HB_{P_S^{tr}} = \emptyset$, that is, all symbols in P_S^{tr} are assumed to be fresh.

Example 1. Consider $\Phi = \{p \vee q \text{ :- } \ ; \ r \text{ :- } p \ ; \ r \text{ :- } q\}$ for which $AS(\Phi) = \{\{p, r\}, \{q, r\}\}$, $BC(\Phi) = \{p, q, r\}$ and $CC(\Phi) = DC(\Phi) = \{r\}$. When forming the manifold for $HB_\Phi = \{p, q, r\}$, we obtain

$$\Phi_{HB_\Phi}^{tr} = \left\{ \begin{array}{l} p^p \vee q^p \text{ :- } c \ ; \ r^p \text{ :- } c, p^p \ ; \ r^p \text{ :- } c, q^p \ ; \ c \text{ :- not } i \ ; \\ p^q \vee q^q \text{ :- } c \ ; \ r^q \text{ :- } c, p^q \ ; \ r^q \text{ :- } c, q^q \ ; \ i \text{ :- not } c \ ; \\ p^r \vee q^r \text{ :- } c \ ; \ r^r \text{ :- } c, p^r \ ; \ r^r \text{ :- } c, q^r \end{array} \right\}$$

Note that given a strong program P and $S \subseteq HB_P$, the construction of P_S^{tr} can be done in polynomial time (w.r.t. the size of P). The answer sets of the transformed program consist of (and extend) all combinations (of size $|S|$) of answer sets of the original program (augmented by c) plus the special answer set $\{i\}$ which we shall use to indicate inconsistency of P.

Example 2. For Φ of Example 1, we obtain that $AS(\Phi_{HB_\Phi}^{tr})$ consists of $\{i\}$ plus (copies of $\{q, r\}$ are underlined for readability)

$$\{c, p^p, r^p, p^q, r^q, p^r, r^r\}, \{c, \underline{q^p, r^p}, p^q, r^q, p^r, r^r\}, \{c, p^p, r^p, \underline{q^q, r^q}, p^r, r^r\},$$
$$\{c, p^p, r^p, p^q, r^q, \underline{q^r, r^r}\}, \{c, \underline{q^p, r^p}, \underline{q^q, r^q}, p^r, r^r\}, \{c, \underline{q^p, r^p}, p^q, r^q, \underline{q^r, r^r}\},$$
$$\{c, p^p, r^p, \underline{q^q, r^q}, \underline{q^r, r^r}\}, \{c, \underline{q^p, r^p}, \underline{q^q, r^q}, \underline{q^r, r^r}\}.$$

Using this transformation, each answer set encodes an association of an atom with some answer set of the original program. If an atom a is a brave consequence of the original program, then a witnessing answer set exists, which contains the atom a^a. The idea is now to prefer those atom-answer set associations where the answer set is a witness. We do this by means of weak constraints and penalize each association where the atom is not in the associated answer set, that is, where a^a is not in the answer set of the transformed program. Doing this for each atom means that an optimal answer set will not contain a^a only if there is no answer set of the original program that contains a, so each a^a contained in an optimal answer set is a brave consequence of the original program.

Definition 6. *Given a strong propositional program P and $S \subseteq HB_P$, let*

$$P_S^{bc} = P_S^{tr} \cup \{:\sim \text{not } a^a \mid a \in S\} \cup \{:\sim i\}$$

Observe that all weak constraints are violated in the special answer set $\{i\}$, while in the answer set $\{c\}$ (which occurs if the original program has an empty answer set) all but $:\sim i$ are violated.

Example 3. For the program Φ as given Example 1, $\Phi_{HB_\Phi}^{bc}$ is given by $\Phi_{HB_\Phi}^{tr} \cup \{:\sim \text{not } p^p \ ; \ :\sim \text{not } q^q \ ; \ :\sim \text{not } r^r \ ; \ :\sim i\}$. We obtain that $AS(\Phi_{HB_\Phi}^{bc}) = \{A_1, A_2\}$, where $A_1 = \{c, p^p, r^p, q^q, r^q, p^r, r^r\}$ and $A_2 = \{c, p^p, r^p, q^q, r^q, q^r, r^r\}$, as these two answer sets are the only ones that violate no weak constraint. We can observe that $\{a \mid a^a \in A_1\} = \{a \mid a^a \in A_2\} = \{p, q, r\} = BC(\Phi)$.

For cautious consequences, we use a similar idea, taking into account that if a program is inconsistent (in the sense that it does not have any answer set), each atom is a cautious consequence.

Definition 7. *Given a strong propositional program P and $S \subseteq HB_P$, let*

$$P_S^{cc} = P_S^{tr} \cup \{:\sim a^a \mid a \in S\} \cup \{a^a \text{ :- } i \mid a \in S\} \cup \{:\sim i\}$$

Example 4. Recall program Φ from Example 1. We have $\Phi_{HB_\Phi}^{cc} = \Phi_{HB_\Phi}^{tr} \cup \{:\sim p^p \ ; \ :\sim q^q \ ; \ :\sim r^r \ ; \ p^p \text{ :- } i \ ; \ q^q \text{ :- } i \ ; \ r^r \text{ :- } i \ ; \ :\sim i\}$. We obtain that $AS(\Phi_{HB_\Phi}^{cc}) = \{A_3, A_4\}$, where $A_3 = \{c, q^p, r^p, p^q, r^q, p^r, r^r\}$ and $A_4 = \{c, q^p, r^p, p^q, r^q, q^r, r^r\}$, as these two answer sets are the only ones that violate only one weak constraint, namely $:\sim r^r$. We observe that $\{a \mid a^a \in A_3\} = \{a \mid a^a \in A_4\} = \{r\} = CC(\Phi)$.

Finally we slightly adapt the construction for definite consequences.

Definition 8. *Given a strong propositional program P and $S \subseteq HB_P$, let*

$$P_S^{dc} = P_S^{tr} \cup \{:\sim a^a;\ i^a \text{ :- } i;\ :\sim i^a \mid a \in S\} \cup \{:\sim i\}$$

Example 5. Recall program Φ from Example 1. We have $\Phi_{HB_\Phi}^{dc} = \Phi_{HB_\Phi}^{tr} \cup \{:\sim p^p \ ; \ :\sim q^q \ ; \ :\sim r^r \ ; \ i^p \text{ :- } i \ ; \ i^q \text{ :- } i \ ; \ i^r \text{ :- } i :\sim i^p \ ; \ :\sim i^q \ ; \ :\sim i^r \ ; \ :\sim i\}$. As in Example 4, A_3 and A_4 are the only ones that violate only one weak constraint, namely $:\sim r^r$, and thus are the answer sets of $\Phi_{HB_\Phi}^{dc}$.

Proposition 1. *Given a strong propositional program P and $S \subseteq HB_P$, for any $B \in AS(P_S^{bc})$, $\{b \mid b^b \in B\} = BC(P) \cap S$; for any $C \in AS(P_S^{cc})$, $\{c \mid c^c \in C\} = CC(P) \cap S$; for any $D \in AS(P_S^{dc})$, $\{d \mid d^d \in D\} = DC(P) \cap S$.*

Obviously, one can compute all brave, cautious, or definite consequences of a program by choosing $S = HB_P$. We also note that the programs from Definitions 6, 7 and 8 yield multiple answer sets. However each of these yields the same atoms a^a, so it is sufficient to compute one of these. This issue will be addressed in Section 5.2.

We now generalize these techniques to non-ground strong programs. In principle, one could annotate each predicate (rather than atom as before) with ground atoms of

a subset of the Herbrand Base. However, one can also move the annotations to the non-ground level: For example, instead of annotating a rule $p(X,Y) \text{ :- } q(X,Y)$ by the set $\{r(a), r(b)\}$ yielding $p^{r(a)}(X,Y) \text{ :- } q^{r(a)}(X,Y)$ and $p^{r(b)}(X,Y) \text{ :- } q^{r(b)}(X,Y)$ we will annotate using only the predicate r and extend the arguments of p, yielding the compact rule $\mathrm{d}_p^r(X,Y,Z) \text{ :- } \mathrm{d}_q^r(X,Y,Z)$ (we use predicate symbols d_p^r and d_q^r rather than p^r and q^r just for pointing out the difference between annotation by predicates versus annotation by ground atoms). In this particular example we have assumed that the program is to be annotated by all ground instances of $r(Z)$; we will use this assumption also in the following for simplifying the presentation. In practice, one can clearly add atoms to the rule body for restricting the instances of the predicate by which we annotate, in the example this would yield $p^r(X,Y,Z) \text{ :- } q^r(X,Y,Z), dom(Z)$ where the predicate dom should be defined appropriately. In the following, recall that $\alpha(p)$ denotes the arity of a predicate p.

Definition 9. *Given an atom $a = p(t_1, \ldots, t_n)$ and a predicate q, let a_q^{tr} be the atom $\mathrm{d}_p^q(t_1, \ldots, t_n, X_1, \ldots, X_{\alpha(q)})$ where $X_1, \ldots, X_{\alpha(q)}$ are fresh variables and d_p^q is a new predicate symbol with $\alpha(\mathrm{d}_p^q) = \alpha(p) + \alpha(q)$. Furthermore, given a set $\mathcal{L}$ of literals, and a predicate q, let $\mathcal{L}_q^{tr}$ be $\{a_q^{tr} \mid \text{atom } a \in \mathcal{L}\} \cup \{\text{not } a_q^{tr} \mid \text{not } a \in \mathcal{L}\}$.*

Note that we assume that even though the variables $X_1, \ldots, X_{\alpha(q)}$ are fresh, they will be the same for each a_q^{tr}. One could define similar notions also for partially ground atoms or for sets of atoms characterized by a collection of defining rules, from which we refrain here for the ease of presentation. We define the manifold program in analogy to Definition 5, the only difference being the different way of annotating.

Definition 10. *Given a strong program P and a set S of predicates, define its* manifold *as*

$$P_S^{tr} = \bigcup_{r \in P} \{H(r)_q^{tr} \text{ :- } \{c\} \cup B(r)_q^{tr} \mid q \in S\} \cup \{c \text{ :- not } i \ ; \ i \text{ :- not } c\}.$$

Example 6. Consider program $\Psi = \{p(X) \vee q(X) \text{ :- } r(X); \ ; \ r(a) \text{ :- } \ ; \ r(b) \text{ :- }\}$ for which $AS(\Psi) = \{\{p(a), p(b), r(a), r(b)\}, \{p(a), q(b), r(a), r(b)\}, \{q(a), p(b), r(a), r(b)\}, \{q(a), q(b), r(a), r(b)\}\}$. Hence, $BC(\Psi) = \{p(a), p(b), q(a), q(b), r(a), r(b)\}$ and $CC(\Psi) = DC(\Psi) = \{r(a), r(b)\}$. Forming the manifold for $S = \{p\}$, we obtain

$$\Psi_S^{tr} = \left\{ \begin{array}{l} \mathrm{d}_p^p(X, X_1) \vee \mathrm{d}_q^p(X, X_1) \text{ :- } \mathrm{d}_r^p(X, X_1), c \ ; \\ \mathrm{d}_r^p(a, X_1) \text{ :- } c \ ; \ \mathrm{d}_r^p(b, X_1) \text{ :- } c \ ; \ c \text{ :- not } i \ ; \ i \text{ :- not } c \end{array} \right\}$$

$AS(\Psi_S^{tr})$ consists of $\{i\}$ plus 16 answer sets, corresponding to all combinations of the four answer sets in $AS(\Psi)$.

Now we are able to generalize the encodings for brave, cautious, and definite consequences. These definitions are direct extensions of Definitions 6, 7, and 8, the differences are only due to the non-ground annotations. In particular, the diagonalization atoms a^a should now be written as $\mathrm{d}_p^p(X_1, \ldots, X_{\alpha(p)}, X_1, \ldots, X_{\alpha(p)})$ which represent the set of ground instances of $p(X_1, \ldots, X_{\alpha(p)})$, each annotated by itself. So, a weak constraint $:\sim \ \mathrm{d}_p^p(X_1, \ldots, X_{\alpha(p)}, X_1, \ldots, X_{\alpha(p)})$ gives rise to $\{:\sim \ \mathrm{d}_p^p(c_1, \ldots, c_{\alpha(p)}, c_1, \ldots, c_{\alpha(p)}) \mid c_1, \ldots, c_{\alpha(p)} \in U\}$ where U is the Herbrand base of the program in question, that is one weak constraint for each ground instance annotated by itself.

Definition 11. *Given a strong program P and a set S of predicate symbols, let*

$$P_S^{bc} = P_S^{tr} \cup \{:\sim \text{not } \Delta_q \mid q \in S\} \cup \{:\sim i\}$$
$$P_S^{cc} = P_S^{tr} \cup \{:\sim \Delta_q;\ \Delta_q \text{ :- } i \mid q \in S\} \cup \{:\sim i\}$$
$$P_S^{dc} = P_S^{tr} \cup \{:\sim \Delta_q;\ I_q \text{ :- } i;\ :\sim I_q \mid q \in S\} \cup \{:\sim i\}$$

where $\Delta_q = \mathrm{d}_q^q(X_1, \ldots, X_{\alpha(q)}, X_1, \ldots, X_{\alpha(q)})$ and $I_q = i_q(X_1, \ldots, X_{\alpha(q)})$.

Proposition 2. *Given a strong program P and a set S of predicates, for an arbitrary $A \in AS(P_S^{bc})$, (resp., $A \in AS(P_S^{cc})$, $A \in AS(P_S^{dc})$), the set $\{p(c_1, \ldots, c_{\alpha(p)}) \mid \mathrm{d}_p^p(c_1, \ldots, c_{\alpha(p)}, c_1, \ldots, c_{\alpha(p)}) \in A\}$ is the set of brave (resp., cautious, definite) consequences of P with a predicate in S.*

Example 7. Consider again Ψ and $S = \{p\}$ from Example 6. We obtain $\Psi_S^{bc} = \Psi_S^{tr} \cup \{:\sim \text{not } \mathrm{d}_p^p(X_1, X_1)\ ;\ :\sim i\}$ and we can check that $AS(\Psi_S^{bc})$ consists of the sets

$$R \cup \{\mathrm{d}_p^p(a,a), \mathrm{d}_p^p(b,b), \mathrm{d}_q^p(a,b), \mathrm{d}_q^p(b,a)\}, R \cup \{\mathrm{d}_p^p(a,a), \mathrm{d}_p^p(b,b), \mathrm{d}_p^p(a,b), \mathrm{d}_q^p(b,a)\},$$
$$R \cup \{\mathrm{d}_p^p(a,a), \mathrm{d}_p^p(b,b), \mathrm{d}_q^p(a,b), \mathrm{d}_p^p(b,a)\}, R \cup \{\mathrm{d}_p^p(a,a), \mathrm{d}_p^p(b,b), \mathrm{d}_p^p(b,a), \mathrm{d}_p^p(b,a)\};$$

where $R = \{\mathrm{d}_r^p(a,a), \mathrm{d}_r^p(a,b), \mathrm{d}_r^p(b,a), \mathrm{d}_r^p(b,b)\}$. For each A of these answer sets we obtain $\{p(t) \mid \mathrm{d}_p^p(t,t) \in A\} = \{p(a), p(b)\}$ which corresponds exactly to the brave consequences of Ψ with a predicate of $S = \{p\}$.
For cautious consequences, $\Psi_S^{cc} = \Psi_S^{tr} \cup \{:\sim \mathrm{d}_p^p(X_1, X_1)\ ;\ \mathrm{d}_p^p(X_1, X_1) \text{ :- } i\ ;\ :\sim i\}$ and we can check that $AS(\Psi_S^{cc})$ consists of the sets

$$R \cup \{\mathrm{d}_q^p(a,a), \mathrm{d}_q^p(b,b), \mathrm{d}_q^p(a,b), \mathrm{d}_q^p(b,a)\}, R \cup \{\mathrm{d}_q^p(a,a), \mathrm{d}_q^p(b,b), \mathrm{d}_p^p(a,b), \mathrm{d}_q^p(b,a)\},$$
$$R \cup \{\mathrm{d}_q^p(a,a), \mathrm{d}_q^p(b,b), \mathrm{d}_q^p(a,b), \mathrm{d}_p^p(b,a)\}, R \cup \{\mathrm{d}_q^p(a,a), \mathrm{d}_q^p(b,b), \mathrm{d}_p^p(b,a), \mathrm{d}_p^p(b,a)\};$$

where $R = \{\mathrm{d}_r^p(a,a), \mathrm{d}_r^p(a,b), \mathrm{d}_r^p(b,a), \mathrm{d}_r^p(b,b)\}$. For each A of these answer sets we obtain $\{p(t) \mid \mathrm{d}_p^p(t,t) \in A\} = \emptyset$ and indeed there are no cautious consequences of Ψ with a predicate of $S = \{p\}$.
Finally, for definite consequences, $\Psi_S^{dc} = \Psi_S^{tr} \cup \{:\sim \mathrm{d}_p^p(X_1, X_1)\ ;\ i_p(X_1) \text{ :- } i\ ;\ :\sim i_p(X_1)\ ;\ :\sim i\}$. It is easy to see that $AS(\Psi_S^{dc}) = AS(\Psi_S^{cc})$ and so $\{p(t) \mid \mathrm{d}_p^p(t,t) \in A\} = \emptyset$ for each answer set A of Ψ_S^{dc}, and indeed there is also no definite consequence of Ψ with a predicate of $S = \{p\}$.

These definitions exploit the fact that the semantics of non-ground programs is defined via their grounding with respect to their Herbrand Universe. So the fresh variables introduced in the manifold will give rise to one copy of a rule for each ground atom. In practice, ASP systems usually require rules to be safe, that is, that each variable occurs (also) in the positive body. The manifold for a set of predicates may therefore contain unsafe rules (because of the fresh variables). But this can be repaired by adding a *domain atom* $dom_q(X_1, \ldots, X_m)$ to a rule which is to be annotated with q. This predicate can in turn be defined by a rule $dom_q(X_1, \ldots, X_m)$:- $u(X_1), \ldots, u(X_m)$ where u is defined using $\{u(c) \mid c \in U_P\}$. One can also provide smarter definitions for dom_q by using a relaxation of the definition for q.

5.2 Transforming Consequence Module Frameworks by Manifolding

The main intuition is to replace each module by a suitable manifold program. In particular, given a module $M = \langle P, I, O, m\rangle$ in a framework F, we intend to create its manifold transform as $P^{bc}_{Pred(O)}$ if $m = \mathsf{brave}$, $P^{cc}_{Pred(O)}$ if $m = \mathsf{cautious}$, $P^{dc}_{Pred(O)}$ if $m = \mathsf{definite}$. Together with suitable *adaptor rules*, which map the transformed predicates back to predicates of the original program, these will be joined to the base program of the framework.

However, there are two main issues to resolve: As remarked earlier, the various manifold programs may admit more than one answer set, which are equivalent with respect to the consequences represented in them. Still, in the context of modules we would like to have a single answer set. The second issue deals with the fact that the manifold transforms of different modules should not interfere with each other.

The first issue can be dealt with by adding penalties in a way that only one answer set remains. In order to avoid interference with other weak constraints, these should be put into a separate level of lower importance. To this end one should fix an arbitrary order of the ground atoms in $X = \{\mathrm{d}^q_q(c_1, \ldots, c_{\alpha(q)}, c'_1, \ldots, c'_{\alpha(q)}) \mid c_i, c'_j \in U_F, c_k \neq c'_k\} \cup \{\mathrm{d}^p_q(c_1, \ldots, c_{\alpha(q)}, c'_1, \ldots, c'_{\alpha(p)}) \mid p \neq q, c_i, c'_j \in U_F\}$ and assigning weights of the exponential sequence $1, 2, 4, 8, \ldots$ to them. This is because each atom should incur a penalty which is greater than the sum of penalties of all preceding atoms. In particular, if $a_0, a_1, \ldots$ is an enumeration of X respecting the chosen order, add a weak constraint $:\sim a_i.[2^i : 1]$. The weak constraints of the original manifold programs should be put in the more important level 2 (higher levels are more important in the semantics of weak constraints), so all weak constraints introduced in Section 5.1 should be extended by $[1 : 2]$ (weight 1 is the default for weights, which was implicitly used in Section 5.1).

The weak constraints introduced in this way can be thought of reducing the set of answer-set candidates in the following way: If answer sets without a_0 exist, further consider only those, otherwise there is no reduction. So the resulting candidates either all do not contain a_0, or all do. Then, among the result, if answer sets without a_1 exist, further consider only those, otherwise there is no reduction. The remaining candidates do not differ on the presence of a_0 and a_1. Continuing like this, in the end the remaining candidates will not differ on the presence of any element in X. If the original set of answer-set candidates differs only on elements in X, then only one answer set remains.

The second modification regards combinability of manifold programs. We would like to be able to simply form the union of all manifold programs replacing the modules. The way in which manifold programs have been defined in Section 5.1, they could in principle share predicate names, which would lead to unwanted interferences. We therefore ensure that each manifold program introduces a unique set of predicates by extending the predicates d^q_p, i_q (p^a, i^a in the propositional case) and i, c by a string uniquely identifying the module, which the manifold program represents.

Definition 12. *For a module $M = \langle P, I, O, m\rangle$, let its* manifold transform *be defined as $T(M) = P^m_{Pred(O)}$, where $P^{\mathsf{brave}}_{Pred(O)}$, $P^{\mathsf{cautious}}_{Pred(O)}$, $P^{\mathsf{definite}}_{Pred(O)}$ correspond to the manifold programs of Section 5.1 with the modifications described above.*

The adaptor rules *for the module M are defined as*

$$A_M = \{p(t_1, \ldots, t_{\alpha(p)}) \text{ :- } \mathrm{d}^p_p(t_1, \ldots, t_{\alpha(p)}, t_1, \ldots, t_{\alpha(p)}) \mid p(t_1, \ldots, t_{\alpha(p)}) \in O\}$$

The manifold program *for a consequence module framework* $F = \langle B, \mathcal{M} \rangle$ *is then* $T(F) = B \cup \bigcup_{M \in \mathcal{M}} (T(M) \cup A_M)$.

Now we can state the correspondence result.

Proposition 3. *For a consequence module framework* F, $AS(F) = AS(T(F)) \cap HB_F$. *In fact, there is a one-to-one correspondence between* $AS(F)$ *and* $AS(T(F)) \cap HB_F$.

Some of the key observations for this correspondence result are that the dependencies of predicates of the base program remain unaltered in $T(F)$, and that module dependencies between predicates in F become standard dependencies in $T(F)$ via the predicates introduced by manifolding. This allows for applying the splitting set theorem of [21] to the program without weak constraints, mimicking the sequence $AS_i(\cdot)$ of Definition 4. However, the manifold parts of $T(F)$ give rise to many answer-set candidates, among which there are also answer sets that contain exactly the consequences under the respective reasoning mode. The combined program $T(F)$ thus will contain many answer-set candidates, but among these there are also precisely the answer sets of the framework, because the latter are defined by replacing the modules by the respective consequences.

The combination of all weak constraints then eliminates all but these candidates. Combining the weak constraints, considering first only those weak constraints described in Section 5.1, has the desired effect because the symbols introduced by $T(M)$ are not contained in any other $T(M')$. Because of this and since these weak constraints all have weight 1, any global optimum must also be an optimum locally for any $T(M)$. Therefore, without adding the additional weak constraints for enforcing uniqueness, the first part of Proposition 3 already holds. One can then show that the differences between multiple answer sets of $T(F)$ representing a single answer set of F is only due to atoms in the sets X described above, which are reduced to precisely one using the method described earlier, thus obtaining a one-to-one correspondence.

6 Conclusion

In this paper, we provided a novel framework for specifying ASP programs, which involve the consequences of subprograms, defining syntax and semantics of the proposed language. We gave examples for problems that possess a comparably natural representation in this language, while a traditional ASP specification is not obvious. Moreover, we proposed a transformation of consequence module frameworks to ASP with weak constraints, based on an adaption of the recently proposed manifold program technique, which allows for using a standard ASP solver supporting weak constraints for computing answer sets of consequence module frameworks.

For future work, we are interested in studying the effects of lifting the restriction of module stratification and of module nesting. We would also like to explore the possibility to use alternative optimization constructs offered by ASP solvers, such as `minimize` supported by `lparse` and `gringo`, in order not to be restricted by the

availability of weak constraints. Also on our agenda is analyzing the relationship between our framework and other proposals for modular ASP (see, e.g. [22, 13]). Finally, we would also like to implement a system that supports consequence module programming.

References

1. Marek, V.W., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm – A 25-Year Perspective. (1999) 375–398
2. Niemelä, I.: Logic programming with stable model semantics as a constraint programming paradigm. AMAI **25**(3–4) (1999) 241–273
3. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2002)
4. Gelfond, M.: Representing knowledge in A-Prolog. In: Computational Logic: From Logic Programming into the Future. LNCS 2408, (2002) 413–451
5. Gebser, M., Liu, L., Namasivayam, G., Neumann, A., Schaub, T., Truszczyński, M.: The first answer set programming system competition. In: LPNMR'07. LNCS 4483, (2007) 3–17
6. Bravo, L., Bertossi, L.E.: Logic programs for consistently querying data integration systems. In: IJCAI 2003, (2003) 10–15
7. Saccà, D.: Multiple total stable models are definitely needed to solve unique solution problems. Inf. Process. Lett. **58**(5) (1996) 249–254
8. Faber, W., Woltran, S.: Manifold answer-set programs for meta-reasoning. In: LPNMR'09. LNCS 5753, (2009) 115–128
9. Buccafurri, F., Leone, N., Rullo, P.: Enhancing disjunctive datalog by constraints. IEEE Trans. Knowl. Data Eng. **12**(5) (2000) 845–860
10. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. TOCL **7**(3) (2006) 499–562
11. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. **9**(3/4) (1991) 365–386
12. Oikarinen, E., Janhunen, T.: Achieving compositionality of the stable model semantics for smodels programs. TPLP **8**(5-6) (2008) 717–761
13. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Modular nonmonotonic logic programming revisited. In: Proceedings of the ICLP'09. LNCS 5649, (2009) 145–159
14. Reiter, R.: On closed world data bases. In: Logic and Databases. Plenum Press (1978) 55–76
15. Goldman, R.P., Boddy, M.S.: Expressive planning and explicit knowledge. In: AIPS'96, AAAI Press (1996) 110–117
16. Smith, D.E., Weld, D.S.: Conformant Graphplan. In: AAAI'98, AAAI Press (1998) 889–896
17. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning: Semantics and complexity. TOCL **5**(2) (2004) 206–263
18. Son, T.C., Tu, P.H., Gelfond, M., Morales, A.R.: An approximation of action theories of and its application to conformant planning. In: LPNMR'05. LNCS 3662, (2005) 172–184
19. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning, II: the DLV$^{\mathcal{K}}$ system. Artif. Intell. **144**(1–2) (2003) 157–211
20. Balduccini, M.: A general method to solve complex problems by combining multiple answer set programs. In: Proceedings ASPOCP'09. (2009)
21. Lifschitz, V., Turner, H.: Splitting a logic program. In: ICLP'94, MIT Press (1994) 23–37
22. Oikarinen, E.: Modularity in Answer Set Programs. PhD thesis, Helsinki University of Technology, Finland (2008)

A Pragmatic Programmer's Guide to Answer Set Programming

Martin Brain, Owen Cliffe* and Marina De Vos*

Department of Computer Science
University of Bath
Bath, BA2 7AY, UK
{mjb,occ,mdv}@cs.bath.ac.uk

Abstract. With the increasing speed and capacity of answer set solvers and showcase applications in a variety of fields, Answer Set Programming (ASP) is maturing as a programming paradigm for declarative problem solving. Comprehensive programming methodologies have been developed for procedural and object-oriented paradigms to assist programmers in developing their programs from the problem specification. In many cases, however it is not clear how, or even if, such methodologies can be applied to answer set programming. In this paper, we present a first and rather pragmatic methodology for ASP and illustrate our approach through the encoding of graphical puzzle.

1 Introduction

Answer Set Programming (ASP) is a declarative programming paradigm based on the answer set semantics [16, 1]. Like other declarative programming languages, the programmer specifies *what* needs to be achieved, rather than *how* it should be achieved. It therefore lends itself naturally to applications in the domain of artificial intelligence, such as plan generation and reasoning in agents. In ASP, programs are written in *AnsProlog* and describe the requirements for the solutions of certain problem. The answer sets of the program are interpreted to give theses solutions. The possible answer sets for an *AnsProlog* input program are computed with a program called a solver. Current solvers include SMODELS [19, 21], DLV [11, 12], CLASP [14], CMODELS [17], SUP [18].

A report by the Working group on Answer Set Programming (WASP)[1] acknowledges that better tools are required to support programming in this paradigm [20]. However in order to identify the aspects that require better support, and thus develop the appropriate tools to support them, a better understanding of the programming process is needed.

The process of engineering solutions to problems in declarative languages differs from conventional procedural software engineering in many regards. Conventional software engineering approaches (e.g. UML) (but excluding more recent agile approaches) focus on building specifications of data structures and functional units in advance, before proceeding to their implementation. However the declarative approach used in

* Work supported by the ALIVE project (FP7 215890)

[1] http://wasp.unime.it/

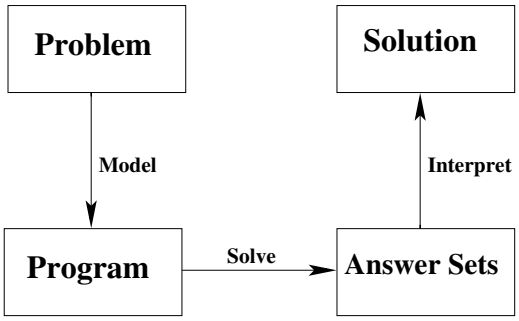

Fig. 1. The Four Box Diagram

AnsProlog means that programs essentially act as their own specification. Thus the process of understanding, decomposing and encoding problem structures (the specification) is necessarily blurred with the process of solving the problem itself (the implementation).

AnsProlog is increasingly being used to solve practical problems both in and out of the academic domain. At present it is our experience that developers who use these systems to solve real problems tend to develop either without a specific methodology, or build their own methodological process. Just as the community is trying to reach consensus on language standards [22] and intermediate formats for program representation and such [15, 22], we feel there is a need to reach a similar consensus on practical processes by which programs are developed and maintained. However, as with all programing techniques we cannot claim to know *the best* way to solve problems using ASP, all we can talk about is how *we* write programs and relate our experiences working in a number of domains. Thus, in this paper we do not try to document "best practice", simply "our practice". We hope that this can be the start of a discussion and that we, as a community, can start building up an idea of best practice.

The structure of the remainder of paper is as follows, we start by addressing issues relating to the whole process of problem solving using ASP, specifically we focus on methodological questions relating to how problems are captured. We then discuss some specific observations relating to problem encodings themselves through a guided example. Finally we look at the process of resolving problems within existing *AnsProlog* programs (debugging).

2 Towards a Development Process

ASP can be described via the four box diagram, as shown in Figure 1. One starts with a problem which is modelled as a *AnsProlog* program. This program is passed then to a solver. The answer sets are then interpreted to obtain the solutions.

A common problem we have found when encouraging students and collaborators from other fields to become active involved in developing applications using ASP is that while they understand how the tools work (the solving link in Figure 1) and finished

models (the program box), they do not understand how to go from their problem to a program (the modelling link). As far as we know there is no documentation that we can point them to and say "this is how to solve a problem using ASP". The authors of [8] give a very nice break down of how a logical model is structured and is currently the best resource for this. However they present the models as a finished product and do not discuss the process, reasoning or tools that created them.

3 The Methodology

In this section we provide a detailed description of our methodology for using ASP to solve problems. As a running example we will use the Hashiwokakero puzzle. The puzzles creators, Nikoli[2] describe the puzzle as follows:

> Hashiwokakero is played on a rectangular grid with no standard size, although the grid itself is not usually drawn. Some cells start out with (usually encircled) numbers from 1 to 8 inclusive; these are the islands. The rest of the cells are empty.
>
> The goal is to connect all of the islands into a single connected group by drawing a series of bridges between the islands. The bridges must follow certain criteria:
>
> 1. Connect islands (the dots with numbers) with as many bridges as the number in the island.
> 2. There can be no more than two bridges between two islands.
> 3. Bridges cannot go across islands or other bridges.
> 4. The bridges will form a continuous link between all the islands.

We also use examples from our music composition system ANTON [3] and our superoptimisation application, TOAST [5] which are among the largest applications using ASP.

Although some debugging tools exist [4], we have come to believe that a more incremental, test-driven [2] approach allows for easy verification at every stage. While it does not make debugging tools superfluous, a systematic approach prevents programmers from making certain errors and increases productivity.

3.1 Step 1: Start Simple

We advocate starting with a simplified model of the problem that to be solved, because it is much easier to build up a correct model into something more complicated than it is to fix a complicated but broken program. For example, in the case of ANTON we started out composing one part for 8 notes. This point cannot be stressed enough, start simple, start laughably simple, and work upwards.

[2] http://www.nikoli.com/en/puzzles/hashiwokakero/rule.html

Example 1. To start encoding our puzzle, we need to decide which literals to use to represent each concept. We need to consider what information will be in the instances, what the constraints are and what information you wish to infer. In this case, we decided to use the following:

Atom	Concept represented
Instance Specific Information	
`col(X)`	There is a column labelled X (ascending integers from 1).
`row(Y)`	There is a row labelled Y (ascending integers from 1).
`island(X,Y,N)`	There is an island in column X, row Y with value N.
Information to be Inferred	
`singleHorizontal(X,Y)`	There is a single horizontal bridge in column X, row Y.
`doubleHorizontal(X,Y)`	There is a double horizontal bridge in column X, row Y.
`singleVertical(X,Y)`	There is a single vertical bridge in column X, row Y.
`doubleVertical(X,Y)`	There is a double vertical bridge in column X, row Y.

At this stage, our program now looks like as Listing 1.

```
row(1..height).
col(1..width).
```

Listing 1. The first step towards our Hashiwokakero program

Deciding the meaning of the base literals should be enough to create and visualise (see next step) a problem instance. It is best to start with as small an instance as is meaningful, as otherwise it will be difficult to check by hand. This also helps mitigate any scaling issues during development. Given we are advocating incremental development, one does not want to have to wait more than a few seconds per run during development, so one needs to pick an appropriately sized example.

Listing 2 shows (with modified formatting) an instance of our puzzle.

3.2 Step 2: Visualisation

The next step is a visualisation or post-processing mechanism. This is effectively the interpretation link in the four box diagram. Post-processing and visualisation are oft-neglected parts of the development process, but very important ones as they allow the developer to see the program and its development in terms of the initial problem and solution, allowing a much more intuitive development process. This stage closes the semantic gap between the program encoding and the problem domain and aids debugging as it allows programmers to see what they wrote and easily compare it with what they intended to write. As argued in [6], when writing programs there is often a mismatch between the answer set you get and what you expected. Visualisation makes it much easier to see what one has, and specifically it often makes it easier to see *when* what one

```
#const height=13.
#const width=13.

uniqueStart(1,1).

island(1,1,2).   island(3,1,4).   island(5,1,3).
island(1,4,2).   island(3,4,3).   island(5,3,2).
island(1,6,1).   island(3,6,5).   island(5,5,2).
island(1,10,2).  island(3,8,4).   island(5,8,4).
island(1,13,3).  island(3,10,2).  island(5,10,3).
                 island(3,12,1).  island(5,12,2).

island(6,4,2).   island(7,1,1).   island(8,6,1).   island(9,1,2).
island(6,6,2).   island(7,3,3).   island(8,8,3).   island(9,3,2).
island(6,11,2).  island(7,5,5).   island(8,11,4).  island(9,5,3).
island(6,13,3).  island(7,7,2).   island(8,13,1).  island(9,7,2).
                                                   island(9,10,3).

island(10,2,3).  island(11,5,4).  island(12,1,1).  island(13,2,1).
island(10,4,3).  island(11,7,4).  island(12,4,1).  island(13,7,2).
island(10,11,4). island(11,10,2). island(12,6,2).  island(13,10,3).
island(10,13,2).                  island(12,8,3).  island(13,13,2).
                                  island(12,11,3).
```

Listing 2. An Hashiwokakero instance

has is incorrect. How the visualisations/interpretations are produced is up to the programmer and different types of program will lend themselves to different visualisation and processing approaches. One possibility is using a scripting language (e.g. Perl) and ASCII art. A more sophisticated choice would be ASPVIZ [9], a ASP visualisation tool using *AnsProlog* as its representation language. GraphViz [3] is also frequently a useful tool for visualising problems solutions.

The utility of visualisation cannot be overstated. It is often the most time consuming part of development but it does pay dividends by simplifying the process of isolating programming errors. After this it should be possible to visualise all of the elements of the full search space.

Example 2. The visualisation program written in ASPVIZ for Hashiwokakero can be found in Listing 3 (see [9] for a description of the language). The rules produce graphical artifacts for each island, shown as an ellipse containing the number of required bridges and the connecting bridges. An example of the output, for the program and instance given so far can be found in Figure 2.

3.3 Step 3: The Search Space Generator

The first part of the program to be written should be the search space generator. This tends to be a set of choice rules that define the search space of the problem in question[4]; to start with, this should give an answer set for each possible element of the search space. In the case when the problem being solved is co-NP, search generation is especially important. Working with programs with no answer sets is problematic. This should be obvious if one accepts the argument [6] that one knows there is a bug if there

[3] http://www.graphviz.org/

[4] We are of the opinion that all uses of ASP should effectively be phrased as search problems. As a consequence, we see the 'hard' part of the problem as the search phase. We note that there is a view that the procedural, data processing side is more significant.

```
% Draw islands and numbers
draw_ellipse(dflb,p(X*16,Y*16),14,14) :- island(X,Y,N).
draw_text(dflf,c,c,p(X*16,Y*16),N):- island(X,Y,N).

% Draw single bridges
draw_line(dflb,p(X*16,(Y*16)-8),p(X*16,(Y*16)+8)):-
                    verticalBridge(X,Y), not doubleVertical(X,Y).
draw_line(dflb,p((X*16)-8,Y*16),p((X*16)+8,Y*16)) :-
                    horizontalBridge(X,Y), not doubleHorizontal(X,Y).

% Draw double bridges
draw_line(dflb,p((X*16)-8,Y*16+2),p((X*16)+8,Y*16+2)) :- doubleHorizontal(X,Y).
draw_line(dflb,p((X*16)-8,Y*16-2),p((X*16)+8,Y*16-2)) :- doubleHorizontal(X,Y).

draw_line(dflb,p(X*16+2,(Y*16)-8),p(X*16+2,(Y*16)+8)):- doubleVertical(X,Y).
draw_line(dflb,p(X*16-2,(Y*16)-8),p(X*16-2,(Y*16)+8)):- doubleVertical(X,Y).
```

Listing 3. The visualisation program for Hashiwokakero

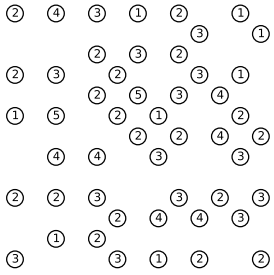

Fig. 2. Visualisation of an unsolved instance of the Hashiwokakero puzzle

is a difference between the expected and given answers. When the expected result is that no answer sets are produced and one gets no answer sets, how does one know if the code is working correctly or whether one has introduced a contradiction? Just using the number of branches or other solver performance stats is not reliable. Thus one wants to spend as much time as possible *with* answer sets, to the point of commenting out some constraints or deliberately working with instances that are 'incorrect'.

Example 3. We now need to add rules to generate the search space. The program up to this stage can be found in Listing 4. Adding these first rules allows us to see the solution taking shape as we build the model.

We now run our program together with the instance provided in Listing 2. From this, we obtain an answer set for every combination of bridges (a very large number). At this point, we do not yet care whether bridges join up or not. We can exploit the randomisation functionality of most answer set solvers to generate a small selection of answer sets which are visualised in Figure 3.

3.4 Commenting

Comments are very important during development. *AnsProlog* is very expressive and allows modelling of some very sophisticated concepts with very little code. Without

```
row(1..height).
col(1..width).

%% We need to know which squares contain island
%% So we project out the value of the island
isIsland(X,Y) :- island(X,Y,N).

%% For each square that is not an island, pick whether it should contain
%% a bridge or not and if so, what kind of bridge.
1 { empty(X,Y), singleHorizontal(X,Y), doubleHorizontal(X,Y),
    singleVertical(X,Y), doubleVertical(X,Y) } 1 :- row(Y), col(X),
                                                    not isIsland(X,Y).
```

Listing 4. Adding the search space for Hashiwokakero

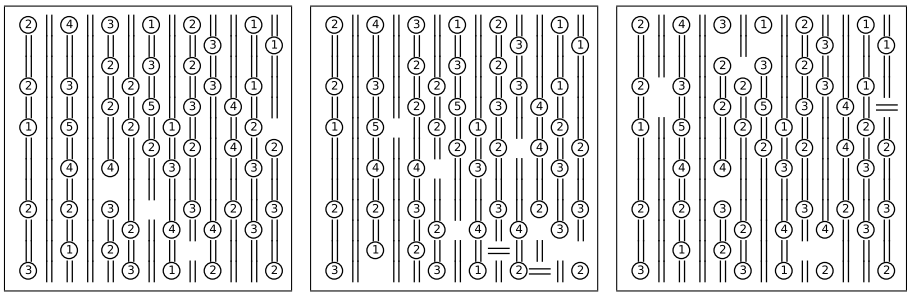

Fig. 3. Example outputs for random bridge assignments

comments this compactness can lead to terseness and unreadability (commonly referred to as write-only code in procedural languages). Normally each rule or group of related rules will be preceded by a comment, saying (in natural language) what aspect or idea from the problem it is intended to encode and (in the few cases where the encoding is complicated enough), how it is encoded. As well as making the program more intelligible, heavy commenting is very important for debugging and maintenance as it expresses, as much as is possible, what was intended by the code. As noted in [6] a bug in an *AnsProlog* program is a mismatch between what is written and what was intended. When no expression of intent is available except for the rules, then only the programmer who wrote the code can debug it, and generally only while writing it. Maintenance and further development become very difficult as first one has to infer what the programmer was thinking when they wrote the code. Given the (frequently) small semantic gap between the comments and the code, natural language generation may offer an interesting route to program support/debugging tools. If what is written does not provide a correct description of the problem, then generally one can see which part of the description is a problem. We also tend to use comments to record what each of the key propositions presents about the world, i.e "move(T,X,Y) is known if the move at time step T is to position (X,Y)". Ideally it should be possible to read the comments top to bottom as a description of the model.

```
row(1..height).
col(1..width).

%% We need to know which squares contain island
%% So we project out the value of the island
isIsland(X,Y) :- island(X,Y,N).

%% For each square that is not an island, pick whether it should contain
%% a bridge or not and if so, what kind of bridge.
1 { empty(X,Y), singleHorizontal(X,Y), doubleHorizontal(X,Y),
    singleVertical(X,Y), doubleVertical(X,Y) } 1 :- row(Y), col(X),
                                                    not isIsland(X,Y).

%% A single horizontal bridge must lead to another single bridge or an island
 :- singleHorizontal(X,Y), not singleHorizontal(X+1,Y), not isIsland(X+1,Y).

%% Likewise for double horizontal bridges
 :- doubleHorizontal(X,Y), not doubleHorizontal(X+1,Y), not isIsland(X+1,Y).

%% Similarly for vertical bridges
 :- singleVertical(X,Y), not singleVertical(X,Y+1), not isIsland(X,Y+1).
 :- doubleVertical(X,Y), not doubleVertical(X,Y+1), not isIsland(X,Y+1).
```

Listing 5. The first increment of Hashiwokakero

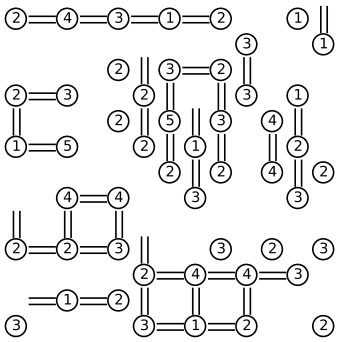

Fig. 4. An answer set visualisation of Hashiwokakero after the first increment.

3.5 Step 4: Incremental Development

Development now progresses in an incremental fashion, using very small increments. A set of rules (normally corresponding to one comment / idea) is written, then the solver is re-run and the answer set(s) visualised. This may seem slow but it removes a lot of bugs and saves time while trying to diagnose the causes of problems. In more complex applications the visualisation program may need to be updated along with the program to show the effect of new literals.

Example 4. Increment One: We now add rules saying that bridges must continue in their existing direction (and with the same width) or stop at an island. The program at this stage is shown in Listing 5. Note that these are only phrased in terms of increasing row / width as the decreasing case follows via symmetry. A visualisation of one the answer sets of our puzzle after we added the first increment is shown in Figure 4.

```
%% We want to be able to talk about either kind of horizontal / vertical bridge
horizontalBridge(X,Y) :- singleHorizontal(X,Y).
horizontalBridge(X,Y) :- doubleHorizontal(X,Y).
verticalBridge(X,Y) :- singleVertical(X,Y).
verticalBridge(X,Y) :- doubleVertical(X,Y).

%% A horizontal bridge cannot start a vertical bridge
 :- horizontalBridge(X,Y), verticalBridge(X,Y+1).

%% Neither can an empty square
 :- empty(X,Y), verticalBridge(X,Y+1).

%% Correspondingly for vertical bridges
 :- verticalBridge(X,Y), horizontalBridge(X+1,Y).
 :- empty(X,Y), horizontalBridge(X+1,Y).
```

Listing 6. The second increment of Hashiwokakero

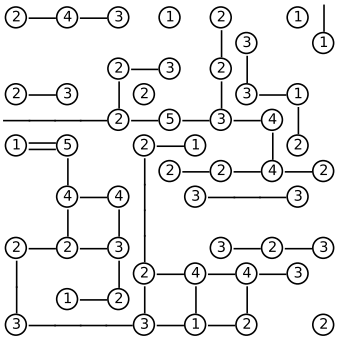

Fig. 5. An answer set visualisation of Hashiwokakero after the second increment.

Example 5. Increment Two: From the visualisation in Figure 4 is is clear that answer sets are beginning to resemble solutions but a few conditions are missing as there are cases where bridges are running into each other or starting from nothing. Thus we project the existence of horizontal / vertical bridges and prevent the other kind from starting immediately after them. Similarly for empty squares. The projection is a trade off between the number of literals and the number of rules. Given that rules are more often the limit of scalability, it is often worth doing. The rules added to the program are shown in Listing 6 and its output in Figure 5.

Example 6. Increment Three: The bridges are beginning to look correct, however the visualisation (Figure 5) points out a few conditions that have been missed. As we have been focusing on the 'next' location, we have edge conditions at the border. These are easily removed as there should never be vertical bridges in the top or bottom row, nor horizontal ones in either edge. The code in Listing 7 is added to our program which results in output like Figure 6.

Example 7. Increment Four: The bridges look correct but are a little sparse, so the next condition to add is the number of bridges coming in to each island. The obvious way to

```
%% No vertical bridges in the top or bottom rows
 :- verticalBridge(X,1).
 :- verticalBridge(X,height).

%% Likewise no horizontal bridges in the edge columns.
 :- horizontalBridge(1,Y).
 :- horizontalBridge(width,Y).
```

Listing 7. The third increment of Hashiwokakero

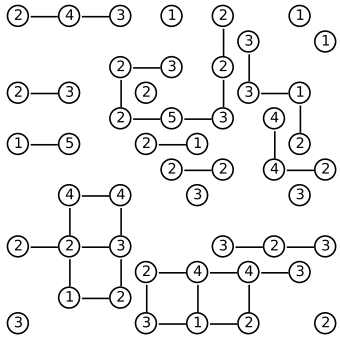

Fig. 6. An answer set visualisation of Hashiwokakero after the third increment.

do this is using a weight constraint. The additions to the program are shown in Listing 8. Again, the visualisation (Figure 7) makes it relatively easy to check that the effect of these rules matches the intend behind them.

Example 8. Fifth and final Increment: Now for the last constraint: all of the islands need to be connected. Careful examination of the output (Figure 7) will show that the islands are not connected. We formalise connectivity as reachability from a unique starting point and then require that all islands must be reachable. This raises the question of how to pick a unique starting point. We opt for the easiest solution of adding this to the instance requirements. The final addition to our program is shown in Listing 9. The entire program resulting in a complete (and unique) solution (Figure 8)

3.6 Coding Conventions

A number of coding conventions have been found to be useful and practical. Firstly literals are only ever used with one arity. This means that omissions of variables should be a detectable problem, rather than silently changing the meaning of the programs. Likewise, where at all possible, variable names are only ever used for one domain, i.e. `T` will always be a quantification over the `time()` domain, if `X` is a distance, it will always be used as one, in the same direction. These meanings are program specific, the key point is that it should be consistent across the program. Likewise the position, and ordering of variables should always be the same. If several literals refer to a 2D position at a given time step then they will all start with `(T,X,Y,...)`, etc.

```
%% For an island to be correct connected the number of bridges
%% entering a island must equal the weight at the island.
correctlyConnected(X,Y) :- island(X,Y,N),
                           N [ singleHorizontal(X-1,Y) = 1,
                               doubleHorizontal(X-1,Y) = 2,
                               singleHorizontal(X+1,Y) = 1,
                               doubleHorizontal(X+1,Y) = 2,
                               singleVertical(X,Y-1) = 1,
                               doubleVertical(X,Y-1) = 2,
                               singleVertical(X,Y+1) = 1,
                               doubleVertical(X,Y+1) = 2 ] N.

%% All islands must be correctly connected
 :- isIsland(X,Y), not correctlyConnected(X,Y).
```

Listing 8. The fourth increment of Hashiwokakero

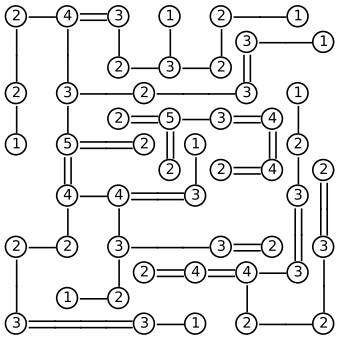

Fig. 7. An answer set visualisation of Hashiwokakero after the fourth increment.

3.7 Extension and Maintenance

Once the program is working, development shifts to extension and maintenance. While previously we were intentionally cutting down the space of possibilities, in the maintenance and enhancement phase the aim is often to make changes without inadvertently removing portions of the solution space (or making the program inconsistent, i.e. removing all of the solutions). Regression testing was used during the development of ANTON to address this problem. If we view ASP as solving a search problem, then the solution is a point in the search space and can be used as an initial condition for the search. If it gives the (single) solution that was found originally then it is still a solution, if it is inconsistent then the space of solutions has decreased, indicating a possible error. Once the original composition system was working, its output was checked by an expert and a collection of valid (and respectively, invalid) pieces was created. After each step of the development (one or more new rules), the regression tests could be run to automatically check that each of these was/was not obtainable and thus that no bugs had been introduced. This helped isolate bugs early in the development cycle and add a considerable degree of certainty to the development process.

```
%% The unique start is reachable
reachable(X,Y) :- uniqueStart(X,Y).

%% If an island is reachable then all neighbouring bridges are reachable
reachable(X-1,Y) :- isIsland(X,Y), reachable(X,Y), horizontalBridge(X-1,Y).
reachable(X+1,Y) :- isIsland(X,Y), reachable(X,Y), horizontalBridge(X+1,Y).
reachable(X,Y-1) :- isIsland(X,Y), reachable(X,Y), verticalBridge(X,Y-1).
reachable(X,Y+1) :- isIsland(X,Y), reachable(X,Y), verticalBridge(X,Y+1).

%% If a horizontal bridge is reachable then so are neighbouring bridges/islands
reachable(X-1,Y) :- horizontalBridge(X,Y), reachable(X,Y), horizontalBridge(X-1,Y).
reachable(X+1,Y) :- horizontalBridge(X,Y), reachable(X,Y), horizontalBridge(X+1,Y).
reachable(X-1,Y) :- horizontalBridge(X,Y), reachable(X,Y), isIsland(X-1,Y).
reachable(X+1,Y) :- horizontalBridge(X,Y), reachable(X,Y), isIsland(X+1,Y).

%% Likewise vertical bridges
reachable(X,Y-1) :- verticalBridge(X,Y), reachable(X,Y), verticalBridge(X,Y-1).
reachable(X,Y+1) :- verticalBridge(X,Y), reachable(X,Y), verticalBridge(X,Y+1).
reachable(X,Y-1) :- verticalBridge(X,Y), reachable(X,Y), isIsland(X,Y-1).
reachable(X,Y+1) :- verticalBridge(X,Y), reachable(X,Y), isIsland(X,Y+1).

%% Every island must be reachable
 :- isIsland(X,Y), not reachable(X,Y).
```

Listing 9. The final additions to Hashiwokakero

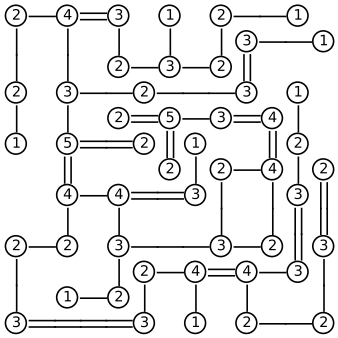

Fig. 8. The visualisation of the solution to the Hashiwokakero.

3.8 Generating Program Instances and Encodings

Literature on ASP often refers to splitting programs into instances and encoding. This is an intuitive division because in many cases there is a general class of problems and a particular instance of the problem we wish to solve. However there is often an assumption that this split is syntactic; facts for the instance and rules for the encoding. From our experience, this is not always the case. It is perfectly possible for the instance to include constants, extra constraints as well as atoms. Constants tend to be the most significant because these often alter the remaining program significantly.

In the case where the instance encoding is non-trivial we normally write a script that takes obvious, human readable parameters (as in ANTON) or a problem specific input format (as in TOAST) and generates at least the instance if not the whole program. This is where the distinction between instance and encoding begins to break down, as

part of the reason for having these scripts is combining the necessary program fragments, which are often instance specific. ANTON includes a `#include` construct to manage dependencies between program fragments and simplify program generation. In TOAST , the search program component included the architecture file which could be considered both instance and encoding. If one thinks of the visualisation scripts as the interpretation arrow in the four-box diagram Figure 1, then the program generation script that puts together the *AnsProlog* program is the representation arrow.

4 Debugging Programs

While the current debugging tools [6, 4] have their use, we have come to believe that existing tools do not yet provide the necessary support for programmers. The main problem is the amount of time the interface takes because one has to (implicitly or explicitly) mark which parts of the program are right and wrong. In practise this is too slow. One may think that it is no coincidence that none of the papers on debugging give more than a trivial, propositional example. All of the existing debuggers are primarily focused on computation of the reasons why certain properties of the answer sets hold. A topic that has received little research or implementation attention is the question of how to present the resultant symbolic information back to the user. This problem is discussed in more detail in [7].

However, with careful, incremental development and computing and visualising models after each change the programmer will already have a rough idea about what caused the problem and probably also why. The regression tests should catch the more obscure cases the programmer did not necessarily think of when going back and changing an older definitions.

5 Conclusion

In this paper we have presented an anecdotal methodology for software development in answer set programming. In summary:

1. Our approach is incremental and test-driven, allowing for early error detection.
2. After a suitable characterisation of the key concepts is determined, we recommend that a visualisation tool for the answer sets is used.
3. Having a graphical, auditory or more readable representation of the answer sets in terms of the problem domain will make it a lot easier to spot problems with the code.
4. Code documentation is even more important in ASP than it is in procedural languages. As errors are typically the result of a misalignment between the what was meant and what was written down, clearly stating what was intended in a human-readable form near to the relevant code makes identifying such errors easier.
5. The methodology is constructed around the notion that all our problems are search problems.
6. We model the entire search space first, before adding constraints to find acceptable solutions.

7. The characterisation of solutions and the constraints that reduce the search space should be done incrementally in order to detect bugs as early as possible.
8. Regression testing is used to verify that previous functionality is maintained after changes. A database containing good and bad solutions (cf. unit tests in procedural programming) can be constructed for this purpose.

While we find our approach to be useful in the context of our experience in developing complex *AnsProlog* systems, the methodology itself is informal and in no way comprehensive. The objective of this paper is to stimulate a discussion on possible best practice in the field.

One area which we have not considered is the ability to add assertions or 'sanity checks' that verify implicit properties of the encoding (for example the symmetric case of the constraints added in Listing 4). Even with the current state of the art in preprocessors [13], adding these as conventional constraints seems to bulk the program and result in much slower computation of solutions. Thus some technique for marking constraints as implicit, so that the encoding could be verified against them but they would be omitted from the normal program, could prove a useful development aid.

References

1. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.
2. Kent Beck. *Test-driven Development*. Addison-Wesley, 2003.
3. Georg Boenn, Martin Brain, Marina De Vos, and John ffitch. Automatic Composition of Melodic and Harmonic Music by Answer Set Programming. In *Proceedings of ICLP08*, December 2008.
4. M. Brain, M. Gebser, J. Pührer, T. Schaub, H. Tompits, and S. Woltran. "That is illogical captain!" – The debugging support tool spock for answer-set programs: System description. In De Vos and Schaub [10], pages 71–85.
5. Martin Brain, Tom Crick, Marina De Vos, and John Fitch. Toast: Applying answer set programming to superoptimisation. In *International Conference on Logic Programming*, LNCS. Springer, August 2006.
6. Martin Brain and Marina De Vos. Debugging Logic Programs under the Answer Set Semantics. In Marina De Vos and Alessandro Provetti, editors, *ASP05: Answer Set Programming: Advances in Theory and Implementation*, pages 142–152, Bath, UK, July 2005. Research Press International. Also available from http://CEUR-WS.org/Vol-142/files/page141.pdf.
7. Martin Brain and Marina De Vos. Answer set programming – a domain in need of explanation. In *Exact08: International Workshop on Explanation-aware Computing*, 2008.
8. M. Cayli, A. G. Karatop, E. Kavlak, H. Kaynar, F. Ture, and E. Erdem. Solving challenging grid puzzles with answer set programming. In *Proceedings of the 4th Workshop on Answer Set Programming: Advances in Theory and Implementation*, pages 175–190, Porto, Portugal, September 2007.
9. Owen Cliffe, Marina De Vos, Martin Brain, and Julian Padget. Aspviz: Declarative visualisation and animation using answer set programming. In *Logic Programming*, Lecture Notes in Computer Science, pages 724–728. Springer Berlin / Heidelberg, 2008.
10. M. De Vos and T. Schaub, editors. *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, 2007.
11. Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR system `dlv`: Progress report, comparisons and benchmarks. In Anthony G. Cohn,

Lenhart Schubert, and Stuart C. Shapiro, editors, *KR'98: Principles of Knowledge Representation and Reasoning*, pages 406–417. Morgan Kaufmann, San Francisco, California, 1998.
12. Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *KR'98: Principles of Knowledge Representation and Reasoning*, pages 406–417. Morgan Kaufmann, San Francisco, California, 1998.
13. Niklas En and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT05*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.
14. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/The MIT Press, 2007. Available at http://www.ijcai.org/papers07/contents.php.
15. Martin Gebser, Tomi Janhunen, Max Ostrowski, Torsten Schaub, and Sven Thiele. A versatile intermediate language for answer set programming. In Maurice Pagnucco and Michael Thielscher, editors, *Proceedings of the 12th International Workshop on Nonmonotonic Reasoning*, pages 150–159, Sydney, Australia, September 2008. University of New South Wales, School of Computer Science and Engineering, Techical Report, UNSW-CSE-TR-0819.
16. Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, pages 1070–1080, Seattle, Washington, August 1988. The MIT Press.
17. Y. Lierler and M. Maratea. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 2923 of *LNCS*, pages 346–350. Springer, 2004.
18. Yuliya Lierler. Abstract Answer Set Solvers. In *ICLP '08: Proceedings of the 24th International Conference on Logic Programming*, pages 377–391, Berlin, Heidelberg, 2008. Springer-Verlag.
19. I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal LP. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proceedings of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning*, volume 1265 of *LNAI*, pages 420–429, Berlin, July 28–31 1997. Springer.
20. Ilkka Niemelä, editor. *WASP WP3 Report: Language Extensions and Software Engineering for ASP*. 2005.
21. T. Syrjänen and I. Niemelä. The Smodels System. In *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, 2001.
22. Marek Truszczyński, Janhunen Tommi, Martin Brain, Wolfgang Faber, Marco Maratea, Axel Polleres, Torsten Schaub, and Roman Schindlauer. Language forum. In De Vos and Schaub [10], pages 3–39.

Yet Another Modular Action Language

Michael Gelfond and Daniela Inclezan

Computer Science Department
Texas Tech University
Lubbock, TX 79409 USA
Michael.Gelfond@ttu.edu, daniela.inclezan@ttu.edu

Abstract. The paper presents an action language, $\mathcal{ALM}$, for the representation of knowledge about dynamic systems. It extends action language $\mathcal{AL}$ by allowing definitions of new objects (actions and fluents) in terms of other, previously defined, objects. This, together with the modular structure of the language, leads to more elegant and concise representations and facilitates the creation of libraries of knowledge modules.

1 Introduction

This paper presents an extension, $\mathcal{ALM}$, of action language $\mathcal{AL}$ [1], [2] by simple but powerful means for describing modules. $\mathcal{AL}$ is an action language used for the specification of dynamic systems which can be modeled by transition diagrams whose nodes correspond to possible physical states of the domain and whose arcs are labeled by actions. It has a developed theory, methodology of use, and a number of applications [3]. However, it lacks the structure needed for expressing the hierarchies of abstractions often necessary for the design of larger knowledge bases and the creation of KR-libraries. The goal of this paper is to remedy this problem. System descriptions of our new language, $\mathcal{ALM}$, are divided into two parts. The first part contains *declarations* of the sorts, fluents, and actions of the language. Intuitively, it defines an uninterpreted theory of the system description. The second part, called *structure*, gives an interpretation of this theory by defining particular instances of sorts, fluents, and actions relevant to a given domain. Declarations are divided into *modules* organized as tree-like hierarchies. This allows for actions and fluents to be defined in terms of other actions and fluents. For instance, the action *carry* (defined in a dictionary as "to move while supporting") can be declared as a special case of *move*. There are two other action languages with modular structure. Language MAD [4],[5] is an expansion of action language $\mathcal{C}$ [6]. Even though $\mathcal{C}$ and $\mathcal{AL}$ have a lot in common, they differ significantly in the underlying assumptions incorporated in their semantics. For example, the semantics of $\mathcal{AL}$ incorporates the *inertia axiom* [7] which says that "*Things normally stay the same.*" The statement is a typical example of a default, which is to a large degree responsible for the very close and natural connections between $\mathcal{AL}$ and ASP [8]. $\mathcal{C}$ is based on a different assumption – the so called *causality principle* – which says that "*Everything true in the world must be caused.*" Its underlying logical basis is causal logic [9].

There is also a close relationship between ASP and $\mathcal{C}$ but, in our judgment, the distance between ASP and $\mathcal{ALM}$ is much smaller than that between ASP and $\mathcal{C}$. Another modular language is TAL-C [10], which allows definitions of classes of objects that are somewhat similar to those in $\mathcal{ALM}$. TAL-C, however, seems to have more ambitious goals: the language is used to describe and reason about various dynamic scenarios, whereas in $\mathcal{ALM}$ the description of a scenario and that of reasoning tasks are not viewed as part of the language.

The differences in the underlying languages and in the way structure is incorporated into $\mathcal{ALM}$, MAD and TAL-C lead to very different knowledge representation styles. We believe that this is a good thing. Much more research and experience of use is needed to discover if one of these languages has some advantages over the others, or if different languages simply correspond to and enhance different habits of thought.

This paper consists of two parts. First we define the syntax and semantics of an auxiliary extension of $\mathcal{AL}$ by so called defined fluents. The resulting language, $\mathcal{AL}_d$, will then be expanded to $\mathcal{ALM}$.

2 Expanding $\mathcal{AL}$ by Defined Fluents

2.1 Syntax of $\mathcal{AL}_d$

A *system description* of $\mathcal{AL}_d$ consists of a sorted *signature* and a collection of *axioms*. The signature contains the names for primitive *sorts*, a *sorted universe* consisting of non-empty sets of object constants assigned to each such name, and names for *actions* and *fluents*. The fluents are partitioned into *statics*, *inertial fluents*, and *defined fluents*. The truth values of statics cannot be changed by actions. Inertial fluents can be changed by actions and are subject to the law of inertia. Defined fluents are non-static fluents which are defined in terms of other fluents. They can be changed by actions but only indirectly. An atom is a string of the form $p(\bar{x})$ where p is a fluent and $\bar{x}$ is a tuple of primitive objects. A *literal* is an atom or its negation. Depending on the type of fluent forming a literal we will use the terms *static*, *inertial*, and *defined literal*. We assume that for every sort s and constant c of this sort the signature contains a static, $s(c)$. Direct causal effects of actions are described in $\mathcal{AL}_d$ by *dynamic causal laws* – statements of the form:

$$a \textbf{ causes } l \textbf{ if } p \tag{1}$$

where l is an inertial literal, a is an action name, and p is a collection of arbitrary literals. (1) says that if action a were executed in a state satisfying p then l would be true in a state resulting from this execution. Dependencies between fluents are described by *state constraints* — statements of the form:

$$l \textbf{ if } p \tag{2}$$

where l is a literal and p is a set of literals. (2) says that every state satisfying p must satisfy l. *Executability conditions* of $\mathcal{AL}_d$ are statements of the form:

$$\textbf{impossible } a_1, \ldots, a_k \textbf{ if } p \tag{3}$$

The statement says that actions $a_1, \ldots, a_k$ cannot be executed together in any state which satisfies p. We refer to l as the head of the corresponding rule and to p as its body. The collection of state constraints whose head is a defined fluent f is referred to as the *definition of* f. As in logic programming definitions, f is true in a state σ if the body of at least one of its defining constraints is true in σ. Otherwise, f is false. Finally, an expression of the form

$$f \equiv g \textbf{ if } p \tag{4}$$

where f and g are inertial or static fluents and p is a set of literals, will be understood as a shorthand for four state constraints:

f **if** p, g $\quad$ $\neg f$ **if** $p, \neg g$ $\quad$ g **if** p, f $\quad$ $\neg g$ **if** $p, \neg f$

An $\mathcal{AL}_d$ axiom with variables is understood as a shorthand for the set of all its ground instantiations.

2.2 Semantics of $\mathcal{AL}_d$

To define the semantics of $\mathcal{AL}_d$, we define the transition diagram $\mathcal{T}(\mathcal{D})$ for every system description $\mathcal{D}$ of $\mathcal{AL}_d$. Some preliminary definitions: a set σ of literals is called *complete* if for any fluent f either f or $\neg f$ is in σ; σ is called *consistent* if there is no f such that $f \in \sigma$ and $\neg f \in \sigma$. Our definition of the transition relation $\langle \sigma_0, a, \sigma_1 \rangle$ of $\mathcal{T}(\mathcal{D})$ *will be based on the notion of an answer set of a logic program.* We will construct a program $\Pi(\mathcal{D})$ consisting of logic programming encodings of statements from $\mathcal{D}$. The answer sets of the union of $\Pi(\mathcal{D})$ with the encodings of a state σ_0 and an action a will determine the states into which the system can move after the execution of a in σ_0.

The signature of $\Pi(\mathcal{D})$ will contain: (a) names from the signature of $\mathcal{D}$; (b) two new sorts: *steps* with two constants, 0 and 1, and *fluent_type* with constants *inertial*, *static*, and *defined*; and (c) the relations: $holds(fluent, step)$ ($holds(f, i)$ says that fluent f is true at step i), $occurs(action, step)$ ($occurs(a, i)$ says that action a occurred at step i), and $fluent(fluent\_type, fluent)$ ($fluent(t, f)$ says that f is a fluent of type t). If l is a literal, $h(l, i)$ will denote $holds(f, i)$ if $l = f$ or $\neg holds(f, i)$ if $l = \neg f$. If p is a set of literals $h(p, i) = \{h(l, i) : l \in p\}$; if e is a set of actions, $occurs(e, i) = \{occurs(a, i) : a \in e\}$.

Definition of $\Pi(\mathcal{D})$

(r1) For every constraint (2), $\Pi(\mathcal{D})$ contains:

$$h(l, I) \leftarrow h(p, I). \tag{5}$$

(r2) $\Pi(\mathcal{D})$ contains the closed world assumption for defined fluents:

$$\begin{aligned} \neg holds(F, I) \leftarrow\ & fluent(defined, F), \\ & \text{not } holds(F, I). \end{aligned} \tag{6}$$

(r3) For every dynamic causal law (1), $\Pi(\mathcal{D})$ contains:

$$h(l, I+1) \leftarrow h(p, I), \\ occurs(a, I). \tag{7}$$

(r4) For every executability condition (3), $\Pi(\mathcal{D})$ contains:

$$\neg occurs(a_1, I) \text{ v } \ldots \text{ v } \neg occurs(a_k, I) \leftarrow h(p, I). \tag{8}$$

(r5) $\Pi(\mathcal{D})$ contains the Inertia Axiom:

$$holds(F, I+1) \leftarrow fluent(inertial, F), \\ holds(F, I), \\ \text{not } \neg holds(F, I+1). \tag{9}$$

$$\neg holds(F, I+1) \leftarrow fluent(inertial, F), \\ \neg holds(F, I), \\ \text{not } holds(F, I+1). \tag{10}$$

(r6) and the following rules:

$$fluent(F) \leftarrow fluent(Type, F). \tag{11}$$

$$\leftarrow fluent(F), \text{not } holds(F, I), \text{not } \neg holds(F, I). \tag{12}$$

$$\leftarrow fluent(static, F), holds(F, I), \neg holds(F, I+1). \tag{13}$$

$$\leftarrow fluent(static, F), \neg holds(F, I), holds(F, I+1). \tag{14}$$

(The last four encodings ensure the completeness of states – (11) and (12) – and the proper behavior of static fluents – (13) and (14)). This ends the construction of $\Pi(\mathcal{D})$. Let $\Pi_c(\mathcal{D})$ be a program constructed by rules (r1), (r2), and (r6) above. For any set σ of literals, σ_{nd} denotes the collection of all literals of σ formed by inertial and static fluents. $\Pi_c(\mathcal{D}, \sigma)$ is obtained from $\Pi_c(\mathcal{D}) \cup h(\sigma_{nd}, 0)$ by replacing I by 0.

Definition 1 (State). A set σ of literals is a *state* of $\mathcal{T}(\mathcal{D})$ if $\Pi_c(\mathcal{D}, \sigma)$ has a unique answer set, A, and $\sigma = \{l : h(l, 0) \in A\}$.

Now let σ_0 be a state and e a collection of actions.

$$\Pi(\mathcal{D}, \sigma_0, e) =_{def} \Pi(\mathcal{D}) \cup h(\sigma_0, 0) \cup occurs(e, 0) \ .$$

Definition 2 (Transition). A *transition* $\langle \sigma_0, e, \sigma_1 \rangle$ is in $\mathcal{T}(\mathcal{D})$ iff $\Pi(\mathcal{D}, \sigma_0, e)$ has an answer set A such that $\sigma_1 = \{l : h(l, 1) \in A\}$.

To illustrate the definition we briefly consider

Example 1 (Lin's Briefcase). ([11])
The system description defining this domain consists of: (a) a signature containing the sort name *latch*, the sorted universe $\{l_1, l_2\}$, the action *toggle*(*latch*), the inertial fluent *up*(*latch*) and the defined fluent *open*, and (b) the following axioms:

$toggle(L)$ **causes** $up(L)$ **if** $\neg up(L)$
$toggle(L)$ **causes** $\neg up(L)$ **if** $up(L)$
$open$ **if** $up(l_1), up(l_2)$.

One can use our definitions to check that the system contains transitions
$\langle\{\neg up(l_1), up(l_2), \neg open\}, toggle(l_1), \{up(l_1), up(l_2), open\}\rangle$,
$\langle\{up(l_1), up(l_2), open\}, toggle(l_1), \{\neg up(l_1), up(l_2), \neg open\}\rangle$, etc.

Note that a set $\{\neg up(l_1), up(l_2), open\}$ is not a state of our system.

System descriptions of $\mathcal{AL}_d$ not containing defined fluents are identical to those of $\mathcal{AL}$. For such descriptions our semantics is equivalent to that of [12], [13]. (To the best of our knowledge, [12] is the first work which uses ASP to describe the semantics of action languages. The definition from [1],[13] is based on rather different ideas.) Note that the semantics of $\mathcal{AL}_d$ is non-monotonic and hence, in principle, the addition of a new definition could substantially change the diagram of $\mathcal{D}$. The following proposition shows that this is not the case. To make it precise we will need the following definition from [14].

Definition 3 (Residue). Let $\mathcal{D}$ and $\mathcal{D}'$ be system descriptions of $\mathcal{AL}_d$ such that the signature of $\mathcal{D}$ is part of the signature of $\mathcal{D}'$. $\mathcal{D}$ is a *residue* of $\mathcal{D}'$ if restricting the states and actions of $\mathcal{T}(\mathcal{D}')$ to the signature of $\mathcal{D}$ establishes an isomorphism between $\mathcal{T}(\mathcal{D})$ and $\mathcal{T}(\mathcal{D}')$.

Proposition 1. Let $\mathcal{D}$ be a system description of $\mathcal{AL}_d$ with signature Σ, $f \notin \Sigma$ be a new symbol for a defined fluent, and $\mathcal{D}'$ be the result of adding to $\mathcal{D}$ the definition of f. Then $\mathcal{D}$ is a residue of $\mathcal{D}'$.

3 Syntax of $\mathcal{ALM}$

A *system description*, $\mathcal{D}$, of $\mathcal{ALM}$ consists of the *system's declarations* (a non-empty set of *modules*) followed by the *system's structure*.

system description *name*
  declarations of *name*
    $[module]^+$
  structure of *name*
    structure description

A *module* can be viewed as a collection of declarations of *sort*, *fluent* and *action* classes of the system, i.e.

module *name*
  sort declarations
  fluent declarations
  action declarations

If the system declaration contains only one module then the first line above can be omitted. In the next two subsections we will define the declarations and the structure of a system description $\mathcal{D}$.

3.1 Declarations of $\mathcal{D}$

(1) A *sort declaration* of $\mathcal{ALM}$ is of the form

$$s_1 \ : \ s_2$$

where s_1 is a sort name and s_2 is either a sort name or the keyword **sort**[1]. In the latter case the statement simply declares a new sort s_1. In the former, s_1 is declared as a subsort of sort s_2.

The sort declaration section of a module is of the form

sort declarations
  $[sort\ declaration]^+$

(2) A *fluent declaration* of $\mathcal{ALM}$ is of the form

$f(s_1, \ldots, s_k)$: *type* **fluent**
  axioms
    $[state\ constraint\ .]^+$
end of f

where f is a fluent name, $s_1, \ldots, s_k$ is a list of sort names, and *type* is one of the following keywords: **static**, **inertial**, **defined**. If the list of sort names is empty we omit the parentheses and simply write f. The remaining part – consisting of the keyword **axioms** followed by a non-empty list of state constraints of $\mathcal{AL}_d$ and the line starting with the keywords **end of** – is optional and can be omitted. The statement declares the fluent f with parameters from $s_1, \ldots, s_k$ respectively as static, inertial, or defined.

The fluent declaration section of a module is of the form

fluent declarations
  $[fluent\ declaration]^+$

(3) An *action declaration* of $\mathcal{ALM}$ is of the form

a_1 : a_2
  attributes
    $[attr \ : \ sort]^+$
  axioms
    $[law\ .]^+$
end of a_1

where a_1 is an action name, a_2 is an action name or the keyword **action**, *attr* is an identifier used to name an attribute of the action, and *law* is a dynamic causal law or an executability condition similar to the ones of $\mathcal{AL}_d$ [2]. If $a_2 =$ **action**,

[1] Syntactically, names are defined as identifiers starting with a lower case letter.

[2] Due to space limitations, we only allow executability conditions of $\mathcal{ALM}$ for single actions, i.e. statements of the form **impossible** a_1 **if** p**.**

the first statement declares a_1 to be a new action class. If a_2 is an action name then the statement declares a_1 as a special case of the action class a_2. The two remaining sections of the declaration contain the names of attributes of this action, and causal laws and executability conditions for actions from this class. Both the attribute and the axiom part of the declaration are optional and can be omitted. With respect to axioms, the difference between $\mathcal{ALM}$ and $\mathcal{AL}_d$ is that in $\mathcal{AL}_d$ actions are understood as action instances while here they are viewed as action *classes*. Also, in $\mathcal{ALM}$ in addition to literals, the bodies of these laws can contain attribute atoms: expressions of the form $attr = c$, where $attr$ is the name of an attribute of the action and c is an element of the corresponding sort. The action declaration section of a module is of the form

action declarations
$[action\ declaration]^+$

The set of sort, fluent and action declarations from the modules of the system description $\mathcal{D}$ will be called the *declaration* of $\mathcal{D}$ and denoted by $decl(\mathcal{D})$. In order to be "well-defined" the declaration of a system description $\mathcal{D}$ should satisfy certain natural conditions designed to avoid circular declarations and other unintuitive constructs. To define these conditions we need the following notation and terminology:
Sort declarations of $decl(\mathcal{D})$ define a directed graph $S(\mathcal{D})$ such that $\langle sort_2, sort_1\rangle \in S(\mathcal{D})$ iff $sort_1 : sort_2 \in \mathcal{D}$. Similarly, the graph $A(\mathcal{D})$ is defined by action declarations from $decl(\mathcal{D})$. We refer to them as the *sort* and *action hierarchies* of $\mathcal{D}$.

Definition 4. The declaration, $decl(\mathcal{D})$, of a system description $\mathcal{D}$ is called *well-formed* if

1. The sort and action hierarchies of $\mathcal{D}$ are trees with roots **sort** and **action** respectively.
2. If $decl(\mathcal{D})$ contains the declarations of $f(s_1, \ldots, s_k)$ and $f(s'_1, \ldots, s'_k)$ then $s_i = s'_i$ for every $1 \leq i \leq k$.
3. If $decl(\mathcal{D})$ contains the declaration of action a with attributes $attr_1 : s_1$, ..., $attr_k : s_k$ and the declaration of action a with attributes $attr'_1 : s'_1$, ..., $attr'_m : s'_m$ then $k = m$, and $attr_i = attr'_i$ and $s_i = s'_i$ for every $1 \leq i \leq k$.

From now on we only consider system descriptions with well-formed declarations.

3.2 Structure of $\mathcal{D}$

The structure of a system description $\mathcal{D}$ defines an interpretation of the sorts, fluents, and actions declared in the system's declaration. It consists of the definitions of the sorts and actions of $\mathcal{D}$, and truth assignments for the statics of $\mathcal{D}$. The sorts are defined as follows:

sorts
$[constants \in\ s]^+$

where *constants* is a non-empty list of identifiers not occurring in the declarations of $\mathcal{D}$ and s is a sort name. We will refer to them as *objects* of $\mathcal{D}$. The definition of the sorts is followed by the definition of actions:

actions
[*instance description*]$^+$

where an *instance description* is defined as follows:

instance $a_1(t_1, \ldots, t_k)$ **where** *cond* : a_2
$attr_1 := t_1$
$\ldots$
$attr_k := t_k$

where $attr_1, \ldots, attr_k$ are attributes of an action class a_2 or of an ancestor of a_2 in $A(\mathcal{D})$, t's are objects of $\mathcal{D}$ or variables – identifiers starting with a capital letter –, and *cond* is a set of static literals. An instance description without variables will be called an *action instance*. An instance description containing variables will be referred to as an *action schema*, and viewed as a shorthand for the set of action instances, $a_1(c_1, \ldots, c_k)$, obtained from the schema by replacing the variables $V_1, \ldots, V_k$ by their possibles values $c_1, \ldots, c_k$. We say that an *action instance* $a_1(c_1, \ldots, c_k)$ *belongs to the action class* a_2 *and to any action class which is an ancestor of* a_2 *in* $A(\mathcal{D})$. Finally, we define statics as:

statics
[*state constraint* .]$^+$

where the head of the state constraint is an expression of the form $f(c_1, \ldots, c_k)$ (where f is a static fluent and $c_1, \ldots, c_k$ are properly sorted elements of the universe of $\mathcal{D}$), and the body of the state constraint is a collection of similar expressions. As usual, if the list is empty the keyword **statics** should be omitted.

Example 2. [Basic Travel]
Let us now consider an example of a system description of $\mathcal{ALM}$.

system description *basic_travel*

declarations of *basic_travel*

module *basic_geometry*

sort declarations

areas : **sort**

fluent declarations

within(*areas*, *areas*) : **static fluent**
axioms
$within(A_1, A_2)$ **if** $within(A_1, A)$, $within(A, A_2)$.
$\neg within(A_2, A_1)$ **if** $within(A_1, A_2)$.
$\neg within(A_1, A_2)$ **if** $disjoint(A_1, A_2)$.

end of *within*

$disjoint(areas, areas)$: **static fluent**
axioms
$disjoint(A_2, A_1)$ **if** $disjoint(A_1, A_2)$.
$disjoint(A_1, A_2)$ **if** $within(A_1, A_3), disjoint(A_2, A_3)$.
$\neg disjoint(A, A)$.
end of *disjoint*

module *move_between_areas*

sort declarations

$things$: **sort**
$movers$: $things$
$areas$: **sort**

fluent declarations

$loc\_in(things, areas)$: **inertial fluent**
axioms
$loc\_in(T, A_2)$ **if** $within(A_1, A_2),\ loc\_in(T, A_1)$.
$\neg loc\_in(T, A_2)$ **if** $disjoint(A_1, A_2),\ loc\_in(T, A_1)$.
end of *loc_in*

action declarations

$move$: **action**
attributes
$actor : movers$
$origin, dest : areas$
axioms
$move$ **causes** $loc\_in(O, A)$ **if** $actor = O,\ dest = A$.
impossible $move$ **if** $actor = O,\ origin = A,\ \neg loc\_in(O, A)$.
impossible $move$ **if** $origin = A_1,\ dest = A_2,\ \neg disjoint(A_1, A_2)$.
end of *move*

structure of *basic_travel*

sorts
$michael, john \in movers$
$london, paris, rome \in areas$

actions
instance $move(O, A_1, A_2)$ **where** $A_1 \neq A_2$: $move$
$actor := O$
$origin := A_1$
$dest := A_2$

statics
$disjoint(london, paris).\ disjoint(paris, rome).\ disjoint(rome, london).$

4 Semantics of $\mathcal{ALM}$

The semantics of a system description $\mathcal{D}$ of $\mathcal{ALM}$ is defined by mapping $\mathcal{D}$ into the system description $\tau(\mathcal{D})$ of $\mathcal{AL}_d$.

1. The signature, Σ, of $\tau(\mathcal{D})$:
The sort names of Σ are those declared in $decl(\mathcal{D})$. The sorted universe of Σ is given by the sort definitions from $\mathcal{D}$'s structure. We assume the domain closure assumption [15], i.e. the sorts of Σ will have no other elements except those specified in their definitions. An expression $f(c_1, \ldots, c_k)$ is a fluent name of Σ if $s_1, \ldots, s_k$ are the sorts of the parameters of f in the declaration of f from $decl(\mathcal{D})$, and for every i, $c_i \in s_i$. The set of action names of Σ is the set of all action instances defined by the structure of $\mathcal{D}$.

2. Axioms of $\tau(\mathcal{D})$:

(i) The state constraints of $\tau(\mathcal{D})$ are the result of grounding the variables of state constraints from $decl(\mathcal{D})$ and of static definitions from the structure of $\mathcal{D}$ by their possible values from the sorted universe of Σ. Already grounded static definitions from the structure of $\mathcal{D}$ are also state constraints of $\tau(\mathcal{D})$.

(ii) To define dynamic causal laws and executability conditions of $\tau(\mathcal{D})$ we do the following: For every action instance a_i of Σ and every action class a such that a_i belongs to a do:

For every causal law and executability condition L of a:

(a) Construct the expression obtained by replacing occurrences of a in L by a_i. For instance, the result of replacing *move* by $move(john, london, paris)$ in the dynamic causal law for the action class *move* will be:

$$move(john, london, paris) \ \textbf{causes} \ loc\_in(O, A) \ \textbf{if} \ actor = O, \ dest = A.$$

(b) Ground all the remaining variables in the resulting expressions by properly sorted constants of the universe of $\mathcal{D}$.
The above axiom will turn into the set containing:

$$move(john, london, paris) \ \textbf{causes} \ loc\_in(john, paris) \ \textbf{if} \ actor = john, \ dest = paris.$$
$$move(john, london, paris) \ \textbf{causes} \ loc\_in(michael, london) \ \textbf{if} \ actor = michael, \ dest = london.$$

etc.

(c) Remove the axioms containg atoms of the form $attr = y$ where y is not the value assigned to *attr* in the definition of instance a_i. Remove atoms of the form $attr = y$ from the remaining axioms.

This transformation turns the first axiom above into:

$$move(john, london, paris) \ \textbf{causes} \ loc\_in(john, paris).$$

and eliminates the second axiom.

It is not difficult to check that the resulting expressions are causal laws and executability conditions of $\mathcal{AL}_d$ and hence $\tau(\mathcal{D})$ is a system description of $\mathcal{AL}_d$.

5 Representing Knowledge in $\mathcal{ALM}$

In this section we illustrate the methodology of representing knowledge in $\mathcal{ALM}$ by way of several examples.

5.1 Actions as Special Cases

In the introduction we mentioned the action *carry*, defined as "to move while supporting". Let us now declare a new module containing such an action. The example will illustrate the use of modules for the elaboration of an agent's knowledge, and the declaration of an action as a special case of another action.

Example 3. [Carry]
We expand the system description *basic_travel* by a new module, *carrying_things*.

module *carrying_things*

sort declarations

$areas$: **sort**
$things$: **sort**
$movers$: $things$
$carriables$: $things$

fluent declarations

$holding(things, things)$: **inertial fluent**

$is\_held(things)$: **defined fluent**
axioms
$is\_held(O)$ **if** $holding(O_1, O)$.
end of is_held

$loc\_in(things, areas)$: **inertial fluent**
axioms
$loc\_in(T, A) \equiv loc\_in(O, A)$ **if** $holding(O, T)$.
end of loc_in

action declarations

$move$: **action**
attributes
$actor : movers$
$origin, dest : areas$
axioms
impossible $move$ **if** $actor = O,\ is\_held(O)$.
end of $move$

carry : *move*
attributes
carried_thing : *carriables*
axioms
impossible *carry* **if** $actor = O,\ carried\_thing = T,\ \neg holding(O,T)$.
end of *carry*

grip : **action**
attributes
actor : *movers*
patient : *things*
axioms
grip **causes** $holding(C,T)$ **if** $actor = C,\ patient = T$.
impossible *grip* **if** $actor = C,\ patient = T,\ holding(C,T)$.
end of *grip*

Similarly for action *release*.

Let us add this module to the declarations of *basic_travel* and update the structure of *basic_travel* by the definition of sort *carriables*:

$suitcase \in carriables$

and a new action

instance $carry(O,T,A)$: *carry*
$actor := O$
$carried\_thing := T$
$dest := A$

It is not difficult to check that, according to our semantics, the signature of $\tau(travel)$ of the new system description *travel* will be obtained from the signature of $\tau(basic\_travel)$ by adding the new sort, $carriables = \{suitcase\}$, new fluents like $holding(john, suitcase)$, $is\_held(suitcase)$ *etc.*, and new actions like $carry(john, suitcase, london)$, $carry(john, suitcase, paris)$, etc.

In addition, the old system description will be expanded by axioms:

$carry(john, suitcase, london)$ **causes** $loc\_in(john, london)$

$loc\_in(suitcase, london) \equiv loc\_in(john, london)$ **if** $holding(john, suitcase)$

Using Proposition 1 it is not difficult to show that the diagram of *travel* is a conservative extension of that for *basic_travel*.

5.2 Library Modules

The modules from the declaration part of *travel* are rather general and can be viewed as axioms describing our commonsense knowledge about motion. Obviously, such axioms can be used for problem solving in many different domains. It is therefore reasonable to put them in a *library* of commonsense knowlege.

A *library module* can be defined simply as a collection of modules available for public use. Such modules can be imported from the library and inserted in the declaration part of a system description that a programmer is trying to build. To illustrate the use of this library let us assume that all the declarations from *travel* are stored in a library module *motion*, and show how this module can be used to solve the following classical KR problem.

Example 4. [Monkey and Banana]
A monkey is in a room. Suspended from the ceiling is a bunch of bananas, beyond the monkey's reach. On the floor of the room stands a box. How can the monkey get the bananas? The monkey is expected to take hold of the box, push it under the banana, climb on the box's top, and grasp the banana.

We are interested in finding a reasonably general and elaboration tolerant declarative solution to this problem. The first step will be identifying sorts of objects relevant to the domain. Clearly the domain contains *things* and *areas*. The things move or are carried from one place to another, climbed on, or grasped. This suggests the use of the library module *motion* containing commonsense axiomatization of such actions. We start with the following:

system description *monkey_and_banana*

declarations of *monkey_and_banana*

import *motion* **from** *commonsense_library*

An $\mathcal{ALM}$ compiler will simply copy all the declarations from the library module *motion* into our system description. Next we will have:

module *main*

% The module will contain specific information about the problem domain.

sort declarations

$things$: **sort**	$boxes$: $carriables$
$movers$: $things$	$bananas$: $things$
$monkeys$: $movers$	$areas$: **sort**
$carriables$: $things$	$places$: $areas$

fluent declarations

$under(places, things)$: **static fluent**

$is\_top(areas, things)$: **static fluent**

$can\_reach(movers, things)$: **defined fluent**

axioms

$can\_reach(M, Box)$ **if** $monkeys(M)$,
$boxes(Box)$,
$loc\_in(M, L)$,
$loc\_in(Box, L)$.

$can\_reach(M, Banana)$ **if** $monkeys(M)$,
$bananas(Banana)$,
$boxes(Box)$,
$loc\_in(M, L_1)$,
$is\_top(L_1, Box)$,
$loc\_in(Box, L)$,
$under(L, Banana)$.

end of *can_reach*

action declarations

grip : **action**

attributes

$actor : movers$

$patient : things$

axioms

impossible *grip* **if** $actor = C,\ patient = T,\ \neg can\_reach(C, T)$.

end of *grip*

structure of *monkey_and_banana*

sorts

$m \in monkeys$
$b \in bananas$
$box \in boxes$
$floor, ceiling \in areas$
$l_1, l_2, l_3, l_4 \in places$

actions

instance $move(m, L)$ **where** $places(L)$: $move$
$actor := m$
$dest := L$

instance $carry(m, box, L)$ **where** $places(L)$: $carry$
$actor := m$
$carried\_thing := box$
$dest := L$

instance $grip(m, O)$ **where** $O \neq m$: $grip$
$actor := m$
$patient := O$

statics

$disjoint(L_1, L_2)$ **if** $places(L_1), places(L_2), L_1 \neq L_2$.
$disjoint(floor, ceiling)$.
$within(L, floor)$ **if** $places(L), \neg is\_top(L, box)$.
$under(l_1, b)$. $is\_top(l_4, box)$.

One can check that the system description defines a correct transition diagram of the problem. Standard ASP planning techniques can be used together with the ASP translation of the description to solve the problem.

6 Conclusions

In this paper we introduced a modular extension, $\mathcal{ALM}$, of action language $\mathcal{AL}$. $\mathcal{ALM}$ allows definitions of fluents and actions in terms of already defined fluents and actions. System descriptions of the language are divided into a general uninterpreted theory and its domain dependent interpretation. We believe that this facilitates the reuse of knowledge and the organization of libraries. We are currently working on proving some mathematical properties of $\mathcal{ALM}$ and implementing the translation of its theories into logic programs. Finally, we would like to thank V. Lifschitz for useful discussions on the subject of this paper.

References

1. Turner, H.: Representing Actions in Logic Programs and Default Theories: A Situation Calculus Approach. Journal of Logic Programming 31(1-3), 245–298 (1997)
2. Baral, C., Gelfond, M.: Reasoning Agents in Dynamic Domains. In: Workshop on Logic-Based Artificial Intelligence, pp. 257–279. Kluwer Academic Publishers, Norwell (2000)
3. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press (2003)
4. Lifschitz, V., Ren, W.: A Modular Action Description Language. In: Proceedings of AAAI-06, pp. 853-859. AAAI Press (2006)
5. Erdoğan, S.T., Lifschitz, V.: Actions as Special Cases. In: Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning, pp. 377–387 (2006)
6. Giunchiglia, E., Lifschitz, V.: An Action Language Based on Causal Explanation: Preliminary Report. In: Proceedings of AAAI-98, pp. 623–630. AAAI Press (1998)
7. Hayes, P.J., McCarthy, J.: Some Philosophical Problems from the Standpoint of Artificial Intelligence. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 4, pp. 463–502. Edinburgh University Press, Edinburgh (1969)
8. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–386 (1991)
9. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic Causal Theories. Artificial Intelligence 153, 105–140 (2004)
10. Gustafsson, J., Kvarnström, J.: Elaboration Tolerance Through Object-Orientation. Artificial Intelligence 153, 239–285 (2004)
11. Lin, F.: Embracing Causality in Specifying the Indirect Effects of Actions. In: Proceedings of IJCAI-95, pp. 1985–1993. Morgan Kaufmann (1995)
12. Baral, C., Lobo, J.: Defeasible Specifications in Action Theories. In: Proceedings of IJCAI-97, pp. 1441–1446. Morgan Kaufmann Publishers (1997)
13. McCain, N., Turner, H.: A Causal Theory of Ramifications and Qualifications. Artificial Intelligence 32, 57–95 (1995)
14. Erdoğan, S.T.: A Library of General-Purpose Action Descriptions. PhD thesis, The University of Texas at Austin (2008)
15. Reiter, T.: On Closed World Data Bases. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp.119–140. Plenum Press, New York (1978)

A Visual Tracer for DLV

Francesco Calimeri, Nicola Leone, Francesco Ricca, and Pierfrancesco Veltri

Dipartimento di Matematica, Università della Calabria, 87030 Rende, Italy
{calimeri,leone,ricca,pf.veltri}@mat.unical.it

Abstract. In software engineering, tracing is a specialized way for recording information about the execution of a program for debugging purposes. The more complex the system, the more difficult is developing a manageable, and thus practically useful, tracer. Answer Set Programming (ASP) systems represent no exception in this respect: the intrinsic complexity of reasoning required the design of elaborated evaluation algorithms.
In this paper, we present a suitable solution to the problem of tracing the execution of an ASP system and its implementation into the ASP system DLV. The tool herein presented features a graphical user interface and an on-line tracing method that puts it on the way between tracing and debugging. The range of applicability counts: bug fixing, system optimization, ASP-developing aids, and educational purposes.

1 Introduction

In software development, tracing is a specialized use of logging in order to record information about the execution of a program. This information is typically used by programmers for debugging purposes or (depending on the type and detail of information provided by the tracing system) by experienced system developers to diagnose problems or optimize implementations. Information provided by a tracing mechanism is usually employed by developers only, since usually there is not a standard output syntax, and the produced result might be either noisy (it might contain a lot of information which is useless for a specific purposes) or very long. Indeed, a common problem with tracing consists on the impossibility of isolating in a generic mechanism the information which is needed for detecting a specific problem, and thus a lot of useless information is inexorably printed.

Since the day of its first release the DLV system [1] is equipped with a simple tracing mechanism that is available to the developers; tracing instructions are instead removed by official release versions of DLV, for obvious optimization purposes. When enabled, this simple mechanism prints to the standard output a log of all internal system events and the value of some (relevant) internal variables. However, the unavoidable complexity of the algorithms employed for evaluating an ASP-program, and exploited by DLV, makes it quite difficult, or even impossible, to store and analyze such tracing output. It is worth noting that the evaluation of (non-ground) disjunctive ASP programs is a NEXPTIME^{NP} task [1, 2]; thus, the execution of an ASP system might produce a trace that is both inexorably large and difficult to handle, even in the case of small inputs.

In order to cope with this situation, we designed a general architecture for controlling and tracing the execution; we implemented such proposal into DLV, thus coming out with an advanced tracing system that has two important features: the execution of each task can be controlled (started, paused, or restarted), and the information produced can be set dynamically during the execution.

Our advanced tracer combines a graphical user interface and an on-line tracing method that puts it on the way between a tracing and a specialized system debugging tool.[1] The user can control the execution of the DLV system by means of suitable commands, and also ask the system to display some specific information, like e.g., the content of the internal data structures or the status of the system.

The resulting tool enjoys a wide range of applicability: bug fixing, system optimization, ASP-developing aids, and also educational purposes. Indeed, by following the trace of system execution, a program developer might analyze the behavior of the system and detect bugs or inefficient branches of the computation (as a by-product, a developer can be given an error-detection in the input specification). Moreover, by following the evaluation of a given encoding step-by-step, an ASP program developer might understand the reason for an inefficient evaluation, and tune its encoding for obtaining a more efficient of evaluation .

As already mentioned, the system features a graphical interface, that eases the interaction with the advanced tracing techniques; such interface makes the system suitable also for didactic purposes. Indeed, ASP systems act as a "black-box", and it is not possible to see what is really happening inside. Conversely, professors can explain evaluation techniques by preparing and showing a live-demo of their actual implementation. Students might follow the execution, step-by-step, on their own machines; moreover, "playing" with the system, they can gain a more direct understanding of the working principles, which is facilitated by a clear image of what is going on "under the hood".

Remark. It is worth noting that controlling and tracing the execution of an ASP system is quite a different task from debugging an ASP program. Indeed, the first task -the one addressed by the present work- aims at finding bugs and analyzing/controlling the behavior of an ASP system (which is a piece of software usually written in some imperative language); whereas, on the other hand, debugging an ASP program has the purpose of finding bugs in an logic program (which consists of ASP-language code to be then evaluated by means of an ASP system). While scanning the trace produced by the ASP system, the developer could also find errors within the logic program in input; but this kind of error-detection in the input specification is not the main purpose of our tracer. Techniques and tools specifically devised for debugging ASP programs (see e.g., [4–8]), are more appropriate than the tool herein presented for the second task. Note also that analogous considerations do not hold for the tracing-based debuggers for Prolog systems (see, e.g., [9]), where, due to the operational nature of the semantics of the supported language, tracing the execution results to be very useful also for debugging the logic program in input.

[1] This tool provides a method for tracing and debugging ASP-systems; the interested reader can find details on debugging techniques for ASP programs in [3].

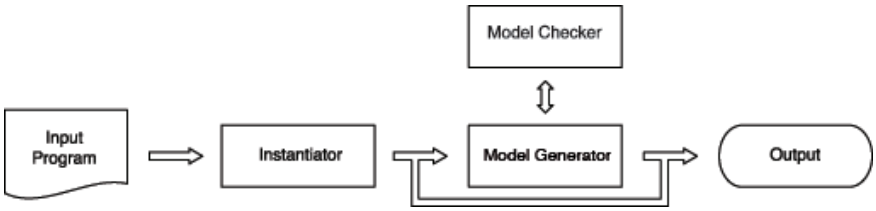

Fig. 1. DLV architecture.

The remainder of the paper is organized as follows: in Section 2 we describe the architecture of the DLV system; in Section 3 we describe our advanced tracing system; finally, we describe the system usage and the graphical user interface in Section 4.

2 The DLV System Architecture

We now outline the general architecture of DLV, which is schematically reported in Figure 1. Upon startup, the input specified by the user is parsed and transformed into the internal data structures of the system. In general, an input program $\mathcal{P}$ contains variables, and the first step of a computation of an ASP system is to eliminate these variables, generating a ground instantiation $ground(\mathcal{P})$ of $\mathcal{P}$. This variable-elimination process is called *instantiation* of the program (or *grounding*), and is performed by the *Instantiator* module (see Figure 1). A naïve Instantiator would produce the full ground instantiation $Ground(\mathcal{P})$ of the input, which is, however, undesirable from a computational point of view, as in general many useless ground rules would be generated. DLV therefore employs sophisticated techniques which are geared towards keeping the instantiated program as small as possible. A necessary condition is, of course, that the instantiated program must have the same answer sets as the original program. Moreover, if the input program is normal and stratified, the DLV Instantiator is able to directly compute its stable model (if it exists). The subsequent computations, which constitute the nondeterministic part of an ASP system, are then performed on $ground(\mathcal{P})$ by both the *Model Generator* and the *Model Checker*. Roughly, the former produces some "candidate" answer set, whose stability is subsequently verified by the latter. The Model generator of DLV implements a backtracking search algorithm, similar to a DPLL procedure of SAT solvers, which works directly on the ground instantiation of the input program. As previously pointed out, the *Model Checker* verifies whether an answer set candidate at hand is an answer set for the input program. This task is solved in DLV by calling a specialized procedure, since it is as hard as the problem solved by the Model Generator for disjunctive programs, while it is trivial for non-disjunctive programs.[2]

Finally, once an answer set has been found, DLV prints it in text format, and possibly the *Ground Reasoner* resumes in order to look for further answer sets. Note that, other

[2] However, there is also a class of disjunctive programs, called Head-Cycle-Free programs [10], for which the task solved by the Model Checker is provably simpler, which is exploited in the system algorithms.

traditional ASP-systems basically agree on the same general architecture even if they employ different techniques for implementing the same system components.

In sum, the evaluation of an ASP program in DLV can be divided in three *main tasks:* Instantiation, Model Generation, and Model Checking. Each of them requires to be *traced* for debugging purposes, and in the following Section we describe how our advanced tracing mechanisms deals with this requirement.

3 Advanced Tracing for DLV

The old DLV tracing mechanism is simple: it prints to standard output a log of all the internal system events, followed by the value of some relevant internal variables (e.g. current partial interpretation, current rule to be processed, etc.). More in detail, each module of DLV has a specialized set of traced variables and events, and the detail of the information produced can be set statically, for each module, to a given level ranging from 0 to 3 (minimum and maximum level roughly correspond to tracing-disabled and full information, respectively). This information allows for reconstructing an entire DLV execution, and/or to focus on the details of a single (or some) task, like e.g. Instantiation.

The main problem of this tracing system is however very easy to be seen: even small inputs can produce a huge tracing output, which might be either very difficult to be handled or even impossible to be stored in the file system. Indeed, tracing information is noisy, in the sense that the developer cannot isolate in a generic tracing mechanism the information which is needed for detecting a specific problem, and a lot of useless information is inexorably printed. Moreover, the unavoidable complexity of the algorithms employed for solving each single task of ASP-program evaluation can make even impossible to store and analyze the tracing output. Note that, Instantiation is in general EXPTIME-hard (the produced ground program being potentially of exponential size with respect to the input program), and both the Model Generator and the Model Checker implements a backtracking procedure that might require to execute (and, thus trace) an exponential number of operations (w.r.t. the size of the ground instantiation of the program!).

In order to cope with this situation, we designed and implemented in the DLV system a general architecture for controlling and tracing the execution, that has two important features: the execution of each task can be controlled (started, paused, or restarted) and the information to be printed can be set dynamically during the execution.

In Figure 2 is depicted the general architecture of our tracing method. In particular, the system is able to receive and recognize a sequence of commands in XML format from the standard input (or from a given file); those commands are recognized by the *command parser* module and inserted in a *command queue*. Each evaluation task of the system has been modified in order to stop periodically its normal execution in some predefined *breakpoints*, pick-up a new command from the queue and execute it. Commands might require to: set the system events to be printed, print the value of some status variable or the content of some internal data structure (e.g. one might ask whether some atom is true or false in the current partial interpretation); to continue the execution up to the next breakpoint; to undo the execution of some task; to terminate the process;

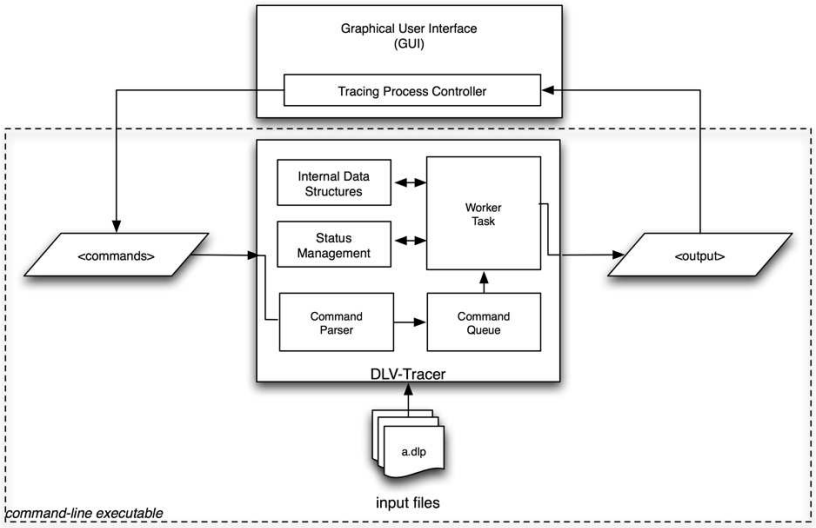

Fig. 2. Tracing Architecture.

etc.[3] The result of each command, together with the output generated by the system are printed to the standard output according to a specifically designed XML format. In this way, one can control dynamically the execution, and select the information it needs. Thus, the information to be printed is dramatically reduced, and the user can focus on the information regarding a specific moment of the execution. It is worth noting that, we carefully placed several breakpoints in the evaluation algorithms; breakpoints, that can be enabled of disabled by setting the *grain* level of execution, e.g. in the Model Generator one can decide to stop the execution at each choice point (lowest level of grain) or at the end of each propagation rule (finest level of control). The level of grain itself can be set dynamically by exploiting a specific command.

The DLV system enriched with this advanced tracing can be controlled either manually (by writing the commands from the console) or by exploiting a graphical user interface that is described in the next Section.

4 System Usage and Graphical User Interface

This section describes the usage of the herein presented system, then illustrates the Graphical User Interface (GUI) that has been conceived in order to ease interaction.

Commnad line interface. The tracing system is embedded into the DLV system, thus it features a command-line interface; data are exchanged through standard input/output

[3] For a complete listing of the available commands see Appendix 4.

streams. The tracer can be started by invoking DLV with "$-control$" option, and an optional XML input file containing a list of commands:

```
\$./dl -control [commandFile]
```

If input file is not provided, then the system awaits for commands on the standard input. We now report the snapshots of the command-line debugger while running on an example program.

Suppose now that we want to trace the Model Generator, initially, we start the DLV system with the option - control:

```
frankie@FrankiePC:~/Desktop/TesiSpecialistica/ControllerToServer/DifferenzeControllerVsOriginale/dlvServerIFNDEF$ dl -control
DLV [build DEV/Jul  6 2009   gcc 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu9)]

<message> System started </message>
```

The logic program we will give in input to DLV is stored in the file *prova.dl*, and contains the following rules: $a \vee b.\ c \vee d.\ e :- a.$. The image below shows how to set this program as input for DLV: just type the <*files*> tag.

```
frankie@FrankiePC:~/Desktop/TesiSpecialistica/ControllerToServer/DifferenzeControllerVsOriginale/dlvServerIFNDEF$ dl -control
DLV [build DEV/Jul  6 2009   gcc 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu9)]

<message> System started </message>
<files>"../prova.dl"</files>
```

Obviously, one might set more than one file by repeating the same command. Once the input is set, we can start the parser and then the Instantiator as follows:

```
frankie@FrankiePC:~/Desktop/TesiSpecialistica/ControllerToServer/DifferenzeControllerVsOriginale/dlvServerIFNDEF$ dl -control
DLV [build DEV/Jul  6 2009   gcc 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu9)]

<message> System started </message>
<files>"../prova.dl"</files>
<message> Input file established </message>

<parser/>
<message> Parser executed </message>

<grounding/>
<message> Grounding executed </message>
```

To enable tracing we set the tracing mode by inserting the <*sbs_w/*> command. The tag has two attributes: detail and grain. The detail level determines (as in the old tracing method) the quantity of information to be printed; whereas, the grain level determines the number of active breakpoints, respectively. In the following we set both Trace and Grain levels to two:

```
frankie@FrankiePC:~/Desktop/TesiSpecialistica/ControllerToServer/DifferenzeControllerVsOriginale/dlvServerIFNDEF$ dl -control
DLV [build DEV/Jul  6 2009   gcc 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu9)]

<message> System started </message>
<files>"../prova.dl"</files>
<message> Input file established </message>

<parser/>
<message> Parser executed </message>

<grounding/>
<message> Grounding executed </message>

<sbs_w detail="2" grain="2"/>
<message> Dlv is in step_by_step mode. MessageDetail = 2; TraceGrain = 2 </message>
```

Then, we run the Model Generator, the system executes a part of the computation and then DLV stops at the first breakpoint (enabled for this level of grain) and prints the tracing log. Then, we go to the next breakpoint with the command <*sbs_n*/>. In this case we will see the answer sets found written between braces (see Figure 3)

At each breakpoint we can modify the tracing configuration by adding or removing the information to be printed. In this example we require to add some additional information to the log (see Figure 4).

Going forward, Model generator finishes, and DLV waits for commands. In this example we reset the grounding of the program and then we restart both Grounding and Model Generator requiring to visualize the answer sets of the program without any tracing (see Figure 5).

Alternatively, the user might start the debugging session by exploiting the graphic interface.

Graphical User Interface. The GUI allows the user to exploit the full power of DLV Controller and Advanced Tracer in a simpler and more intuitive fashion. In the following, we describe how he interface is structured.

On the left of the main window is the management area for DLV input. On top of this area, a tree structure represents the part of the file system of the machine running DLV (Figure 7). The user can choose the root of such tree while starting the application, but if it can also be modified later.

Below the tree structure there is the list of the files given as input to DLV for current session (Figure 8).

A *session* is initialized when the system starts and it closes when the DLV process ends; a session can also be forced to close by the user through an appropriate button.

The central area of the main window is divided into two parts: the main available commands and a tracing management area are placed in the upper side, while the lower consists of a “Console”.

The tracing management area features some fields that remember the overall status of the application.

Fields are in charge of showing: the current status of DLV (Figure 10), the last command given by the user (Figure 11), DLV options that are currently enabled (Figure 12), and the grain and detail levels set during the last tracing analysis (Figure 13).

Under these fields a table, initially empty, is placed which show all the information printed by the Advanced Tracer during the session (Figure 14).

As described above, the information printed at each breakpoint changes according to the detail level; however, the user can also customize the current configuration by means of appropriate buttons.

In order to facilitate a better understanding of what is happening during the session, the part of the information displayed which is updated by the Tracer at each breakpoint is highlighted in red; the rest is gray (Figure 15).

For each piece of information name and current value are available. If a value is too long, it can be entirely displayed by clicking the "Enlarge" button.

The "Manage Info" button allows to customize the current display configuration for the information printed by the Tracer.

The GUI will show only the pieces of information explicitly included in the first list of Figure 16; nevertheless, such list can be customized by adding other pieces of information from the second list, or by removing currently selected pieces of information.

Finally, a "Console" is showed in the lowest area of the window (Figure 17). This text-area shows the output of the Controller as it is released; thus, when a command is invoked, the user can view the results.

5 Conclusion

In this paper we have presented an advanced tracing methodology which has been especially conceived for controlling and monitoring the execution of an ASP system.[4] We have implemented it on the DLV system, and developed a Graphical User interface that allows to manage tracing operations in a friendly environment.

The advanced tracing technique herein presented can be fruitfully exploited by system developers, with the purpose of finding bugs or optimizing the execution of internal algorithms. Moreover, tracing can be exploited by ASP program developers in order to optimize (and, in some cases, fix) input ASP programs: indeed, by following the trace of system execution, the ASP program developer might better understand the behavior of the exploited ASP system when a specific encoding is evaluated, and thus provide an alternative (hopefully more-efficiently-evaluable) encoding, or fix an incorrect one.

Thanks to its friendly interface, the advanced tracer might also be employed for didactic purposes; indeed, students can discover what is going on "under the hood", thus better understand underlying techniques and evaluation algorithms.

References

1. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7**(3) (2006) 499–562
2. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys **33**(3) (2001) 374–425

[4] Even though each ASP system features its own algorithms and techniques (and thus, also peculiar variables an data structures), the idea of controlling the execution by means of proper breakpoints and an external graphical interface which deals with the system by means of an XML syntax can be easily adapted and implemented into ASP systems different from DLV.

3. De Vos, M., Schaub, T., eds.: SEA'07: Software Engineering for Answer Set Programming. Volume 281., CEUR (2007) Online at http://CEUR-WS.org/Vol-281/.
4. Pontelli, E., Son, T.C., El-Khatib, O.: Justifications for logic programs under answer set semantics. TPLP **9**(1) (2009) 1–56
5. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A Meta-Programming Technique for Debugging Answer-Set Programs. In: AAAI'08, AAAI Press (2008) 448–453
6. Perri, S., Ricca, F., Terracina, G., Cianni, D., Veltri, P.: An integrated graphic tool for developing and testing DLV programs. In: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07). (2007) 86–100
7. Syrjänen, T.: Debugging Inconsistent Answer Set Programs. In: Proceedings of the 11th International Workshop on Non-Monotonic Reasoning, Lake District, UK (2006) 77–84
8. Brain, M., De Vos, M.: Debugging Logic Programs under the Answer Set Semantics. In: Proceedings ASP05 - Answer Set Programming: Advances in Theory and Implementation, Bath, UK (2005)
9. Roychoudhury, A., Ramakrishnan, C.R., Ramakrishnan, I.V.: Justifying proofs using memo tables. In: PPDP. (2000) 178–189
10. Ben-Eliyahu, R., Dechter, R.: Propositional Semantics for Disjunctive Logic Programs. AMAI **12** (1994) 53–87

A Appendix: Tracing Commands

In the following we report some of the most important tracing commands that allow to customize the tracing.

Customizing the Configuration Once the tracing mode has been set, or during the analysis phase, the information to be printed can be customized according the user's need. This is done by means of the following statements.

- `<show_info_tracer/>`
  shows all information concerning current configuration; this will be printed every time an active breakpoint is reached. The short version of this statement is `<show_t/>`.
- `<add_info_to_tracer>"infoName1; ...; infoNameN"</add_info_to_tracer>`
  add a piece of information to the current tracing configuration; information will be printed at each step from now on. The information to be added must be written between start tag and end tag. A hort version of this statement is available as `<a_to_t>"infoName1... infoNameN"</a_to_t>`.
- `<delete_info_of_tracer>"infoName1; ...; "infoNameN</delete_info_of_tracer>`
  removes a piece of information from the current tracing configuration. The short version is `<d_of_t>"infoName1... infoNameN"</d_of_t>`.
- `<empty_info_of_tracer/>`
  empties the current tracing configuration; this will force the tracer to print no information at all. The short version is `<e_of_t/>`.

Breakpoints Commands When DLV stops, at any breakpoint, the user can inspect and trace the execution by means of the following commands:

- `<step_by_step_next/>`
  allows to exit the current breakpoint and go forward to the next one. Short version is `<sbs_n/>`.
- `<step_by_step_continue/>`
  allows to leave the Tracing mode; hence, DLV goes ahead until the end of computation with no stops (thus ignoring any other breakpoint). The short version is `<sbs_c/>`.
- `<step_by_step_view>"InfoName"</step_by_step_view>`
  this command prints information "on-demand". Indeed, the user can ask the Tracer to print some pieces of information which are not contained in the current configuration. Short version is `<sbs_v>"InfoName"</sbs_v>`.

There are also some commands which are defined as "special"; these can be invoked by the user only at some specific breakpoints.

- `<step_by_step_go_back>"X"</step_by_step_go_back>`
  or
  `<sbs_gb>"X"</sbs_gb>`
  forces DLV to go back of X level; if L is the current level, the computation starts over from level: $L - X$. This command can be invoked only while the Model Generator is being traced; in particular, only at a breakpoint where DLV checks the stability of the current model, or when it is waiting for the next choice.
- `<step_by_step_stop_when>"X"</step_by_step_stop_when>`
  or
  `<sbs_sw>"X"</sbs_sw>`
  forces DLV to go ahead without any stop while the atom X is not true; once the atom X becomes true, the system will stop at next breakpoint. This command can be executed at any breakpoint, both during the Grounding or the Model Generation phases.
- `<step_by_step_go_component>"X"</step_by_step_go_component>`
  or
  `<sbs_gc>"X"</sbs_gc>`
  forces the system to go ahead and stop just before the evaluation of the component X has to start. It can be executed only during while the Grounding is traced, in particular only during the evaluation of the components of the input program.
- `<step_by_step_go_rule>"X"</step_by_step_go_rule>`
  or
  `<sbs_gr>"X"</sbs_gr>`
  forces the system to go ahead and stop just before the evaluation of the rule X has to start. It can be executed only while the Grounding is being traced, in particular during the evaluation of the components of the input program. This command has effect only if the grain level is set to 2: indeed, there are no stops at rule level with a lower grain level.

– `<step_by_step_go_constraint>"X"</step_by_step_go_constraint>`
  or
  `<sbs_gcn>"X"</sbs_gcn>`
  force the system to go ahead and stop just before the evaluation of the constraint X has to start. It can be executed only during while the Grounding is being traced, in particular during the evaluation of the components or the constraints of the input program. This command has effect only if the grain level is set to 2: indeed, there are no stops at constraint level with a lower grain level.
– `<step_by_step_go_wconstraint>"X"</step_by_step_go_wconstraint>`
  or
  `<sbs_gwcn>"X"</sbs_gwcn>`
  forces the system to go ahead and stop just before the evaluation of the weak constraint X has to start. It can be executed only while the Grounding is being traced, in particular during the evaluation of the weak constraints. This command has effect only if the grain level is set to 2: indeed, there are no stops at weak constraint level with a lower grain level.

```
frankie@FrankiePC:~/Desktop/TesiSpecialistica/ControllerToServer/DifferenzeControllerVsOriginale/dlvServerIFNDEF$ dl -control
DLV [build DEV/Jul  6 2009   gcc 4.0.2 20050808 (prerelease) (Ubuntu 4.0.1-4ubuntu9)]

<message> System started </message>
<files>"../prova.dl"</files>
<message> Input file established </message>

<parser/>
<message> Parser executed </message>

<grounding/>
<message> Grounding executed </message>

<sbs_w detail="2" grain="1"/>
<message> Dlv is in step_by_step mode. MessageDetail = 2; TraceGrain = 1 </message>

<model_generator/>
<message> Model generator start :
        <message>
                Breakpoint Reached.
                Well-Founded has been computed
        </message>
        <tracer>
        <MGCurrentChoice>The current choice has not yet initialized</MGCurrentChoice>
        <MGPositiveOrNegativeChoice>The current choice has not yet initialized</MGPositiveOrNegativeChoice>
        </tracer>
<sbs_n/>
    <message> Next step </message>

        <message>
                Breakpoint Reached.
                Wait for next choice
        </message>
        <tracer>
        <MGCurrentChoice>b</MGCurrentChoice>
        <MGPositiveOrNegativeChoice>Positive</MGPositiveOrNegativeChoice>
        </tracer>
   <message> Next step </message>

        <message>
                Breakpoint Reached.
                Wait for next choice
        </message>
        <tracer>
        <MGCurrentChoice>d</MGCurrentChoice>
        <MGPositiveOrNegativeChoice>Positive</MGPositiveOrNegativeChoice>
        </tracer>
<sbs_n/>
    <message> Next step </message>

{a, d, e}
        <message>
                Breakpoint Reached.
                Found stable model
        </message>
        <tracer>
        <MGCurrentChoice>d</MGCurrentChoice>
        <MGPositiveOrNegativeChoice>Positive</MGPositiveOrNegativeChoice>
        </tracer>
```

Fig. 3. Run the Model Generator.

```
<show_info_tracer/>
<message> The tracer can print these info:
          MGCurrentChoice,MGLevel,MGPositiveOrNegativeChoice,
          MGCurrentInterpretation,MGPTQueue,
          MGEuristicValues,MGPropagationLiteral,
          MGLookaheadFlag,MGUnfoundedSet,MGWellFoundedState,
          MGLookaheadFailFlag,MGLiteralsInferredInLastPropagation,
          GRAllComponents, GRCurrentComponent, GRInfoAboutPredicate,
          GRCurrentComponentOutput, GRCurrentIteration, GRInputOfCurrentIteration,
          GROutputOfCurrentIteration, GROriginalRule, GRReorderedRule, GRMatchingResult,
          GRMatchingBackTo, GRAllConstraint, GRConstraintFinalOutput, GROriginalConstraint,
          GRReorderedConstraint, GRCurrentConstraintOutput, GRAllWeakConstraint,
          GRWeakConstraintFinalOutput, GROriginalWeakConstraint, GRReorderedWeakConstraint,
          GRCurrentWeakConstraintOutput
</message>

<message> While the current info are:
          MGCurrentChoice,
          MGPositiveOrNegativeChoice,
          GRAllComponents,
          GRCurrentComponent,
          GRCurrentIteration,
          GRInputOfCurrentIteration,
          GROutputOfCurrentIteration,
          GRCurrentComponentOutput,
          GRAllConstraint,
          GRCurrentConstraintOutput,
          GRConstraintFinalOutput,
          GRAllWeakConstraint,
          GRWeakConstraintFinalOutput,
          GRCurrentWeakConstraintOutput
</message>

<add_info_to_tracer>"MGLevel;MGCurrentInterpretation"</add_info_to_tracer>
<message> Info added </message>
```

Fig. 4. Add tracing information.

```
{b, c}
        <message>
                Breakpoint Reached.
                Found stable model
        </message>
        <tracer>
        <MGCurrentChoice>d</MGCurrentChoice>
        <MGPositiveOrNegativeChoice>Negative</MGPositiveOrNegativeChoice>
        <MGLevel>1</MGLevel>
        <MGCurrentInterpretation>< TRUE (FROM UNDEFINED) = {c}, TRUE (FROM MUST_BE_TRUE) = {b}, MUST_BE_TRUE = {}, UNDEFINED = {}, FALSE = {a, d, e} ></MGCur
rentInterpretation>
        </tracer>
<mbs_n/>
    <message> Next step </message>

Model generator stop!</message>


<r_gr/>
<message> Grounding reset </message>

<grounding/>
<message> Grounding executed </message>

<model_generator/>
<message> Model generator start :
{a, d, e}

{a, c, e}

{b, d}

{b, c}
Model generator stop!</message>

<quit/>

<message> Found quit tag, Goodbye!!! </message>

frankie@FrankiePC:~/Desktop/TesiSpecialistica/ControllerToServer/DifferenzeControllerVsOriginale/dlvServerIFNDEF$
```

Fig. 5. End tracing.

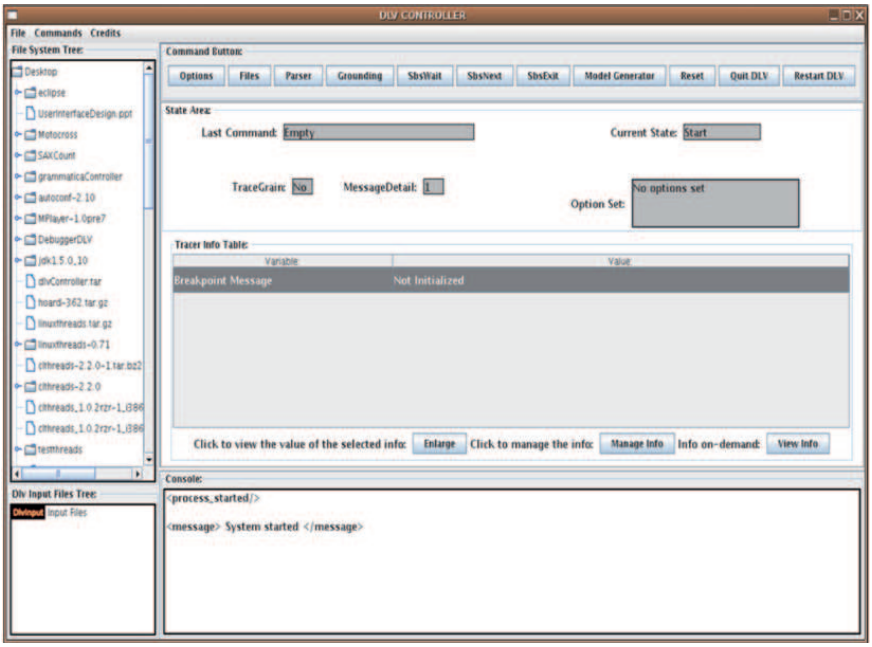

Fig. 6. GUI: Starting interface.

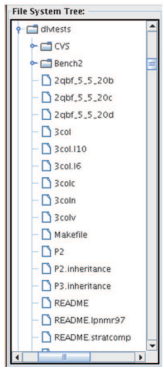

Fig. 7. GUI: Current State of DLV.

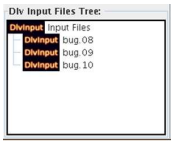

Fig. 8. GUI: Last Command Executed.

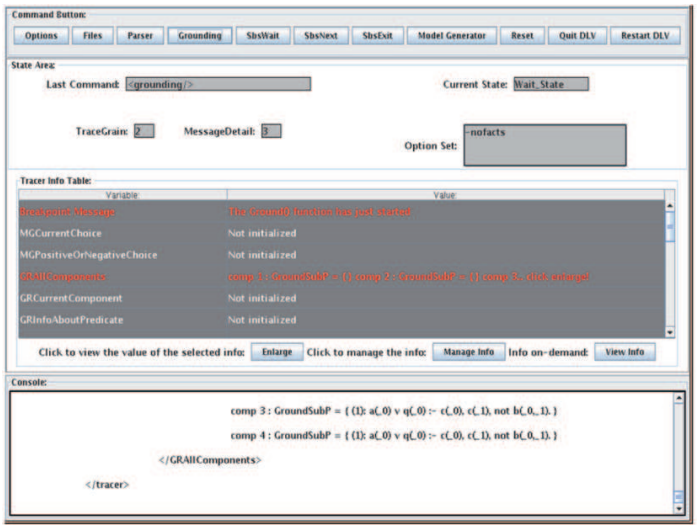

Fig. 9. GUI: Main Area of the Window.

Fig. 10. GUI: Current State of DLV.

Fig. 11. GUI: Last Command Executed.

Fig. 12. GUI: DLV Options Enabled.

Fig. 13. GUI: Grain and Detail Levels.

Fig. 14. GUI: Tracing Table at the Beginning.

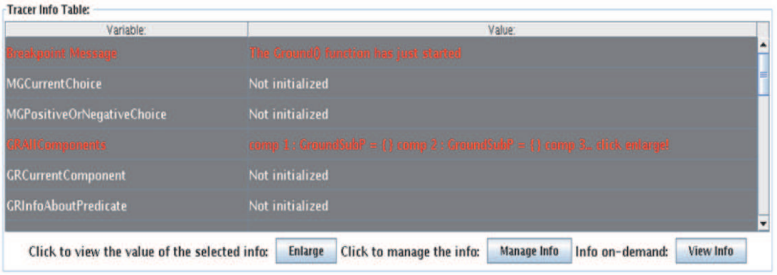

Fig. 15. GUI: Tracing Table during Analysis.

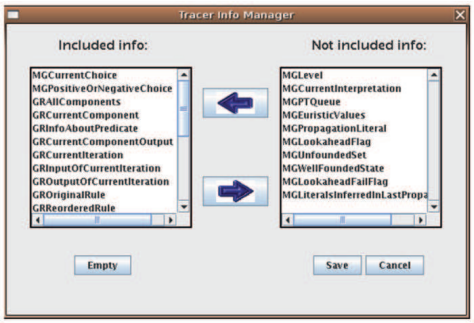

Fig. 16. GUI: Dialog for Information Management.

Fig. 17. GUI: Console.

Author Index