

Combining Agents, ASP and Planning

NICTA, Jul-Aug 2003

- **July and August with the exception of third week in July.**
- **Time:** Thursday, Friday, 14-16, starting on 3rd July 2003.
- Lecture Course is in the **first 3 weeks on theoretical issues in general agent systems and answer set programming**, emphasis on mathematical-logical foundations. **Remaining two weeks devoted to a particular agent system** and some demonstrations.
- `www.cs.man.ac.uk/~jdix/LECTURING/NICTA03.html`.

First and second week (Chapters 1–3)

The first part of this lecture course is mainly based on

Multi-Agent Systems

(Gerhard Weiss)

MIT Press, June 1999.

We describe **general methods** and **techniques**.

Third week (Chapter 4)

The second part of this lecture course is mainly based on

1. **Knowledge representation, reasoning and declarative problem solving with Answer sets**
(Chitta Baral), MIT Press, February 2003.

2. **Planning in Answer Set Programming using Ordered Task Decomposition**
(Jürgen Dix, Ugur Kuter and Dana Nau)
Theory and Practice of Logic Programming, to appear 2004.
<www.cs.umd.edu/users/ukuter/ASP_Planning/>

We give an introduction to the newly emerged paradigm of **Answer Set Programming** and illustrate it with recent research on how to realise HTN-planning in this paradigm.

Fourth and fifth week (Chapters 5–9)

The third part of this lecture course is mainly based on

Heterogenous Agent Systems

(Subrahmanian/Bonatti/Dix/Eiter/Kraus/Özcan/Ross)

MIT Press, August 2000.

We describe the **IMPACT approach** and its **underlying foundations**. We also give two demos and present an approach of monitoring agents through planning (using an ASP engine).

Overview (**Agent Systems in general**)

1. Introduction

2. Distributed Decision Making (2 Lectures)

3. Contract Nets, Coalition Formation

Overview (**Answer Set Programming**)

4. ASP: Foundations and an Application to Planning (2 Lectures)

Overview (**IMPACT**)

- 5. *IMPACT* Architecture**
- 6. Actions and Agent Programs**
- 7. Implementing Agents: An Application**
- 8. Agent Systems and Planning**
- 9. Extensions of IMPACT**

4 Answer Set Programming

4.1 Definite Logic Programs

4.2 Positive Disjunctive Logic Programs

4.3 AnsProlog vs. PROLOG

4.4 Answer Sets and their Semantics

4.5 Declarative Problem Solving Modules

4.6 DLV: Special features

4.7 HTN Planning

4.8 Realising HTN Planning in ASP

4.9 Benchmarks and Comparisons

4.10 Summary and References

Why not just Prolog? What went wrong with Prolog?

ASP as a new programming paradigm.

ASP is a **nonmonotonic** logic. It captures truth in **minimal** models. Thus it is well suited for describing transitive closure, exceptions etc.

I.e.: adding new information makes previously drawn conclusions useless (they have to be given up).

It is therefore stronger than classical logic.

Assessment 1 (Lab: Colouring)

How many possibilities are there to colour the map of (continental) Europe with at most 4 colours (such that adjacent countries get different colours)? Is it possible with just 3?

The same questions for colouring the United States of America (adjacent states get different colours).

Write a positive disjunctive program such that it computes all colourings no matter what the underlying facts are. Solve the problem by using DLV for computing the models.

Make yourself familiar with DLV by going through the online manual: <http://www.dbai.tuwien.ac.at/proj/dlv/>.

4.1 Definite Logic Programs

Programs without negation have nice properties: one unique model, which is minimal and can be computed with database technology. Prolog is based on it.

Definition 4.1 (Definite Logic Program)

A *definite* logic program consists of a finite number of *rules* of the form

$$A \leftarrow B_1, \dots, B_m,$$

where A, B_1, \dots, B_m are positive atoms (containing possibly free variables, like $p(X, Y, c)$). We call A the *head* of the rule and B_1, \dots, B_m its *body*. The comma represents conjunction \wedge . A might also be absent (resp. identical to \perp): we call such a rule a **constraint**.

We can think of a program as formalising our knowledge about the world and how the world behaves. Of course, we also want to derive new information, i.e. we want to ask queries:

Definition 4.2 (Query)

Given a definite program we usually have a definite query in mind that we want to be solved. A definite query Q is a conjunction of positive atoms $C_1 \wedge \dots \wedge C_l$ which we denote by

$$?- C_1, \dots, C_l.$$

These C_i may also contain variables. Asking a query Q to a program P means asking for all possible substitutions Θ of the variables in Q such that $Q\Theta$ follows from P . Θ is also called an answer to Q ($Q\Theta$ may still contain free variables).

Example 4.1 (Connectedness)

$edge(a, b) \leftarrow$
 $edge(b, c) \leftarrow$
 $edge(c, d) \leftarrow$
 $edge(b, e) \leftarrow$
 $connected(X, Y) \leftarrow edge(X, Y)$
 $connected(X, Y) \leftarrow edge(X, Z), connected(Z, Y)$

The facts $edge(\cdot, \cdot)$ are also called EDB relations (**extensional data base**), while the predicate $connect$ is called IDB relation (**intensional data base**).

A query is `?- connected(b, X)` , asking for all nodes that can be reached from b .

How are our programs related to classical predicate logic?

We can map a program-rule into classical logic by interpreting “ \leftarrow ” as material implication “ \rightarrow ” and universally quantifying.

Thus we view it as the universally quantified formula

$$B_1 \wedge \dots \wedge B_m \rightarrow A.$$

Let us apply this to Example 4.1 (What do we get?). Viewed as a predicate logic theory, we can not conclude that there are no other edges other than those stated explicitly.

In fact, we cannot conclude any negative information.

So the query $?- \neg connected(b, X)$ (enumerate all nodes that are not connected with b) would not return any result.

However, we would like to conclude, from the absence of positive information, that the negation holds.

A logic program-rule takes some orientation with it.

Principle 1 (Orientation) *If a ground atom A does not unify with some head of a program rule of P , then this atom is considered to be false. In this case we say that “**not** A ” is derivable from P to distinguish it from classical $\neg A$.*

The orientation principle is nothing but a weak form of *negation-by-failure*. Given an intermediate goal **not** A , we first try to prove A . But if A does not unify with any head, A fails and this is the reason to derive **not** A .

Can we come up with a simple procedural mechanism?

Example 4.2 (SLD-Resolution)

Let the program P_{SLD} consist of the following three clauses

$$(1) \quad p(X, Z) \leftarrow q(X, Y), p(Y, Z)$$

$$(2) \quad p(X, X)$$

$$(3) \quad q(a, b)$$

The query Q we are interested in is given by $p(X, b)$. I.e. we are looking for all substitutions Θ for X such that $p(X, b)\Theta$ follows from P .

Chapter 4: Answer Set Programming		← <u>$p(X, b)$</u>	Combining Agents, ASP and Planning, NICTA
	1	2	
	← $q(X, Y),$	<u>$p(Y, b)$</u>	$\overset{\perp}{[X/b]}$
	1	2	Success
	← $q(X, Y), q(Y, U),$	← <u>$q(X, b)$</u>	
	<u>$p(U, b)$</u>		
	1	2	3
← $q(X, Y), q(Y, U),$	← $q(X, Y),$	<u>$q(Y, b)$</u>	$\overset{\perp}{[X/a]}$
$q(U, V),$	<u>$p(V, b)$</u>		
1	2	3	Success
	← <u>$q(X, a)$</u>		
	Failure		
⋮	⋮		
4.1 Definite Logic Programs			139

The figure on page 138 illustrates the behaviour of SLD-resolution. We start with our query in the form $\leftarrow Q$. In any round the selected atom is underlined and emphasised in red: numbers 1, 2 or 3 indicate the number of the clause which the selected atom is resolved against. There are three different sorts of branches:

1. **infinite branches**,
2. branches that *end up with the empty clause*, and
3. branches that *end in a deadlock* (“*Failure*”): no applicable rule is left.

In this example we always resolve with the *last* atom in the goal under consideration. If we choose always the *first* atom in the goal, we will obtain, at least in this example, a *finite* tree.

Bottom-up approach

- We start first with rules with empty bodies (in our example these are all instantiations of rules (2) and (3)). We get as facts all atoms that are in the heads of rules with empty bodies (namely $p(a, a)$, $p(b, b)$, $q(a, b)$ in Example 4.2 on page 136).
- In the next round we use the facts that we computed before and try to let the rules “fire”, i.e. when their bodies are true, we add their heads to the atoms we already have (this gives us $p(a, b)$).

We introduce the immediate consequence operator T_P which associates to any Herbrand model another Herbrand model.

Example 4.3 (T_P)

Given a definite program P let $T_P : 2^{B_P} \mapsto 2^{B_P}; \mathcal{I} \mapsto T_P(\mathcal{I})$

$$T_P(\mathcal{I}) := \{A \in B_P : \begin{array}{l} \text{there is an instantiation of a rule in } P \\ \text{s.t. } A \text{ is the head of this rule and all} \\ \text{body-atoms are contained in } \mathcal{I} \end{array}\}$$

T_P computes the immediate consequences of program P .
Those that can be obtained in one step.

It turns out that T_P is monotone and continuous so that (by a general theorem of Knaster-Tarski) the least fixpoint is obtained after ω steps. Moreover we have

Theorem 4.1 (T_P and M_P)

$$M_P = T_P \uparrow^\omega = lfp(T_P).$$

In database applications, where the underlying language does not contain function symbols (DATALOG), the Herbrand universe is always finite. Under this condition the iteration stops after finitely many steps. In addition, rules of the form

$$p \leftarrow p$$

do not make any problems. They simply can not be applied or do not produce anything new. **Note that in the top-down approach, such rules give rise to infinite branches!**

Principle 2 (Elimination of Tautologies)

Suppose a program P has a rule which contains the same atom in its body as well as in its head. Then we can eliminate this rule without changing the semantics.

Note that our original aim was to find substitutions Θ such that $Q\Theta$ is derivable from the program P . This task as well as M_P is closely related to SLD:

Theorem 4.2 (Least model)

The following properties are equivalent:

- $P \models \forall Q\Theta$, i.e. $\forall Q\Theta$ is true in all models of P ,
- $M_P \models \forall Q\Theta$,

Example 4.4 (Transitive Closure (again))

Assume we are given a graph consisting of nodes and edges between some of them. We want to know which nodes are reachable from a given one. A natural formalisation of the property “reachable” would be

$$reachable(X) \leftarrow edge(X, Y), reachable(Y).$$

What happens if we are given the following facts

$$edge(a, b), \ edge(b, a), \ edge(c, d)$$

and $reachable(c)$? Of course, we expect that neither a nor b are reachable because there is no path from c to either a or b .

But SLDNF-Resolution does not derive “**not** $reachable(a)$ ”!

Note that its straightforward transformation into predicate logic does not work either:

$$reachable(X) \leftrightarrow (X \doteq c \vee \exists Y (reachable(Y) \wedge edge(Y, X)))$$

from which, together with our facts about the edge-relation, $\neg reachable(a)$ is indeed not derivable.

Note also that our Principle 2 on page 143 does not help, because it simply does not apply.

Let us motivate the principle of *partial evaluation* or GPPE^a. The query “**not** $reachable(a)$ ” leads to “ $reachable(a) \leftarrow edge(a, b), reachable(b)$ ” and “ $reachable(b)$ ” leads to “ $reachable(b) \leftarrow edge(b, a), reachable(a)$ ”. Both rules can be seen as definitions for $reachable(a)$ and $reachable(b)$ respectively.

^aGeneralised Principle of Partial Evaluation

So it should be possible to replace in these rules the body atoms of *reachable* by their definitions. Thus we obtain the two rules

$$reachable(a) \leftarrow edge(a, b), edge(b, a), reachable(a)$$

$$reachable(b) \leftarrow edge(b, a), edge(a, b), reachable(b)$$

that can both be eliminated by applying Principle 2 on page 143.

So we end up with a program that does neither contain *reachable(a)* nor *reachable(b)* in one of the heads. Therefore, according to Principle 1 on page 136 both atoms should be considered false.

Principle 3 (GPPE, (Brass and Dix 1994; Sakama and Seki 1994))

We say that a semantics satisfies GPPE, if the following transformation does not change the semantics. Replace a rule $A \leftarrow B^+$ where B^+ contains a distinguished atom B by the rules

$$A \leftarrow (B^+ \setminus \{B\}) \cup \mathcal{B}_i^+ \quad (i = 1, \dots, n)$$

where $B \leftarrow \mathcal{B}_i^+ \quad (i = 1, \dots, n)$ are all rules with head B .

4.2 Positive disjunctive Programs

So far so good. But a single model is often not what we want. Many problems have several solutions and that should be reflected by several models.

Idea: Allow disjunctions in the head of rules

Example 4.5 (Colourings)

```
node(X) :- edge(X,Y).
node(Y) :- edge(X,Y).

coloured(X,r) v coloured(X,g) v coloured(X,b) :- node(X).

:- edge(X,Y), coloured(X,C), coloured(Y,C).
```

Example 4.6 (Hamiltonian Cycle)

```
in_hm(X,Y) v out_hm(X,Y) :- start(X), arc(X,Y).
in_hm(X,Y) v out_hm(X,Y) :- reached(X), arc(X,Y).

:- in_hm(X,Y), in_hm(X,Y1), Y != Y1.
:- in_hm(X,Y), in_hm(X1,Y), X != X1.

:- arc(X, ), not reached(X).
```

How do we ensure in the above formalisations that a node is not coloured with two different colours (resp. a node is both in and out of the hamiltonian cycle)?

Definition 4.3 (Minimal Model)

A model \mathcal{A} is minimal wrt. predicates p_1, \dots, p_n and a theory T if there is no other model \mathcal{B} (with the same universe) such that for all i : $p_i^{\mathcal{B}} \subseteq p_i^{\mathcal{A}}$ and for at least one i_0 : $p_{i_0}^{\mathcal{B}} \subsetneq p_{i_0}^{\mathcal{A}}$.

What are the minimal models of the following programs?

1. $p(a) \vee p(b) \vee p(c)$
2. $p(a) \vee p(b) \vee p(c) \leftarrow p(a)$
3. $p(a) \vee p(b) \leftarrow p(c)$
 $p(c) \vee p(d) \leftarrow$
4. $p(a) \vee p(b)$
 $p(a) \vee p(c)$
5. $p(a) \vee p(b)$
 $p(b) \leftarrow p(a)$
 $p(a) \leftarrow p(b)$

Is the formula $p(a) \vee p(b)$ equivalent to $p(a) \leftarrow \neg p(b)$?

Example 4.7 (How to formulate the N-Queens problem?)

1. Use predicates $queen(\cdot)$, $row(\cdot)$, $col(\cdot)$. And $at(I, X, Y)$ meaning queen I is at row X and column Y .

2. **Enumerate:**

(a) $at(I, X, Y) \vee not_at(I, X, Y) \leftarrow queen(I), row(X), col(Y)$.

(b) Each queen is at most at one location.

(c) Each queen is at some location.

$$at(I, c_1, Y) \vee \dots \vee at(I, c_N, Y) \leftarrow queen(I), col(Y)$$

$$at(I, X, c_1) \vee \dots \vee at(I, X, c_N) \leftarrow queen(I), row(Y)$$

(d) No two queens are at the same location.

3. Eliminate:

- (a) No two distinct queens in same row.
- (b) No two distinct queens in same column.
- (c) No two queens attack each other.

4.3 AnsProlog and its variants

Where do ASP and PROLOG differ? What are ASP programs?

- PROLOG is a programming language based on predicate logic with function symbols.
- PROLOG does not handle negation-as-failure in a convincing way (recursion through negation).
- PROLOG has problems with infinite loops, non-declarative features (ordering of literals, cut operator).

- **AnsProlog** is purely declarative, based on answer sets and function-free predicate logic (DATALOG).
- There are efficient systems implementing it (DLV, smodels).
They also scale up.
- **Nondeterminism** is nicely captured by **multiple stable models**.
- Specification and programming are merely the same in **AnsProlog**.
- But: **answering a query does no more depend on the call-graph below it** ((Dix and Müller 1994b; Dix and Müller 1994a)).

In KR we do not only want to formulate negative queries, we also want to express *default-statements* of the form

Normally, unless something abnormal holds, then ψ implies ϕ .

How can we formulate such a statement as a logic program?

The most natural way is to use negation “**not**”

$$\phi \leftarrow \psi, \text{ not } ab$$

where ab stands for *abnormality*. Obviously, this forces us to extend definite programs by *negative* atoms, we call them default atoms.

Example 4.8 (Inheritance Hierarchies)

Suppose we know that birds typically fly and penguins are non-flying birds. We also know that Tweety is a bird. Now an agent is hired to build a cage for Tweety. Should the agent put a roof on the cage? After all it could be still the case that Tweety is a penguin and therefore can not fly, in which case we would not like to pay for the unnecessary roof. But under normal conditions, it should be obvious that one should conclude that Tweety is flying.

A natural axiomatisation is given as follows:

$$\begin{aligned}
 P_{Inheritance} : \quad & flies(X) \leftarrow bird(X), \quad \textbf{not } ab(r_1, X) \\
 & bird(X) \leftarrow penguin(X) \\
 & ab(r_1, X) \leftarrow penguin(X) \\
 & make_top(X) \leftarrow flies(X)
 \end{aligned}$$

together with some particular facts, like e.g. $bird(Tweety)$ and $penguin(Sam)$. The first rule formalises our default knowledge, while the third formalises that the default rule should not be applied in abnormal or exceptional cases. In our example, it expresses the famous *specificity principle* which says that more specific knowledge should override more general one.

For the query “ $make_top(Tweety)$ ” we expect the answer “yes” while for “ $make_top(Sam)$ ” we expect the answer “no”.

We are looking for minimal models: models that only make things true that need to be made true. Everything else which does not have a reason to be true should stay false.

Definition 4.4 ($\text{AnsProlog}^{\text{not}}$, $\text{AnsProlog}^{\text{or}}$, $\text{AnsProlog}^{\text{or}, \text{not}}$)

The language $\text{AnsProlog}^{\text{not}}$ consists of rules of the form

$$A \leftarrow B_1, \dots, B_m,$$

where A is a positive atom, and B_1, \dots, B_m are positive atoms which may be preceded by **not** (all atoms can contain free variables, like $p(X, Y, c)$). We call A the *head* of the rule and B_1, \dots, B_m its *body*.

We often also allow that A is absent (resp. identical to \perp): then we call the language $\text{AnsProlog}^{\text{not}, \perp}$.

The language $\text{AnsProlog}^{\text{or}}$ consist of rules of the form

$$A_1 \text{ or } A_2 \text{ or } \dots A_n \leftarrow B_1, \dots, B_m,$$

where A_i and B_i are positive atoms (which may contain free variables, like $p(X, Y, c)$). Similar to the above, we define

AnsProlog^{or,⊥}.

Finally the language **AnsProlog**^{or,not} (resp. **AnsProlog**^{or,not,⊥}) consist of rules where we allow both disjunctions in the head as well as negations in the body.

A set of rules in **AnsProlog**[⋅] is also called an **AnsProlog**[⋅] program or an **AnsProlog**[⋅] theory.

We also use the notation **AnsProlog**[∅] to denote definite logic programs without constraints.

Example 4.9 (Van Gelder’s Example)

Assume we are describing a two-players game like checkers. The two players alternately move a stone on a board. The moving player wins when his opponent has no more move to make. We can formalise that by an **AnsProlog^{not}** rule

- $wins(X) \leftarrow move_from_to(X, Y), \textbf{not } wins(Y)$

meaning that

- the situation X is won (for the moving player A), if he can lead over^a to a situation Y that can never be won for B.

^aWith the help of a regular move, given by the relation $move_from_to/2$.

Assume we also have the facts

$move_from_to(a, b)$, $move_from_to(b, a)$ and $move_from_to(b, c)$.

Our query to this program P_{game} is $?- wins(b)$.

Using SLDNF we get an infinite sequence of oscillating SLD-trees (none of which finitely fails).

N-Queens revisited

Why should we distinguish the queens (by using names for them)? It only leads to many more solutions. And the search space is larger.

1. **Declarations:** Predicates $row(\cdot)$, $col(\cdot)$ together with constants for them.

2. **Enumerations:**

Each square has a queen or not.

$$in(X, Y) \leftarrow row(X), col(Y), \text{not } not_in(X, Y)$$

$$not_in(X, Y) \leftarrow row(X), col(Y), \text{not } in(X, Y)$$

Each queen must be placed.

$$has_queen(X) \leftarrow row(X), col(Y), in(X, Y)$$

$$\leftarrow row(X), \text{not } has_queen(X)$$
3. **Elimination:** Elimination rules (constraints) as above
(Example 4.7 on page 154).

Assessment 2 (Lab: Colouring in $\text{AnsProlog}^{\text{not}, \perp}$)

Formulate your first labwork as a problem in $\text{AnsProlog}^{\text{not}, \perp}$,
i.e. without using **or** but using **not** instead.

Compare the runtimes.

Do the same with the two formalisations of N-Queens and
compute the number of solutions for $N=1, 2, \dots, 8$.

4.4 Answer Sets and their Semantics

What is the semantics of ASP? What exactly are answer sets?
Guess and check: We reduce a program with negation to one without. Then we check that the original guess is compatible with the program we constructed out of it.

We already know the semantics of **AnsProlog**^{or,⊥} (positive disjunctive) and **AnsProlog**[⊥] (definite) programs.

The idea of defining semantics for arbitrary **AnsProlog**^{or,not,⊥} and **AnsProlog**^{not,⊥} programs is to reduce them by getting rid of the negation **not**. We also assume that the program is given as a ground program (fully instantiated).

Definition 4.5 (ASP Semantics for $\text{AnsProlog}^{\text{or}, \text{not}, \perp}$ programs)

Let a ground $\text{AnsProlog}^{\text{or}, \text{not}, \perp}$ program P be given. Let also a Herbrand model \mathcal{M} wrt. the underlying language be given.

We reduce P wrt. \mathcal{M} as follows:

1. If a rule in P contains an atom of the form $\text{not } a$ and $\mathcal{M} \models a$ then remove the whole rule.
2. If a rule in P contains an atom of the form $\text{not } a$ and $\mathcal{M} \models \neg a$ then remove this occurrence of $\text{not } a$.
3. Applying the above two rules leads eventually to a program $P^{\mathcal{M}}$ without any not atoms.

$P^{\mathcal{M}}$ is in $\text{AnsProlog}^{\text{or}, \perp}$ (if P belongs to $\text{AnsProlog}^{\text{or}, \text{not}, \perp}$)
or it is in AnsProlog^{\perp} (if P belongs to $\text{AnsProlog}^{\text{not}, \perp}$).

\mathcal{M} is an answer set of P if \mathcal{M} is a minimal model of $P^{\mathcal{M}}$.
This includes the case when $P^{\mathcal{M}}$ is a definite program (in which case the least model is the only minimal model).

Homework 1

Go carefully through the examples considered so far and compute their answer sets.

Lemma 4.1 (Properties of Answer Sets)

1. Answer sets are minimal Herbrand models of P .
2. For each atom a that is contained in an answer set of P , there is a rule which contains a in its head and the body of this rule evaluates to true wrt. the given answer set.
3. There might be one, several or no answer set for a given program P .
4. For a definite logic program ([AnsProlog⁰](#)), there is exactly one answer set which coincides with the least Herbrand model.

Definition 4.6 (Dependency-Graph \mathcal{G}_P)

For a **AnsProlog**^{not, or, \perp} program P , the *dependency graph* \mathcal{G}_P is a finite directed graph whose vertices are the predicate symbols from P . There is a *positive* (respectively *negative*) edge from r to r' if and only if (1) there is a rule in P with r in its head and r' occurring positively (respectively negative) in its body, or (2) both r and r' occur in the same head.

We also say

- r *depends on* r' if there is a path in \mathcal{G}_P from r to r' (by definition, r depends on itself),
- r *depends positively* (resp. *negatively*) on r' if there is a path in \mathcal{G}_P from r to r' containing only *positive edges* (resp. *at least one negative edge*). (by definition r depends positively on itself),

- r *depends evenly* (resp. *oddly*) on r' if there is a path in \mathcal{G}_P from r to r' containing an *even* (resp. *odd*) number of negative edges (by definition r depends evenly on itself).

The following properties of a program P turn out to be very important:

stratified: no predicate depends negatively on itself^a,
call-consistent: no predicate depends oddly on itself^b,
allowedness: every variable occurring in a clause must occur in at least one positive atom of the body of that clause.

^aor: there are no cycles containing at least one *negative* edge.

^bor: there are no odd cycles.

Lemma 4.2 (Properties of classes of programs)

We consider programs without constraints.

1. Stratified **AnsProlog^{not}** programs always possess exactly one answer set.
2. Call consistent **AnsProlog^{not, or}** programs which do not have cycles with only positive edges always possess answer sets.
3. Answer sets of allowed **AnsProlog^{not, or}** programs with function symbols can be computed (because the Herbrand universe is finite).

Lets look at the formalisation in Example 4.7 on page 154. How can we simulate the two disjunctions that ensure that each queen is placed at some location by a **AnsProlog^{not}** program (without having to use the constants for rows and columns explicitly)?

One rule is

$$placed(I) \leftarrow queen(I), row(X), col(Y), at(I, X, Y)$$

and the other?

Examples: Graph Colouring, Hamiltonian Paths

```
col_of(V, C)      ←  vertex(V), col(C), not another_col(V, C)
another_col(V, C) ←  vertex(V), col(C), col(D),
                    D ≠ C, col_of(V, D)
                    ←  vertex(U), vertex(V), edge(U, V), col(C),
                    col_of(V, C), col_of(U, C)
```

The answer sets of this program correspond to colourings of the underlying graph.

Difference to other programming languages. Algorithm is built in!

Hamiltonian Path

1. As usual $edge(\cdot, \cdot)$, $vertex(\cdot)$.
2. $chosen(\cdot, \cdot)$: For each node, pick exactly one outgoing edge (for the hamiltonian path).
3. Ensure that there is only one incoming edge (for the hamiltonian path).
4. $reachable(\cdot)$: Define reachable and make sure all nodes are reachable.

$$\textit{other}(U, V) \leftarrow \textit{vertex}(U), \textit{vertex}(V), \textit{vertex}(W), V \neq W, \textit{chosen}(U, W)$$

$$\textit{chosen}(U, V) \leftarrow \textit{vertex}(U), \textit{vertex}(V), \textit{edge}(U, V), \textbf{not } \textit{other}(U, V)$$

$$\leftarrow \textit{chosen}(U, W), \textit{chosen}(V, W), U \neq V$$

$$\textit{reachable}(V) \leftarrow \textit{reachable}(U), \textit{chosen}(U, V)$$

$$\leftarrow \textit{vertex}(U), \textbf{not } \textit{reachable}(U)$$

$$\textit{reachable}(c_1) \leftarrow$$

4.5 Declarative Problem Solving Modules

Given a problem, how to encode it in ASP? How to ensure that solutions of the problem correspond to answer sets of the encoding?

Given a (mathematical) problem. How to encode it in the ASP paradigm?

Representing Constraints in AnsProlog^{not}

A constraint acts as a filter on answer sets. How can we simulate it with **not**?

We replace each constraint

$$\perp \leftarrow body$$

by the rule

$$inconsistent \leftarrow \mathbf{not} inconsistent, body,$$

where *inconsistent* is a new atom not occurring in the program.

Lemma 4.3 (Reducing $\text{AnsProlog}^{\text{or}, \text{not}, \perp}$ to $\text{AnsProlog}^{\text{or}, \text{not}}$)

Let P be a program not containing the atom *inconsistent*. Let P' be the program obtained by replacing all constraints using the transformation above. Then:

A is an answer set of P if and only if A is an answer set of P'

Enumerations

Given propositions p_1, \dots, p_n , how can we construct a program whose answer sets correspond to all possibilities of making some of the p_i true?

We can add predicates not_p_i and add the rules

$$\begin{aligned} p_i &\leftarrow \textbf{not } not_p_i \\ not_p_i &\leftarrow \textbf{not } p_i \end{aligned}$$

We can also use

$$p_i \textbf{ or } not_p_i \leftarrow$$

How can we enumerate terms satisfying a certain criterion?
 I.e.: Suppose we are given a predicate *possible* that is true for certain terms. We want to construct a program s.t. in each answer set of it there is at least one term satisfying the predicate.

We add the predicates *chosen* and the new atom *inconsistent* and add the rules

$$\begin{aligned} chosen(X) &\leftarrow possible(X), \textbf{not} not_chosen(X) \\ not_chosen(X) &\leftarrow possible(X), \textbf{not} chosen(X) \\ some &\leftarrow chosen(X) \\ inconsistent &\leftarrow \textbf{not} inconsistent, \textbf{not} some \end{aligned}$$

How can we ensure that exactly one term is chosen?

$$\begin{aligned} \textit{diff_chosen_than}(X) &\leftarrow \textit{chosen}(Y), X \neq Y \\ \textit{chosen}(X) &\leftarrow \textit{possible}(X), \textbf{not } \textit{diff_chosen_than}(X) \end{aligned}$$

Linear Orderings

We are given objects and a linear ordering between them.
How can we define the smallest, largest or next object using ASP?

$$not_smallest(X) \leftarrow object(X), object(Y), less_than(X, Y)$$

$$smallest(X) \leftarrow object(X), \textbf{not } not_smallest(X)$$

$$not_largest(X) \leftarrow object(X), object(Y), less_than(Y, X)$$

$$largest(X) \leftarrow object(X), \textbf{not } not_largest(X)$$

$$\begin{aligned} not_next(X, Y) &\leftarrow X = Y \\ not_next(X, Y) &\leftarrow less_than(Y, X) \\ not_next(X, Y) &\leftarrow object(X), object(Y), object(Z), \\ &\quad less_than(X, Z), less_than(Z, Y) \\ next(X, Y) &\leftarrow object(X), object(Y), \textcolor{blue}{not} not_next(X, Y) \end{aligned}$$

Suppose we are given a set of objects and want to establish a linear ordering between them. How do we do that?

1. We introduce an IDB relation $prec(\cdot, \cdot)$. To ensure that the ordering is linear and transitive, we use

$$prec(X, Y) \leftarrow \text{not } prec(Y, X), X \neq Y$$

$$prec(X, Z) \leftarrow prec(X, Y), prec(Y, Z)$$

2. We define the successor:

$$not_succ(X, Y) \leftarrow prec(Y, X)$$

$$not_succ(X, Y) \leftarrow prec(Y, Z), prec(Z, Y)$$

$$not_succ(X, X) \leftarrow$$

$$succ(X, Y) \leftarrow \text{not } not_succ(X, Y)$$

3. We have to define the first and last object:

$$\begin{aligned} not_first(X) &\leftarrow prec(Y, X) \\ first(X) &\leftarrow \textbf{not } not_first(X) \end{aligned}$$

$$\begin{aligned} not_last(X) &\leftarrow prec(X, Y) \\ last(X) &\leftarrow \textbf{not } not_last(X) \end{aligned}$$

4. We define reachability:

$$\begin{aligned} \textit{reachable}(X) &\leftarrow \textit{first}(X) \\ \textit{reachable}(X) &\leftarrow \textit{reachable}(Y), \textit{succ}(X, Y) \end{aligned}$$

5. Add a proposition *linear* which is true when there is indeed a linear ordering, i.e. the last element is reachable. If there is no possibility to establish a linear ordering, ensure that there are no answer sets.

$$\begin{aligned} \textit{linear} &\leftarrow \textit{reachable}(X), \textit{last}(X) \\ \textit{inconsistent} &\leftarrow \textbf{not } \textit{inconsistent}, \textbf{not } \textit{linear} \end{aligned}$$

4.6 DLV and smodels

Are there efficient ASP solvers out there? Yes: DLV, smodels.
DLV even handles disjunction.

You are given problems and have to represent them in the ASP framework. And then solve them using DLV or smodels.

Distinguishing features of DLV and smodels

Disjunction: DLV is designed for full **AnsProlog^{or,not,⊥}**, while smodels is designed for **AnsProlog^{not,⊥}**. smodels has only primitive functionality for **or**.

Grounding: Both systems compute **intelligent groundings**, trying to avoid unnecessary instances.

Relational DB: DLV can be seen as an extension to SQL3 and thus has functionality for answering SQL3 queries.

Queries: DLV allows **brave** and **cautious** reasoning: queries can be specified and tested for truth in **in at least one** or **in all** answer sets.

Allowedness: In smodels each variable in a rule must occur in a positive **domain predicate** on the right hand side of this rule. A domain predicate is one with the following property: each path in the dependency graph of the program starting with this predicate does not go through a negative cycle. This property is also called **strongly range restricted**. The idea is that domain predicates can be efficiently computed (no recursion through negation).

In DLV this is more relaxed: each variable must occur in a positive predicate on the right hand side.

Special Constraints: smodels allows weight and cardinality constraints, while DLV allows weak constraints (see next subsections).

Arithmetic: *smodels* allows rules of the form $p(T + 1) \leftarrow p(T)$.

In DLV this must be written as $p(T') \leftarrow p(T), T' = T + 1$.

Classical Negation: In our definition of an answer set

(Definition 4.5) and also in the definition of

AnsProlog ^{or ,not,⊥}, we did not allow atoms that are classically negated. In fact, in several formalisations we used predicates of the form *not_predicate* which, intuitively represented the negation of the predicate *predicate*. We did this mainly to avoid any confusion with classical negation.

But our definition can easily be extended to allow arbitrary **literals** instead of just positive atoms: we allow atoms $p(\dots)$ as well as their classical negations $\neg p(\dots)$ on **both sides of a rule**.

An answer set then consists of arbitrary **literals**: atoms $p(\dots)$ or their classical negations $\neg p(\dots)$. If an answer set contains both an atom and its negation, then it coincides with the set of all atoms and their negations: it is the inconsistent set *Lit* of all literals.

smodels allows *Lit* as an answer set, while DLV does not.

Thus DLV can not distinguish between programs without any answer sets and programs which only have *Lit* as their sole answer set.

Representing classical and exclusive or

The disjunction **or** is not classical disjunction, as we are looking at minimal models. But it is also not exclusive disjunction \oplus . In a sense, **or** is **as exclusive as is consistently possible**.

Homework 2 (Different disjunctions)

Give an example of a **AnsProlog^{not, or}** program where the answer sets do not treat **or** as classical disjunction.

Give an example of a **AnsProlog^{not, or}** program where the answer sets do not treat **or** as exclusive disjunction.

Can we simulate classical disjunction within **AnsProlog^{not}**?

Yes, namely by representing the rule

$$a_1 \vee \dots \vee a_n \leftarrow b_1, \dots, b_m, \textbf{not } c_1, \dots, \textbf{not } c_l$$

as follows.

1. We add the rule $f' \leftarrow f, \textbf{not } f'$.
2. We add the rule $f \leftarrow a'_1, \dots, a'_n, b_1, \dots, b_m, \textbf{not } c_1, \dots, \textbf{not } c_l$.

3. For $i = 1, \dots, n$ we add the rules

$$a_i \leftarrow \text{not } a'_i, b_1, \dots, b_m, \text{not } c_1, \dots, \text{not } c_l$$

$$a'_i \leftarrow \text{not } a_i, b_1, \dots, b_m, \text{not } c_1, \dots, \text{not } c_l$$

Can we simulate exclusive disjunction within **AnsProlog^{not}**?

Yes, namely by representing the rule

$$a_1 \oplus \dots \oplus a_n \leftarrow b_1, \dots, b_m, \text{not } c_1, \dots, \text{not } c_l$$

as follows.

1. We add the rule $f' \leftarrow f, \text{not } f'$.
2. We add the rule $f \leftarrow a'_1, \dots, a'_n, b_1, \dots, b_m, \text{not } c_1, \dots, \text{not } c_l$.

3. For $i = 1, \dots, n$ we add the rules

$$a_i \leftarrow \text{not } a'_i, b_1, \dots, b_m, \text{not } c_1, \dots, \text{not } c_l$$

$$a'_i \leftarrow \text{not } a_i, b_1, \dots, b_m, \text{not } c_1, \dots, \text{not } c_l$$

4. For each $1 \leq i \prec j \leq n$ we add

$$f \leftarrow b_1, \dots, b_m, \text{not } c_1, \dots, \text{not } c_l, a_i, a_j$$

Special Constraints in smodels

smodels allows **cardinality constraints** to ensure that an answer set contains at least and at most a certain number of prespecified atoms.

1 {a,b,**not** c} 2

This means that we are looking for answer sets which contain at least one but at most two of the atoms a, b , **not** c .

smodels also allows **optimisation constraints** to assign to each answer set a number according to which atoms and **not** atoms are true (given their weights). We can then pick answer sets which minimise (maximise) the sum of all weights.

3 [a=2, b= 3, **not** c=1, **not** d=4] 7

The above constraint formalises that we are looking for answer sets having at least weight 3 but not more than 7.

Special Constraints in DLV

DLV allows **weak** constraints.

Definition 4.7 (Weak Constraint)

A weak constraint has the form

$$\perp \quad : \sim \quad (p_1, \dots, p_n, \text{not } q_1, \dots, \text{not } q_m) [weight, level],$$

where p_i and q_j are literals and $weight$ and $level$ are integers or integer variables that can appear in the p_i or q_j .

Answer sets for a program with weak constraints are the answer sets for the program without the weak constraints, but ordered according to the weight and priority level they violate.

To be more specific, first the weights on the first level are minimised. The best answer sets are the ones where the weights on the first level are minimal. Then the answer sets with the second minimal weight etc. To further distinguish, the second level is taken into account, etc.

If no weight or level information is specified, they are set to 1 by default.

Example 4.10 (Weak Constraints)

Consider the following program:

$$\begin{array}{ll} a \textbf{or} b & \leftarrow \\ \neg a \textbf{or} c & \leftarrow \\ \bot & : \sim a \\ \bot & : \sim b, c \end{array}$$

What are its answer sets and which is the best one?

A nice application of weak constraints is the **Travelling Sales Person**. Find a route among a given number of towns, visiting each only once and minimising the overall distance.

Assessment 3 (TSP)

Find the shortest route to travel to all European capital cities.

You can take the mileage by direct line between the capitals.

4.7 HTN Planning

- HTN planner produces a sequence of actions that perform a **task**.
- Description of a planning domain includes a set of **operators**, and also a set of **methods**, to decompose a **task** into **subtasks**.
- Planning proceeds by decomposing tasks recursively into smaller and smaller subtasks, until they can be satisfied.
- SHOP's knowledge base contains **operators** and **methods**.



Figure 4.1: Travel planning example.

- The planner must **resolve interactions** among the subtasks.
- In Figure 4.1, it was always obvious which method to use. Often, more than one method may be applicable to a task. If a dead end is reached, **SHOP will backtrack** and try another method instead.

- A **term** is either a constant or a variable symbol.
- A **task** is of the form $(\text{name } t_1 t_2 \dots t_n)$, where name (the task's name) is a task symbol, and t_1, t_2, \dots, t_n (the task's arguments) are terms.
- A task can be either **primitive** (if it is to be accomplished using an operator) or **compound** (if it is to be decomposed into other tasks).
- A **task list** is a list of tasks, like the following:

```
((!get-taxi ?x) (!ride-taxi ?x ?y)
(!pay-driver ?x ?y)))
```

An operator specifies how to accomplish a primitive task by modifying the current state of the world by removing every atom in its delete list and by adding every atom in its add list.

Definition 4.8 (Operator**: $(\text{Op } h \ \chi_{del} \ \chi_{add})$)**

An operator is an expression of the form $(\text{Op } h \ \chi_{del} \ \chi_{add})$, where h (the *head*) is a primitive task and χ_{add} and χ_{del} are lists of atoms (called the *add-* and *delete-lists*). The set of variables in the atoms in χ_{add} and χ_{del} must be a subset of the set of variables in h .^a

^aUnlike the operators used in action-based planning, ours have no preconditions (these will occur in the methods that invoke the operators).

Methods can be seen as recursive task definitions.

Definition 4.9 (Method: $(\text{Meth } h \ \chi \ t) \)$

A *method* is an expression of the form $(\text{Meth } h \ \chi \ t)$ where h (the method's *head*) is a task, χ (the method's *preconditions*) is a conjunction of literals and t is a totally ordered list of subtasks, called the *task list*.

`get-taxi` operator from Figure 4.1:

```
(:Op (!get-taxi ?x)
  ((service-available-to ?x))
  ((taxi-coming-to ?x)))
```

`travel-by-taxi` method from Figure 4.1:

```
(:Meth (travel ?x ?y)
  ((smaller-distance ?x ?y))
  ((!get-taxi ?x) (!ride-taxi ?x ?y) (!pay-driver ?x ?y)))
```

Definition 4.10 (Axioms (AX**), State (**S**))**

An **axiom** is an expression of the form

$$a \leftarrow l_1, \dots, l_n,$$

where a is an atom and the l_i are literals. Axioms need not be ground. We assume that the set of axioms does not contain cycles through negation. A state **S** is a set of ground atoms.

Definition 4.11 (Literal **caused by (**S**,**AX**))**

A literal l is caused by (**S**, **AX**) if l is true in all answer sets of $\mathbf{S} \cup \mathbf{AX}$.

Due to the assumption on **AX**, there is exactly one answer set. Thus any state described by the stable model of $\mathbf{S} \cup \mathbf{AX}$ is complete.

Definition 4.12 (Simple Reduction of Primitive Tasks)

Let t be a ground primitive task and let $\text{Op} = (\text{Op } h \ \chi_{del} \ \chi_{add})$ be an operator. Suppose that u is a unifier for h and t . Then the ground operator instance Op^u is *applicable* to t , in which case we define the **simple reduction** of t by Op to be Op^u .

Definition 4.13 (Plans, **result(S,P))**

A *plan* is a list of heads of ground operator instances.^a A plan P is called a **simple plan** if it consists of the head of just one ground operator instance.

^aIn Definition 4.15, every planning operator must have a unique name. This guarantees that every plan specifies an unambiguous sequence of operator instances.

For a simple plan $P = (h)$ we define **result**(**S**, P) to be the set

$$\mathbf{S} \setminus \chi_{del} \cup \chi_{add}$$

obtained by deleting from **S** all atoms in χ_{del} and by adding all ground instances of atoms in χ_{add} .

If $P = (h_1, h_2, \dots, h_n)$ is a plan and **S** is a state, then the **result** of applying P to **S** is the state **result**(**S**, P) = **result**(**result**(...(**result**(**S**, h_1), h_2), ...), h_n).

Definition 4.14 (Simple Reduction of Compound Tasks)

Let t be a compound task, S be the current state,

$Meth = (Meth\ h\ \chi\ t)$ be a method, and AX be an axiom set.

Suppose that u is a unifier for h and t , and that v is a unifier such that all literals in $(\chi^u)^v$ are caused wrt. S and AX (see Definition 4.11).

Then the method instance $(Meth^u)^v$ is **applicable** to t in S , and the result of applying it to t is the task list $r = (t^u)^v$. The task list r is the **simple reduction** of t by $Meth$ in S .

Definition 4.15 (Domains and problems)

A **domain representation** D is a triple consisting of (1) a set of axioms, (2) a set of operators such that no two operators have the same name, and (3) a set of methods.

A **planning problem** is a triple (S, t, D) , where S is a state, $t = (t_1, t_2, \dots, t_k)$ is a task list, and D is a domain representation.

Let $(\mathbf{S}, t, \mathbf{D})$ be a planning problem, $P = (p_1, p_2, \dots, p_n)$ a plan. Then we say that P solves $(\mathbf{S}, t, \mathbf{D})$ if any of the following is true:

1. Case 1: t and P are both empty, (i.e., $k = 0$ and $n = 0$);
2. Case 2: t_1 is a primitive task, (p_1) is the simple reduction of t_1 , and $(p_2 \dots p_n)$ solves $(\mathbf{result}(\mathbf{S}, p_1), (t_2, \dots, t_k), \mathbf{D})$;
3. Case 3: t_1 is a composite task, and there is a simple reduction $(r_1 \dots r_j)$ of t_1 in \mathbf{S} s.t. P solves $(\mathbf{S}, (r_1, \dots, r_j, t_2, \dots, t_k), \mathbf{D})$.

$(\mathbf{S}, t, \mathbf{D})$ is *solvable* if there is a plan that solves it.

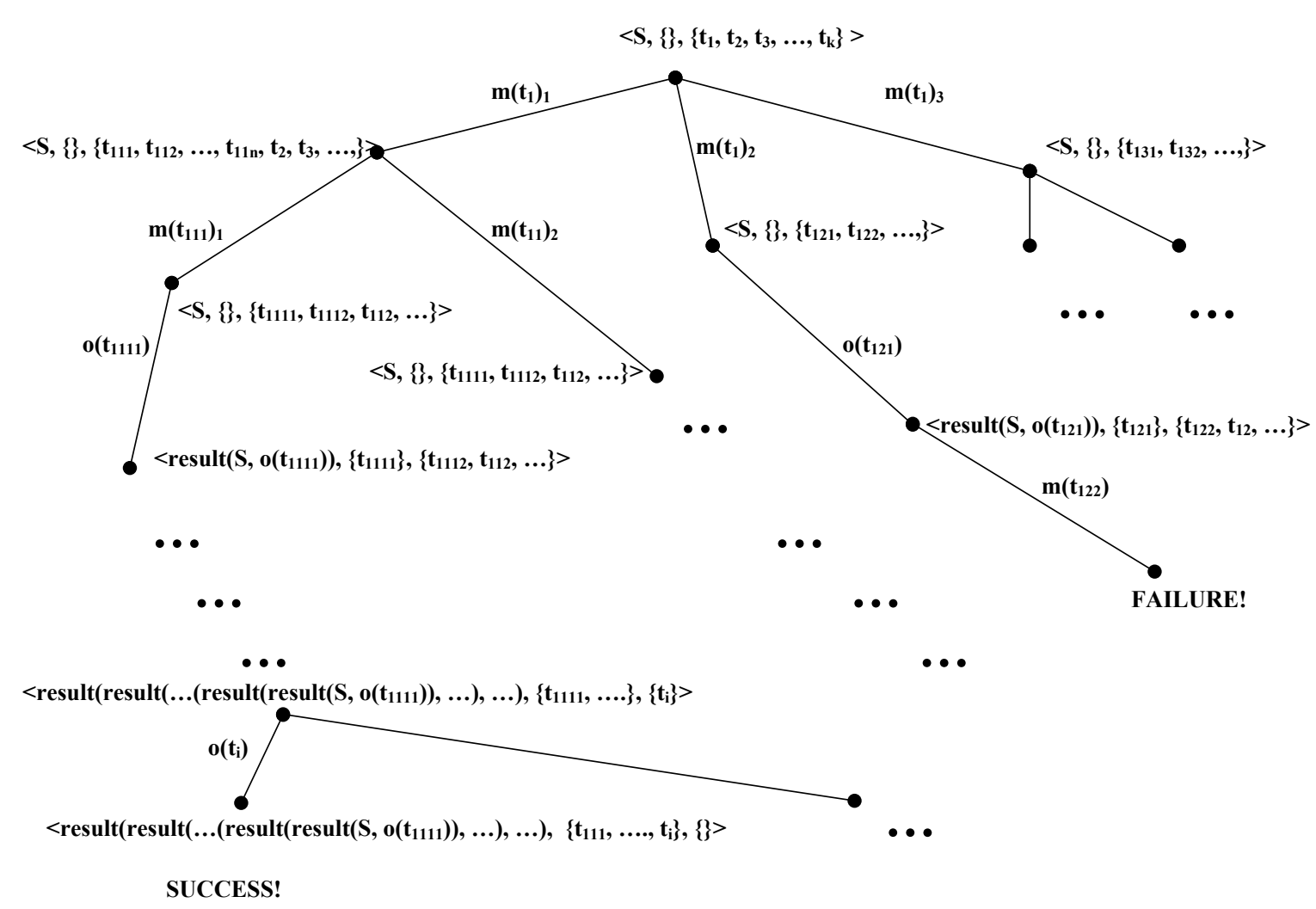


Figure 4.2: Search Tree for $(\mathbf{S}, \mathbf{t}, \mathbf{D})$. Edge labelings $m(t)_i$ ($o(t)$) represent a method (an operator) application to a task t .

Definition 4.16 (Solution set of a planning problem: $\text{Sol}(\mathbf{S}, \mathbf{t}, \mathbf{D})$)

The set of all plans, which we denote by $\text{Sol}(\mathbf{S}, \mathbf{t}, \mathbf{D})$ is a *multi set*: it contains exactly the ordered lists t_{caused} in the leaf nodes that are reached by the successful paths of the search tree. Thus, $\text{Sol}(\mathbf{S}, \mathbf{t}, \mathbf{D})$ may contain more than one copy of the same plan.

Theorem 4.3

Let a planning problem $(\mathbf{S}, \mathbf{t}, \mathbf{D})$ be given, where \mathbf{S} is the initial state, \mathbf{t} is the list of tasks to be achieved and \mathbf{D} is the domain description.

There is a solution to $(\mathbf{S}, \mathbf{t}, \mathbf{D})$ if and only if the list \mathbf{t} is causable w.r.t. (\mathbf{S}, \mathbf{D}) .

4.8 Realising HTN Planning in ASP

In this section, we present our translation method for encoding planning problems as logic programs with ASP semantics.

1. Planning starts at time point **0** (Def. 4.19 and 4.22).
2. Planning proceeds by selecting a task as the **current task** (the one to be decomposed next (see Def. 4.22 and the rules about *currentTask* and *causable* in items 3. and 4. of Def. 4.29). The current task to be decomposed may be either primitive or compound.

3. The **time variable T** is incremented only when the current task is a primitive task and there is an operator for it (a simple reduction) in the domain description provided as a part of the planning problem (see Def. 4.21). Thus there may be several tasks that are selected and decomposed at a particular time point **T**. Among them, there is only one primitive task at any particular point in time. Consider Figure 4.2. Task t_1 is a compound task and so are t_{111} and t_{1111} (obtained by respective methods). So $currentTask(t_1, \mathbf{0})$, $currentTask(t_{111}, \mathbf{0})$, $currentTask(t_{1111}, \mathbf{0})$ are all true (resp. hold in a stable model). Only after the primitive task t_{1111} has been established by an operator is the time incremented by 1.
4. As a result of this formulation, the task-depth in the search tree corresponds to the value of the time variable **T**.

Definition 4.17 ($\text{Trans}((\mathbf{S}, t, \mathbf{D}))$: Translating the Planning Problem)

The logic program $\text{Trans}((\mathbf{S}, t, \mathbf{D}))$ that solves the planning problem $(\mathbf{S}, t, \mathbf{D})$ is defined as

$$\begin{aligned} \text{Trans}((\mathbf{S}, t, \mathbf{D})) = & \text{Trans}(I) \cup \text{Trans}(\mathbf{S}) \cup \text{Trans}(t) \cup \text{Trans}(\mathbf{AX}) \\ & \cup \text{Trans}(\mathbf{OP}) \cup \text{Trans}(F) \cup \text{Trans}(\mathbf{METH}), \end{aligned}$$

where

- $\text{Trans}(I)$ is the logic program segment that encodes the planning-independent rules,
- $\text{Trans}(\mathbf{S})$ is the logic program segment that encodes the initial state, \mathbf{S} ,
- $\text{Trans}(t)$ is the logic program segment that encodes the goal task list, t ,

- $\mathcal{T}rans(\mathbf{AX})$ is the logic program segment that encodes the axioms given in the domain description, \mathbf{D} ,
- $\mathcal{T}rans(\mathbf{OP})$ is the logic program segment that encodes the operator descriptions given in \mathbf{D} , and
- $\mathcal{T}rans(F)$ is the logic program segment that encodes the state-transition characteristics of SHOP, and
- $\mathcal{T}rans(\mathbf{METH})$ is the logic program segment that encodes the method descriptions given in \mathbf{D} .

Definition 4.18 ($\mathcal{Trans}(I)$: Planning-Independent Rules)

Given a planning problem $(\mathcal{S}, t, \mathcal{D})$, we define $\mathcal{Trans}(I)$ as the logic program that consists of the following set of rules:

- For each object o that may be used at some point in the solution of $(\mathcal{S}, t, \mathcal{D})$:

$$[type](o) : -$$

- For each atom A that may appear to be true at some state generated during the planning process:

$$atom(A) : -$$

SHOP's initial state is a set of ground atoms. In this respect, given a planning problem $(\mathbf{S}, t, \mathbf{D})$, the logic program encoding for the initial state \mathbf{S} is defined as follows:

Definition 4.19 ($\mathcal{T}rans(\mathbf{S})$: Translation for Initial State)

Given a planning problem $(\mathbf{S}, t, \mathbf{D})$, for each ground atom $a \in \mathbf{S}$, the logic program $\mathcal{T}rans(\mathbf{S})$ contains the rule

$$in_state(a, \mathbf{0}) : -$$

where $\mathbf{0}$ indicates the initial *time*.

Definition 4.20 (Task to be Decomposed)

Given a planning problem $(\mathbf{S}, t, \mathbf{D})$, we define a special predicate $currentTask_n$ for each possible task (e.g. primitive or compound) such that if the current task to be decomposed is $h \equiv (name_h arg_1 arg_2 \dots arg_N)$ then $currentTask_n(name_h, arg_1, arg_2, \dots, arg_N, \mathbf{T})$ denotes this fact and n is a natural number which equals $N + 2$ (n is the number of arguments of this predicate).

As an example, if the current task to be accomplished is travelling from UMD to MIT denoted as $(travel\ umd\ mit)$, then we use the predicate $currentTask_4(travel, umd, mit, \mathbf{T})$ to denote this fact. For the sake of clarity, we will use the shorthand notations $currentTask(h, \mathbf{T})$.

Definition 4.21 (CAUSABLE)

Given a task t , we define $CAUSABLE(t, T_{selected}, T_{accomplished})$:

$false$

$false$

$currentTask(t, T_{selected})^a$

$causable(t, T_{selected}, T_{accomplished})$

if t is a primitive task and
there is no operator for it in \mathbf{D} ,

if t is a compound task and
there is no method for it in \mathbf{D} ,

if t_k is a primitive task and
there is an operator for it in \mathbf{D} ,

if t_k is a compound task and
there is a method for it in \mathbf{D} .

^awhere $T_{accomplished} = T_{selected} + 1$

4.8 Realising HTN Planning in ASP

231

Definition 4.22 ($\mathcal{T}rans(t)$: Translation for Goal Tasks)

Given a planning problem $(\mathcal{S}, t, \mathcal{D})$, let $t = h_1, h_2, \dots, h_n$ be the ordered sequence of goal tasks. Then, $\mathcal{T}rans(t)$ is the logic program that contains one rule for each goal task h_i , where $i = 1, 2, \dots, n$, as follows:

1. **Case 1:** $i = 1$: $currentTask(h_1, \mathbf{0}) : -$.

2. **Case 2:** Otherwise,

$$currentTask(h_i, \mathbf{T}_i) : - \quad CAUSABLE(h_{i-1}, \mathbf{T}_{i-1}, \mathbf{T}_i), \\ \mathbf{T}_i \geq \mathbf{T}_{i-1}.$$

Definition 4.22 enforces the fact that a goal task h_i is designated as the current task to be accomplished if the previous goal task h_{i-1} in t is causable (Theorem 4.3).

Definition 4.23 ($\mathcal{T}rans(\cdot)$: Successful Termination)

Given a planning problem $(\mathcal{S}, t, \mathcal{D})$, the logic program segment $\mathcal{T}rans(\cdot)$ that encodes the successful termination of the planning process (i.e., the fact that a solution to the given planning problem is found) is defined as follows:

$$plan_found : - CAUSABLE(h_n, \mathbf{T}_n, \mathbf{T}_{n+1}).$$

$$plan_found : - \text{not } plan_found.$$

where \mathbf{T}_n denotes the time when the particular method for the last goal task h_n is decomposed, \mathbf{T}_{n+1} is the time at which h_n is causable (accomplished), and k is the number of ways that the goal task h_n can be accomplished.

Definition 4.24 (Translation for Literals)

Given a literal, l , we define $C(l, \mathbf{T})$, the *translation* of l at time \mathbf{T} , as

$$C(l, \mathbf{T}) := \begin{cases} in_state(a, \mathbf{T}) & \text{if } l = a, \\ \text{not } in_state(a, \mathbf{T}) & \text{if } l = \neg a. \end{cases}$$

where a is an atom.

Definition 4.25 ($\mathcal{T}rans(\mathbf{AX})$: Translation for Axioms)

Given a planning problem $(\mathbf{S}, t, \mathbf{D})$, for all $a \leftarrow l_1, \dots, l_n \in \mathbf{AX}$, $\mathcal{T}rans(\mathbf{AX})$ is the logic program segment that contains the single rule

$$in_state(a, \mathbf{T}) : - C(l_1, \mathbf{T}), C(l_2, \mathbf{T}), \dots, C(l_n, \mathbf{T}),$$

where $C(l_i, \mathbf{T})$ is the translation of a literal as defined in Definition 4.24 above.

Definition 4.26 ($\mathcal{T}rans(\mathcal{OP})$: Translation for Operators)

Given a planning problem $(\mathcal{S}, t, \mathcal{D})$, for all $Op \in \mathcal{OP}$, $\mathcal{T}rans(Op)$ is the logic program that contains the following rules:

for all $a \in Del(Op)$:

$$out_state(a, \mathbf{T+1}) : - currentTask(h, \mathbf{T}).$$

and for all $a \in Add(Op)$:

$$in_state(a, \mathbf{T+1}) : - currentTask(h, \mathbf{T}).$$

where h is a primitive task - i.e., the ground head of the operator that is used in the simple reduction of h .

Definition 4.27 ($\mathcal{T}rans(F)$): Keeping Track of the State S)

The logic program segment $\mathcal{T}rans(F)$ that encodes the frame axiom is defined as follows:

$$in_state(A, \mathbf{T+1}) \quad : - \quad atom(A), in_state(A, \mathbf{T}), \\ \mathbf{not} out_state(A, \mathbf{T+1}).$$

where \mathbf{T} is a variable of the sort time. Note that the state of the world in SHOP consists of only positive ground atoms.

Definition 4.28 (Methods for a Task)

Given a planning problem $(\mathbf{S}, t, \mathbf{D})$ and a composite task $h \equiv (name_h \ arg_1 \ arg_2 \ \dots \ arg_N)$, we define a special predicate $method_n(name_h, arg_1, arg_2, \dots, arg_N, Pre(h), T)$ for each method $m \in \mathbf{D}$ whose head unifies with h , where $Pre(h)$ is the label^a for the precondition list of the method m . The symbol n in the predicate name denotes the number arguments of the predicate, i.e. $n = N + 3$.

^aThe label $Pre(h)$ is an implementation issue and is required for uniquely identify different methods that may be applicable to the same compound task.

Definition 4.29 ($\mathcal{T}rans(\mathcal{METH})$: Translation for Methods)

Given a planning problem $(\mathcal{S}, t, \mathcal{D})$, let h be a compound task that needs to be accomplished in the solution of the given planning problem. Suppose the domain description \mathcal{D} contains N methods whose heads unify with h ; namely, m_1, m_2, \dots, m_N . Let $Pre(h)_i$ be the label for the precondition list of the method m_i . Then, the logic program segment that encodes these methods is defined as follows:

1. *The nondeterministic choice of which method to apply to the task h :*

$$\begin{array}{lll}
 method_1(h, \text{Pre}(h)_1, \mathbf{T}) & \leftarrow & currentTask(h, \mathbf{T}), \\
 & & \mathbf{not} \, method_2(h, \text{Pre}(h)_2, \mathbf{T}), \dots, \\
 & & \mathbf{not} \, method_N(h, \text{Pre}(h)_N, \mathbf{T}) \\
 \vdots & & \vdots \\
 method_N(h, \text{Pre}(h)_N, \mathbf{T}) & \leftarrow & currentTask(h, \mathbf{T}), \\
 & & \mathbf{not} \, method_1(h, \text{Pre}(h)_1, \mathbf{T}), \dots, \\
 & & \mathbf{not} \, method_{N-1}(h, \text{Pre}(h)_{N-1}, \mathbf{T})
 \end{array}$$

2. The precondition list χ_i of each method m_i :

For each precondition $p \in \chi_i$, we have one of the following two cases:

(a) p is a positive literal and it contains free variables:

The free variables in a precondition literal are the variable symbols that do not appear in the head of the method m_i . We denote p as $p = p(Y_1, Y_2, \dots, Y_f)$, Y_1, Y_2, \dots, Y_f are the free variables in p .

Let R_j denote the *range* of the free variable Y_j - i.e. the set of all possible values for the variable Y_j -, and for each such variable Y_j , let $Y_{j,k}$ be a new variable symbol such that $k = 1, \dots, R_j$. Then, **Trans**(**METH**) contains the following rule to encode the precondition $p \in \chi_i$:

$$\begin{aligned}
& checked_state(p(Y_{1,1}, Y_{2,1}, \dots, Y_{f,1}), \mathbf{T}) \quad : - \\
& \quad method_i(h, \text{Pre}(h)_i, \mathbf{T}), in_state(p(Y_{1,1}, Y_{2,1}, \dots, Y_{f,1}), \mathbf{T}), \\
& \quad \mathbf{not} \, checked_state(p(Y_{1,1}, Y_{2,1}, \dots, Y_{f,1}), \mathbf{T}), \\
& \quad \mathbf{not} \, checked_state(p(Y_{1,1}, Y_{2,1}, \dots, Y_{f,2}), \mathbf{T}), \\
& \quad \vdots \\
& \quad \mathbf{not} \, checked_state(p(Y_{1,1}, Y_{2,1}, \dots, Y_{f,R_f}), \mathbf{T}), \\
& \quad \vdots \\
& \quad \mathbf{not} \, checked_state(p(Y_{1,1}, Y_{2,R_2}, \dots, Y_{f,R_f}), \mathbf{T}), \\
& \quad \mathbf{not} \, checked_state(p(Y_{1,2}, Y_{2,1}, \dots, Y_{f,1}), \mathbf{T}), \\
& \quad \vdots \\
& \quad \mathbf{not} \, checked_state(p(Y_{1,R_1}, Y_{2,R_2}, \dots, Y_{f,R_f}), \mathbf{T}). \\
& \bigwedge_{j=1}^f Y_{j,1}! = Y_{j,2}! = \dots! = Y_{j,R_j}.
\end{aligned}$$

(b) *Otherwise:*

The logic program segment $\mathfrak{T}rans(\mathbf{METH})$ contains the following rule to encode the precondition $p \in \chi_i$:

$$checked_state(p_l, \mathbf{T}) : - C(p_l, \mathbf{T}), method_i(h, Pre(h)_i, \mathbf{T}).$$

where $C(p_l, \mathbf{T})$ is as defined in Definition 4.24.

3. *The decomposition list $\{t_1, t_2, \dots, t_n\}$ for m_i :*

Let $p_1, p_2, \dots, |\chi_i|$ be the list of preconditions of the method m_i . Then, the logic program segment $\mathfrak{T}rans(\mathbf{METH})$ contains the following rules to encode the decomposition list of m_i (note that the time variable \mathbf{T}_1 in the following rule definitions in this item denote the same value as the time variable \mathbf{T} in the

rule definitions presented in other items does):

$$\begin{aligned}
 currentTask(t_1, \mathbf{T}_1) & : - \quad method_i(h, Pre(h)_i, \mathbf{T}_1), \\
 & \quad \bigwedge_{k=1}^{|\chi_i|} checked_state(p_k, \mathbf{T}_1). \\
 currentTask(t_2, \mathbf{T}_2) & : - \quad method_i(h, Pre(h)_i, \mathbf{T}_1), \\
 & \quad \bigwedge_{k=1}^{|\chi_i|} checked_state(p_k, \mathbf{T}_1), \\
 & \quad CAUSABLE(t_1, \mathbf{T}_1, \mathbf{T}_2), \\
 & \quad \mathbf{T}_2 > \mathbf{T}_1. \\
 & \quad \vdots \\
 & \quad \vdots \\
 & \quad \vdots \\
 currentTask(t_n, \mathbf{T}_n) & : - \quad method_i(h, Pre(h)_i, \mathbf{T}_1) \\
 & \quad \bigwedge_{k=1}^{|\chi_i|} checked_state(p_k, \mathbf{T}_1), \\
 & \quad CAUSABLE(t_{n-1}, \mathbf{T}_{n-1}, \mathbf{T}_n), \\
 & \quad \mathbf{T}_n > \mathbf{T}_{n-1}.
 \end{aligned}$$

4. *The accomplishment (i.e., causation) of h by the method m_i :*

$$\begin{aligned}
 causable(h, \mathbf{T}_1, \mathbf{T}_{n+1}) \quad : - \quad & method_i(h, \mathbf{T}_1), \\
 & \bigwedge_{k=1}^{|\chi_i|} checked_state(p_k, \mathbf{T}_1), \\
 & CAUSABLE(t_n, \mathbf{T}_n, \mathbf{T}_{n+1}), \\
 & \mathbf{T}_{n+1} > \mathbf{T}_n.
 \end{aligned}$$

1. Rules given in the first item of Def. 4.29 could also be encoded into one disjunctive rule.
2. Translation in the third item of Def. 4.29 encodes the fact that the decomposition of each subtask t_k can only be done if the previous subtask t_{k-1} has been already accomplished – i.e. t_{k-1} has been already *CAUSABLE* (only exception is the first task).

This property is encoded by using the $CAUSABLE(t_k, T_k, T_{k+1})$ construct for each subtask t_k (see Definition 4.21): T_k denotes the time when the previous subtask is decomposed and T_{k+1} denotes the time when the current task is accomplished – i.e. *causable*.

Detailed formalisations can be obtained from

http://www.cs.umd.edu/users/ukuter/ASP_Planning.

Theorem 4.4 ($\mathcal{T}rans(\cdot)$ and HTN planning using OTD)

Given a planning problem (S, t, D) , where S is the initial state, t is the list of tasks to be achieved and D is the domain description, let $\mathcal{T}rans((S, t, D))$ be the corresponding logic program with negation. We assume that the set of axioms in D does not contain any cycles through negation. Furthermore, let $Sol(S, t, D)$ be the set of solutions as defined in Definition 4.15. Then,

1. $Sol(S, t, D) = \emptyset$ if and only if $\mathcal{T}rans((S, t, D))$ has no answer sets.

2. If $\text{Sol}(\mathbf{S}, \mathbf{t}, \mathbf{D}) \neq \emptyset$, then the following holds:
- (a) For every plan $P \in \text{Sol}(\mathbf{S}, \mathbf{t}, \mathbf{D}) \neq \emptyset$, there is an answer set of $\text{Trans}((\mathbf{S}, \mathbf{t}, \mathbf{D}))$ and a sequence of primitive tasks t_0, t_1, \dots, t_n , such that the predicates $\text{currentTask}(t_i, i)$ that are true in this answer set and the t_i correspond exactly to the steps p_i in P .
 - (b) For every answer set of $\text{Trans}((\mathbf{S}, \mathbf{t}, \mathbf{D}))$ there is a sequence of primitive tasks t_0, \dots, t_n , such that the predicates $\text{currentTask}(t_i, i)$ are true in this answer set and this sequence constitutes a plan $[t_0, \dots, t_n] \in \text{Sol}(\mathbf{S}, \mathbf{t}, \mathbf{D})$.

Corollary 1 (Soundness and Completeness of $\mathcal{T}rans(\cdot)$)

The answer sets of $\mathcal{T}rans((\mathbf{S}, t, \mathbf{D}))$ correspond exactly to the plans in $\mathbf{Sol}(\mathbf{S}, t, \mathbf{D})$. There is a bijection between these two sets and each plan in $\mathbf{Sol}(\mathbf{S}, t, \mathbf{D})$ can be reconstructed from its corresponding answer set in $\mathcal{T}rans((\mathbf{S}, t, \mathbf{D}))$ and vice versa.

Corollary 2 (Soundness and Completeness of $\mathcal{T}rans(\cdot)$ wrt SHOP)

If the axioms \mathbf{AX} in \mathbf{D} do not contain any (positive or negative) cycles, then the answer sets of $\mathcal{T}rans((\mathbf{S}, t, \mathbf{D}))$ correspond exactly to the plans computed by SHOP.

4.9 Benchmarks and Comparisons

We used three different planning domains (finding **all** solutions).

The Travelling Domain: The scenario for the domain as described in (Nau, Cao, Lotem, and Muñoz-Avila 1999) is that we want to travel from one location to another in a city.

The Miconic-10-simtest Domain: It is contained in a series of benchmarks `<http://www.informatik.uni-freiburg.de/~koehler/elev/elev.html>` and it was recently used not only to measure the performance of various planners but also for other translation methods from planning problems into ASP (see `http://www.fcs.nmsu.edu/~tson/asp_planner/`).

The Zeno-Travel Domain: The Zeno-Travel problem was one of the domains that were introduced as recent benchmarks in International Planning Competition (IPC-2002) (IPC-2002 was organised within AIPS-2002: see <http://www.dur.ac.uk/d.p.long/competition.html>). This version is essentially the domain used to illustrate the Zeno planning system developed by Penberthy and Weld [<http://www.cs.washington.edu/ai/zeno.html>]. In our experiments we used the STRIPS version of the domain.

- *Smodels* system v2.27 and *lparse* v1.0.11
- *DLV* (<http://www.dbai.tuwien.ac.at/proj/dlv/>)

- HP Notebook PC with an AMD 900Mhz Processor and 256MB RAM running Linux RedHat v7.2
- We also redid the experiments of (Son, Baral, and McIlraith. 2001) on our machine so that fair comparisons could be done.
- In our experiments on the Travelling Domain using our method together with *DLV*, we got a speed-up of two orders of magnitude compared to *Smodels*.

Comparing our method with (Son, Baral, and McIlraith. 2001)

Problem	$\mathcal{T}rans(\cdot)$	(Son, Baral, and McIlraith. 2001)
S1-0	0.050	0.520
S2-0	0.330	12.410
S3-0	1.390	121.810
S4-0	4.540	883.700
S5-0s1	19.530	no solution
S5-0s2	20.630	no solution
S6-0	23.150	no solution

Table 4.1: Comparison between our *Smodels* encoding of Miconic-10-simtest and the encoding described in (Son, Baral, and McIlraith. 2001).

How to write HTN formulations of a planning domain?

- Formulations may be based on different ways of conceptualising the problem. They would produce very different search spaces.
- The two different formulations may use basically the same tasks, and use them to mean basically the same thing. However, one formulation may take less time because it has lower overhead, or because it does a better job of deciding which tasks should actually be generated and explored.

Thus, we were careful to write our HTN formulation of Miconic-10-simtest so that we used basically the same conceptual representation that they did.

The problems that we used in these experiments are from http://www.cs.nmsu.edu/~tson/asp_planner>. Table 4.1 shows both our results and the results from (Son, Baral, and McIlraith. 2001), which were also obtained on the *Smodels* system.

Comparing *Smodels* and *DLV* using planning benchmarks

Problem	<i>Smodels</i>	<i>DLV</i>	<i>DLV</i> grounding+ <i>Smodels</i>
P1	0.850	0.050	0.040+0.020
P2	0.820	0.050	0.050+0.020
P9	0.680	0.040	0.040+0.010
P10	0.800	0.070	0.050+0.000
P11	0.840	0.050	0.040+0.000
P16	0.940	0.060	0.050+0.010
P17	0.880	0.050	0.050+0.010
P18	0.810	0.030	0.030+0.000

Table 4.2: Comparing *Smodels* and *DLV* on the Simple-Travel Do-main

Problem	<i>Smodels</i>	<i>DLV</i>	<i>DLV</i> grounding+ <i>Smodels</i>
S1-0	0.050	0.040	0.030+0.010
S2-0	0.330	0.060	0.050+0.010
S3-0	1.390	0.080	0.010+0.090
S4-0	4.540	0.260	0.100+0.530
S5-0s1	19.530	0.640	0.080+1.540
S5-0s2	20.630	0.680	0.090+1.840
S6-0	23.150	0.980	0.170+3.560

Table 4.3: Comparison of *Smodels* and *DLV* using **Trans**(\cdot) on the Miconic-10-simtest Domain

Tables 4.2 and 4.3 show our results on the Simple-Travel and Miconic-10-simtest problems. We compared our encodings using *Smodels* with *lparse* for grounding, *DLV*, and *Smodels* with *DLV* for grounding.

Chapter 4: Answer Set Programming		Combining Agents, ASP and Planning, NICTA 2003
Problem	<i>DLV</i>	<i>DLV</i> grounding+ <i>Smodels</i>
P1	0.590	0.510+0.330
P7	16.440	14.340+35.210
P8	26.180	22.630+85.390
P10	27.220	24.840+52.730
P15	17.020	14.960+38.190
P20	168.630	163.660+329.940
P22	2025.16	1578.69+no solution
P23	4275.25	4236.60+no solution
P25	4619.24	4585.35+no solution

Table 4.4: Comparison of *Smodels* and *DLV* on the Zeno-Travel Domain

4.9 Benchmarks and Comparisons	259
--------------------------------	-----

Note that the last column contains the sum of (1) the time for producing the grounded version by *DLV*, and (2) the time it takes for *Smodels* to produce the solution based on this grounding. This sum is larger than the overall time in the *DLV* column.

Chapter 4: Answer Set Programming		Combining Agents, ASP and Planning, NICTA 2003	
Problem	<i>Smodels</i>	<i>DLV</i>	<i>DLV</i> grounding+ <i>Smodels</i>
P1	0.770	0.050	0.050+0.010
P7	0.820	0.050	0.050+0.020
P8	0.810	0.040	0.030+0.010
P9	0.790	0.030	0.030+0.000
P14	0.820	0.050	0.050+0.020
P15	0.680	0.030	0.030+0.000
P16	0.980	0.050	0.050+0.010
P17	0.710	0.040	0.040+0.010
P18	0.750	0.030	0.030+0.000

Table 4.5: Comparison of *Smodels* and *DLV* on the Simple-Travel Domain using disjunctions

4.9 Benchmarks and Comparisons	261
--------------------------------	-----

Comparison with SHOP

Problem	<i>Smodels</i>	<i>DLV</i>	<i>DLV</i> ground+ <i>Smodels</i>	SHOP
P1	0.850	0.050	0.040+0.020	0.000
P2	0.820	0.050	0.050+0.020	0.000
P8	0.760	0.040	0.040+0.010	0.000
P9	0.680	0.040	0.040+0.010	0.000
P10	0.800	0.070	0.050+0.000	0.000
P16	0.940	0.060	0.050+0.010	0.000
P17	0.880	0.050	0.050+0.010	0.000
P18	0.810	0.030	0.030+0.000	0.000

Table 4.6: Comparison of *Smodels* and *DLV* with SHOP on the Simple-Travel Domain

Problem	<i>Smodels</i>	<i>DLV</i>	<i>DLV</i> grnd+ <i>Smodels</i>	SHOP	Ratio (<i>DLV</i> / SHOP)
S1-0	0.050	0.040	0.030+0.010	0.000	-
S2-0	0.330	0.060	0.050+0.010	0.010	6
S3-0	1.390	0.080	0.010+0.090	0.000	-
S4-0	4.540	0.260	0.100+0.530	0.020	13
S5-0s1	19.530	0.640	0.080+1.540	0.060	10.67
S5-0s2	20.630	0.680	0.090+1.840	0.060	11.33
S6-0	23.150	0.980	0.170+3.560	0.090	10.89

Table 4.7: Comparison of *Smodels* and *DLV* using **Trans**(·) with SHOP on the Miconic-10-simtest Domain

Chapter 4: Answer Set Programming		Combining Agents, ASP and Planning, NICTA 2003	
Problem	<i>DLV</i>	SHOP	Performance Ratio (<i>DLV</i> / SHOP)
P7	16.440	0.020	822.00
P8	26.180	0.030	872.67
P9	38.390	0.070	548.43
P13	16.560	0.060	276.00
P15	17.020	0.020	851.00
P16	78.060	0.090	867.34
P17	66.300	0.060	1105.00
P22	2025.16	3.050	663.99
P25	4619.24	12.860	359.19
Table 4.8: Comparison of <i>DLV</i> , SHOP on Zeno-Travel Domain			
4.9 Benchmarks and Comparisons		264	

The results of our experiments on the Zeno-Travel Domain can be seen at Table 4.8. In most cases, the time needed by our program using *DLV* was 1.5 to 2.5 orders of magnitude more than SHOP.

Although there is not enough data to say so conclusively, this suggests that the average-case time complexity of our programs may be roughly the same as that of SHOP.

This gives reason to hope that future improvements in our programs and in ASP solvers may make it possible to get performance competitive with planning systems such as SHOP.

4.10 Summary and References

Answer Set Programming evolved out of PROLOG and deductive databases. It is purely declarative programming and can be seen as propositional PROLOG with loop detection.

1. ASP allows for **disjunctions and negations**.

All problems **located on the second level of the polynomial hierarchy** are expressible in disjunctive ASP.

2. It does not allow **function symbols**, but allows variables.
However, a given program will first be completely grounded before computing answer sets.

3. To avoid problems with bottom-up computation, rules are required to be **safe**.
4. There are efficient implementations of the ASP paradigm: *Smodels*, *DLV*.
5. **HTN planning** can be nicely captured within ASP: the resulting system is often **within 1-2 orders of magnitude** compared to a dedicated planner (SHOP).
It also seems to scale up.

References

Brass, S. and J. Dix (1994). A disjunctive semantics based on unfolding and bottom-up evaluation. In B. Wolfinger (Ed.), *Innovationen bei Rechen- und Kommunikationssystemen*, (IFIP '94-Congress, Workshop FG2: Disjunctive Logic Programming and Disjunctive Databases), Berlin, pp. 83–91. Springer.

Dix, J. and M. Müller (1994a). Partial Evaluation and Relevance for Approximations of the Stable Semantics. In Z. Ras and M. Zemankova (Eds.), *Proceedings of the 8th Int. Symp. on Methodologies for Intelligent Systems, Charlotte, NC, 1994*, LNAI 869, Berlin, pp. 511–520. Springer.

- Dix, J. and M. Müller (1994b). The Stable Semantics and its Variants: A Comparison of Recent Approaches. In L. Dreschler-Fischer and B. Nebel (Eds.), *Proceedings of the 18th German Annual Conference on Artificial Intelligence (KI '94), Saarbrücken, Germany*, LNAI 861, Berlin, pp. 82–93. Springer.
- Nau, D., Y. Cao, A. Lotem, and H. Muñoz-Avila (1999). Shop: Simple hierarchical ordered planner. In *Proceedings of IJCAI-99*.
- Sakama, C. and H. Seki (1994). Partial Deduction of Disjunctive Logic Programs: A Declarative Approach. In *Logic Program Synthesis and Transformation – Meta Programming in Logic*, LNCS 883, Berlin, pp. 170–182. Springer.

Son, T., C. Baral, and S. McIlraith. (2001, September). Planning with domain-dependent knowledge of different kinds – an answer set programming approach. In T. Eiter, M. Truszczyński, and W. Faber (Eds.), *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Sixth International Conference*, LNCS 2173, Berlin, pp. 226–239. Springer.

Subrahmanian, V., P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Özcan, and R. Ross (2000). *Heterogenous Active Agents*. MIT-Press.

Weiss, G. (Ed.) (1999). *Multi-Agent Systems*. MIT-Press.

References

Combining Agents, ASP and Planning, NICTA

References

Arens, Y., C. Y. Chee, C.-N. Hsu, and C. Knoblock (1993). Retrieving and Integrating Data From Multiple Information Sources. *International Journal of Intelligent Cooperative Information Systems* 2(2), 127–158.

Arisha, K., F. Ozcan, R. Ross, V. Subrahmanian, T. Eiter, and S. Kraus (1999, March/April). IMPACT: A Platform for Collaborating Agents. *IEEE Intelligent Systems* 14, 64–72.

Bayardo, R., et al. (1997). Infosleuth: Agent-based Semantic Integration of Information in Open and Dynamic Environments. In J. Peckham (Ed.), *Proceedings of ACM SIGMOD Conference on Management of Data*, Tucson, Arizona, pp. 195–206.

Brass, S. and J. Dix (1994). A disjunctive semantics based on unfolding and bottom-up evaluation. In B. Wolfinger (Ed.), *Innovationen bei Rechen- und Kommunikationssystemen*, (IFIP ’94-Congress, Workshop FG2: Disjunctive Logic Programming and Disjunctive

623

Databases), Berlin, pp. 83–91. Springer.

Bratman, M., D. Israel, and M. Pollack (1988). Plans and Resource-Bounded Practical Reasoning. *Computational Intelligence* 4(4), 349–355.

Brink, A., S. Marcus, and V. Subrahmanian (1995). Heterogeneous Multimedia Reasoning. *IEEE Computer* 28(9), 33–39.

Chawathe, S., et al. (1994, October). The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *Proceedings of the 10th Meeting of the Information Processing Society of Japan*, Tokyo, Japan. Also available via anonymous FTP from host db.stanford.edu, file /pub/chawathe/1994/tsimmis-overview.ps.

Currie, K. and A. Tate (1991). O-plan: the open planning architecture. *Artificial Intelligence* 52(1).

Dix, J., T. Eiter, M. Fink, A. Polleres, and Y. Zhang (2003). Monitoring Agents using Declarative Planning. In R. Kruse (Ed.), *Proceedings of the 27th German Annual Conference on Artificial Intelligence (KI*

References

Combining Agents, ASP and Planning, NICTA

'03), *Hamburg, Germany*, LNAI ???, Berlin. Springer.

Dix, J., T. Eiter, M. Fink, A. Polleres, and Y. Zhang (2004). Monitoring Agents using Declarative Planning. *Fundamenta Informaticae*, to appear.

Dix, J., S. Kraus, and V. Subrahmanian (2001). Temporal agent reasoning. *Artificial Intelligence* 127(1), 87–135.

Dix, J., S. Kraus, and V. Subrahmanian (2002, July). Agents dealing with time and uncertainty. In C. Castelfranchi and W. L. Johnson (Eds.), *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*. New York: ACM Press.

Dix, J., U. Kuter, and D. Nau (2003). Planning in answer set programming using ordered task decomposition. In R. Kruse (Ed.), *Proceedings of the 27th German Annual Conference on Artificial Intelligence (KI '03), Hamburg, Germany*, LNAI ???, Berlin. Springer.

625

Dix, J. and M. Müller (1994a). Partial Evaluation and Relevance for Approximations of the Stable Semantics. In Z. Ras and M. Zemankova (Eds.), *Proceedings of the 8th Int. Symp. on Methodologies for Intelligent Systems, Charlotte, NC, 1994*, LNAI 869, Berlin, pp. 511–520. Springer.

Dix, J. and M. Müller (1994b). The Stable Semantics and its Variants: A Comparison of Recent Approaches. In L. Dreschler-Fischer and B. Nebel (Eds.), *Proceedings of the 18th German Annual Conference on Artificial Intelligence (KI '94), Saarbrücken, Germany*, LNAI 861, Berlin, pp. 82–93. Springer.

Dix, J., H. Munoz-Avila, and D. N. an Lingling Zhang (2002). Theoretical and Empirical Aspects of a Planner in a Multi-Agent Environment. In G. Ianni and S. Flesca (Eds.), *Proceedings of Journees Europeens de la Logique en Intelligence artificielle (JELIA '02)*, LNCS 2424, pp. 173–185. Springer.

Dix, J., H. Munoz-Avila, D. Nau, and L. Zhang (2002, July). Planning in

-
- a multi-agent environment: Theory and practice. In C. Castelfranchi and W. L. Johnson (Eds.), *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*. New York: ACM Press.
- Dix, J., H. Munoz-Avila, D. Nau, and L. Zhang (2003). IMPACTing SHOP: Putting an AI planner into a Multi-Agent Environment. *Annals of Mathematics and AI* 37(4), 381–407.
- Dix, J., M. Nanni, and V. S. Subrahmanian (2000). Probabilistic agent reasoning. *ACM Transactions of Computational Logic* 1(2), 201–245.
- Dix, J., V. Subrahmanian, and G. Pick (2000). Meta Agent Programs. *Journal of Logic Programming* 46(1-2), 1–60.
- Eiter, T., W. Faber, N. Leone, G. Pfeifer, and A. Polleres (2002). A Logic Programming Approach to Knowledge-State Planning, II: The DLV^{κ} System. *Artificial Intelligence* 144(1-2), 157–211.
- Eiter, T. and V. Subrahmanian (1999). Heterogeneous Active Agents, II: Algorithms and Complexity. *Artificial Intelligence* 108(1-2),
-
- 627

- Eiter, T., V. Subrahmanian, and G. Pick (1999). Heterogeneous Active Agents, I: Semantics. *Artificial Intelligence* 108(1-2), 179–255.
- Eiter, T., V. Subrahmanian, and T. Rogers (2000). Heterogeneous Active Agents, III: Polynomially Implementable Agents. *Artificial Intelligence* 117(1), 107–167.
- Franklin, S. and A. Graesser (1997). Is it an Agent, or Just a Program? In J. P. Müller, M. Wooldridge, and N. R. Jennings (Eds.), *Intelligent Agents III*, Berlin, Germany. Springer-Verlag. LNAI Volume 1193.
- Genesereth, M. R. and S. P. Ketchpel (1994). Software Agents. *Communications of the ACM* 37(7), 49–53.
- Georgeff, M. and A. Lansky (1987). Reactive Reasoning and Planning. In *Proceedings of the Conference of the American Association of Artificial Intelligence*, Seattle, WA, pp. 677–682.
- Munoz-Avila, H., D. Aha, D. Nau, R. Weber, L. Breslow, and F. Yaman (2001). Sin: Integrating case-based reasoning with task

decomposition. In *Proceedings of IJCAI-01*.

Nau, D., Y. Cao, A. Lotem, and H. Muñoz-Avila (1999). Shop: Simple hierarchical ordered planner. In *Proceedings of IJCAI-99*.

Rao, A. S. (1995). Decision Procedures for Propositional Linear-Time Belief-Desire-Intention Logics. In M. Wooldridge, J. Müller, and M. Tambe (Eds.), *Intelligent Agents II – Proceedings of the 1995 Workshop on Agent Theories, Architectures and Languages (ATAL-95)*, Volume 890 of *LNAI*, pp. 1–39. Berlin, Germany: Springer-Verlag.

Rao, A. S. and M. Georgeff (1991). Modeling Rational Agents within a BDI-Architecture. In J. F. Allen, R. Fikes, and E. Sandewall (Eds.), *Proceedings of the International Conference on Knowledge Representation and Reasoning*, Cambridge, MA, pp. 473–484. Morgan Kaufmann.

Rao, A. S. and M. Georgeff (1995, June). Formal models and decision procedures for multi-agent systems. Technical Report 61,

Australian Artificial Intelligence Institute, Melbourne.

Sacerdoti, E. (1977). *A Structure for Plans and Behavior*. American Elsevier Publishing.

Sakama, C. and H. Seki (1994). Partial Deduction of Disjunctive Logic Programs: A Declarative Approach. In *Logic Program Synthesis and Transformation – Meta Programming in Logic*, LNCS 883, Berlin, pp. 170–182. Springer.

Son, T., C. Baral, and S. McIlraith. (2001, September). Planning with domain-dependent knowledge of different kinds – an answer set programming approach. In T. Eiter, M. Truszczyński, and W. Faber (Eds.), *Logic Programming and Non-Monotonic Reasoning, Proceedings of the Sixth International Conference*, LNCS 2173, Berlin, pp. 226–239. Springer.

Subrahmanian, V., P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Özcan, and R. Ross (2000). *Heterogenous Active Agents*. MIT-Press.

Tate, A. (1977). Generating Project Networks. In *Proc. IJCAI-77*, pp.

888–893.

Weiss, G. (Ed.) (1999). *Multi-Agent Systems*. MIT-Press.

Wiederhold, G. (1993). Intelligent Integration of Information. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Washington, DC, pp. 434–437.

Wilder, F. (1993). *A Guide to the TCP/IP Protocol Suite*. Artech House.

Wilkins, D. (1988). *Practical planning - extending the classical AI planning paradigm*. Morgan Kaufmann.

Wooldridge, M. J. and N. R. Jennings (1995). Agent Theories, Architectures and Languages: A survey. In M. J. Wooldridge and N. R. Jennings (Eds.), *Intelligent Agents*, Volume 890 of *Lecture Notes in Artificial Intelligence*, pp. 1–39. Springer-Verlag.