ANSWER SET BASED DESIGN OF HIGHLY AUTONOMOUS, RATIONAL AGENTS

by

MARCELLO BALDUCCINI, M.S.

A DISSERTATION

IN

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

DOCTOR OF PHILOSOPHY

December, 2005

To Sara and Nicole.

Nicole, if one day you'll
be reading this, be kind
in pointing out the
mistakes...

ACKNOWLEDGMENTS

This work has been substantially influenced by my advisor, Michael Gelfond, who provided me with constant help and support along all these years. I am deeply grateful to him for all he taught me, and in particular for showing me what it means to be a good researcher. I will do my best to follow his example in my future professional life.

I would like to thank all the members of my dissertation committee, Chitta Baral, Matthew R. Barry, Vladimir Lifschitz and Richard Watson for all their comments, which were invaluable in improving the quality of this dissertation.

I am very grateful to Daniel E. Cooke, Chair of the Computer Science Department, for making such a pleasant research environment possible, and for his support and encouragement since my first day at Texas Tech.

I am also thankful to all the present and former members of the Knowledge Representation Laboratory (KRLab) for their friendship and many interesting discussions. In particular, I would like to thank Monica Nogueira for inviting me to join the research on the Space Shuttle; Joel Galloway for being such a good friend and for his help in my first days as a graduate student; Veena Mellarkod for her support and for so many the enjoyable discussions on all sorts of topics.

I would like to thank United Space Alliance and NASA, whose funding through the Computer Science department has allowed me to concentrate on research all these years.

I am very grateful to my parents, because without their advice I would have never been here. Thank you for showing me the wonders of the world, for teaching me to be curious and not to be afraid of challenges. Most of all, thank you for encouraging me to follow my aspirations, even when that took me far away from you.

Finally, none of this would have been possible without the support, help and understanding of my wife Sara and my daughter Nicole. I am deeply grateful to them for encouraging me all along, bearing with me (not necessarily in relation to this dissertation), and for re-organizing their lives around my research in these past months.

Marcello Balduccini

Texas Tech University

December 2005

CONTENTS

ABSTRACT

Answer set programming is a knowledge representation methodology that combines a high level of abstraction and direct computability. This dissertation shows how complex rational agents can be built with the answer set programming methodology and the associated languages. We describe the design of agents capable of fairly sophisticated reasoning, including planning, diagnosis, inductive learning. More precisely, we show reasoning algorithms that, given a formal description of the domain, allow the agent to:

- generate plans, expected to achieve the agent's goal under reasonable assumptions;
- monitor the execution of actions and detect unexpected observations;
- explain unexpected observations by:
    - hypothesizing that some event occurred, unobserved, in the past;
    - modifying the original description of the domain to match the observations.

The reasoning approach is model-based, with the domain model written in action language $\mathcal{AL}$, and shared by all the reasoning components. The ability to use a single model for all the types of reasoning demonstrates the flexibility deriving from the use of answer set programming. Moreover, having a single model increases the ease of development, update and reuse of the domain description.

For the actual computation, the domain model is translated into A-Prolog using a novel encoding from $\mathcal{AL}$ into A-Prolog. The encoding differs from the previous ones in that it extracts, from the model in $\mathcal{AL}$, information that support not only planning and diagnosis, but also learning.

Two sets of reasoning components are presented. The first set is developed using A-Prolog. We demonstrate that the components in this set are capable of fairly sophisticated reasoning. Next, we show how the approach can be extended to substantially improve the quality of reasoning. For this, we develop CR-Prolog, a language in which A-Prolog is augmented by consistency-restoring rules (cr-rules) and preferences. The new constructs allow the encoding of qualitative preferences on the solutions of reasoning problems. The second set of reasoning components is obtained by extending the basic ones to allow the specification of preferences. This allows to reason, for example, about plans satisfying, if at all possible, a collection of requirements, about most likely diagnoses, and about most reasonable modifications of domain models. We demonstrate how the quality of reasoning performed by this set of components is substantially higher than that of the basic ones.

Since the reasoning techniques sometimes depend on whether the domain model is deterministic or non-deterministic, we also present a sufficient condition for the determinism of action descriptions. We show that the condition can be checked in polynomial time, and describe an A-Prolog program that performs the test.

LIST OF FIGURES

CHAPTER I
THE GOAL OF THIS RESEARCH

"In life, unlike chess, the game continues after checkmate."
Isaac Asimov (1920-1992)

The goal of this research work is to investigate the use of answer set programming and action languages in the design and implementation of intelligent agents. In particular, we are interested in developing reasoning algorithms that, given a formal description of the domain, allow the agent to:

- generate plans, expected to achieve the agent's goal under reasonable assumptions;
- monitor the execution of actions and detect unexpected observations;
- explain unexpected observations by:
  - hypothesizing that some event occurred, unobserved, in the past;
  - modifying the original description of the domain to match the observations.

In this dissertation, agents capable of performing these tasks are called *highly autonomous, rational* agents. The term "rational" denotes the fact that their behavior is based on rational thinking. The term "*highly* autonomous" indicates that they can deal with a broad range of unexpected circumstances of various kinds without requiring external intervention.

To simplify the study, we assume that the agent and the domain satisfy the following conditions:

- the domain can be represented by a transition diagram (for a definition of transition diagram, refer to Section 2.3);

- the agent is capable of making *correct* observations, performing actions, and remembering the domain history;
- normally, the agent is capable of observing all relevant exogenous events occurring in the domain;
- normally, the agent has correct and complete knowledge about the transition diagram that describes the domain.

Our choice is justified by the fact that these assumptions hold in many realistic domains and are suitable for a broad class of applications. Notice however that in other interesting domains some of the assumptions, in particular the first two, do not hold, e.g. when the effects of actions and the truth values of observations can only be known with a substantial degree of uncertainty that cannot be ignored in the modeling process. It remains to be seen if some of our methods can be made to work in such situations.

This research makes use of previous work on architectures of intelligent agents and on answer set based planning algorithms [44]. In recent years, an *observe-think-act* control loop for intelligent agents (refer to Chapter II for details) has been suggested [8], in which most of the reasoning is based on a formal model of the environment, shared by all the reasoning modules. The formal model is specified using an action language, as such high-level languages allow writing domain specifications that are usually compact and easy to read, thus increasing the designer's confidence in the correctness of the formalization.

Approximately in the same period, a translation was found from some relevant action languages (e.g. $\mathcal{AL}$) into A-Prolog. Based on this translation, planning algorithms were developed, in which the planning problem is encoded by an A-Prolog program by means of a translation of the action language specification into A-Prolog, and planning is reduced to finding the answer sets of such program (one-to-one correspondence between plans and answer sets is formally proven). Although the feasibility of the approach was demonstrated in theory and with small examples, initially it was not clear how this planning technique would scale to medium-sized applications.

Thus, the first step of this research work has been my contribution to a project (USA-Advisor) aimed at determining the scalability of answer set based planning techniques. The project and its successful results have been described in detail Monica Nogueira's Ph.D. Dissertation [57]. For a brief description, refer to Section 6.1.1.

Notice that the technique presented in [57] assumes that the action description encoding the domain's behavior be deterministic. Unfortunately, in general checking whether a domain specification is deterministic is in itself a non-trivial task. In fact, for anything but small domains, direct inspection of the domain specification is unreliable, and tests on the transition diagram are not an option due to its typical size. Therefore, we have found a sufficient condition for the determinism of action descriptions (refer to Chapter V), which substantially extends result from [8, 9]. We have also developed an A-Prolog implementation of such condition and have shown that the test can be performed in polynomial time.

Next, we have turned our attention to diagnostic tasks. In this direction, we have developed a theory of diagnosis, substantially simplifying and improving previous work by [11]. In our approach [3], the reasoner's task is that of finding exogenous actions, occurred in the past without him noticing them, whose occurrence justifies the agent's observations. In this sense, diagnosis can be seen as "planning in the past." A particularly attractive feature of this view is that planning and diagnosis share the *same* domain model and the *same formalization* of it. Based on the theory of diagnosis, reasoning algorithms have been developed, which reduce the computation of diagnoses to finding the answer sets of suitable logic programs. Finally, we have proved correctness and completeness of the algorithms, we have implemented them, and tested them on USA-Advisor. Diagnosis will be described in detail in Section 6.2.1.

Analysis of the algorithms for diagnosis has shown that, although reasonable diagnoses are always returned, often too many of them are found. To narrow the search to "best" diagnoses, a way to specify preferences among them had to be developed.

In the initial steps of our investigation on the specification of preferences, we found no natural, general way to specify preferences of this sort using A-Prolog.

This led us to the development of CR-Prolog – an extension of A-Prolog that allows the specification of preferences on the candidate solutions of a problem. CR-Prolog is described in Chapter VII. Together with Loveleen Kolvekal, we also developed an inference engine for CR-Prolog, which was implemented as part of Loveleen's Master Thesis [41].

CR-Prolog is essentially based on abduction of rules. CR-Prolog programs contain "regular rules", which cannot be abduced, and "consistency-restoring rules" (or cr-rules), which can be abduced. It is the separation into abducible and non-abducible rules that allows a seamless integration of deductive and abductive reasoning in the same program (refer to Chapter VII for a comparison with similar approaches).

Working with CR-Prolog, we have initially defined a methodology to specify possible rare events, their relative likelihood, and a way to propagate preferences on rules to preferences on possible solutions (notice that the propagation of preferences is made somewhat more complex by the fact that preferences in CR-Prolog can be *dynamic*). The methodology has been applied to diagnosis – exogenous actions are viewed as rare events, with associated relative likelihood – and yielded a reduction of the number of diagnoses returned, and a substantial increase in their quality.

Later we found that the methodology that had been developed for diagnosis can be applied directly to planning as well: given a collection of requirements that plans should satisfy, we view the violation of a requirement as a rare event. The introduction of such "defeasible" requirements yields a substantial increase in the quality of plans. The use of CR-Prolog in the reasoning modules is described in Chapter VIII.

To verify the applicability of our approach, we have implemented both planning and diagnostic modules using CR-Prolog and tested them on several examples. The planning technique has been successfully tested on medium-size examples including an extension of USA-Advisor. The diagnostic technique has been tested on small examples, and has yielded encouraging results. However, the efficiency of the inference engine needs to be increased before a thorough assessment on more complex examples can be performed.

Notice that the work described so far assumes the correctness of the action description. In some circumstances, however, the action description may be incomplete or incorrect. For example, the correct behavior of a circuit may be completely known in advance, but writing an action description specifying completely its behavior in presence of faults may be difficult or even unfeasible. The last phase of this research work consisted in developing a learning module that allows the agent to modify the action description given its observations on the environment.

Working in this direction, we have developed a theory of learning that is in some sense similar to the theory of diagnosis discussed earlier. One of its most important features is that the theory of learning is based on the *same* domain model (and same formalization of it) as diagnosis and planning. Based on the theory of learning, we have also developed corresponding reasoning algorithms, reducing learning to the computation of answer sets, and proofs of correctness and completeness. A detailed description of the learning module can be found in Section 6.2.2. (From the point of view of the existing research on learning, the reasoning performed by our learning module can be classified as performing incremental inductive learning.)

Finally, we have extended the use of CR-Prolog to the learning module, making it possible to specify preferences on the possible corrections of the action description. The result is a substantial reduction in the number of possible corrections, and an improved quality of the solutions. The use of CR-Prolog for learning is described in Chapter VIII.

CHAPTER II
BACKGROUND

This chapter contains background information about the types of environments that our agent is designed for, about action language $\mathcal{AL}$, about the agent's control loop, and about answer set programming.

2.1 Environments of Interest

In the rest of this dissertation we will often use the term *dynamic domain* to denote an environment whose state changes in response to the execution of actions.

Dynamic domains of interest in this dissertation are those satisfying the following conditions:

1. the evolution of the environment occurs in *discrete* steps;
2. the state of the environment can be defined by a set of boolean statements (called *fluents*, and later precisely defined);
3. actions are *instantaneous* and *deterministic*;
4. all fluents are *observable* (i.e. at any moment the agent can determine if they are true or false).

These conditions are introduced to reduce the complexity of the presentation, and to allow the reader to concentrate on the contributions of this dissertation. Almost none of them is essential to the validity of this study. In fact, most can be relaxed by adopting approaches already described in the literature: the boolean statements mentioned in condition (2) can be replaced by expressions with numerical values like in $\mathcal{C}+$ [36]; actions with duration (3) can be dealt with by introducing processes [63, 21, 22]; non-determinism (3) can be introduced in a way similar to [10]; non-observable fluents (4) have been dealt with for example in [35].

2.2 A-Prolog

A-Prolog is a knowledge representation language with roots in the research on the semantics of logic programming languages and non-monotonic reasoning [32, 33]. Over time, several extensions of the original language have been proposed [17, 56, 53, 23, 16]. In this dissertation, by the term *basic A-Prolog* we identify the language introduced in [32], and later extended with epistemic disjunction [33, 31]. The term basic A-Prolog programs is intended as a synonym of *disjunctive program*.

The syntax of A-Prolog is determined by a typed signature Σ consisting of types, typed object constants, and typed function and predicate symbols. We assume that the signature contains symbols for integers and for the standard functions and relations of arithmetic. Terms are built as in first-order languages.

By *simple arithmetic terms* of Σ we mean its integer constants. By *complex arithmetic terms* of Σ we mean terms built from legal combinations of arithmetic functions and simple arithmetic terms (e.g. $3 + 2 \cdot 5$ is a complex arithmetic term, but $3 + \cdot\ 2\ 5$ is not).

Atoms are expressions of the form $p(t_1, \dots, t_n)$, where p is a predicate symbol with arity n and t's are terms of suitable types. Atoms formed by arithmetic relations are called *arithmetic atoms*. Atoms formed by non-arithmetic relations are called *plain atoms*. We allow arithmetic terms and atoms to be written in notations other than prefix notation, according to the way they are traditionally written in arithmetic (e.g. we write $3 = 1 + 2$ instead of $= (3, +(1, 2))$).

Literals are atoms and negated atoms, i.e. expressions of the form $\neg p(t_1, \dots, t_n)$. Literals $p(t_1, \dots, t_n)$ and $\neg p(t_1, \dots, t_n)$ are called *complementary*. By $\bar{l}$ we denote the literal complementary to l.

Definition 2.2.1. *A* basic rule *r (of A-Prolog) is a statement of the form:*

$$h_1 \text{ OR } h_2 \text{ OR } \dots \text{ OR } h_k \leftarrow l_1, l_2, \dots l_m, not\ l_{m+1}, not\ l_{m+2}, \dots, l_n. \tag{2.1}$$

where $l_1, \dots, l_m$ are literals, and h_i's and $l_{m+1}, \dots, l_n$ are plain literals. We call h_1 OR h_2 OR $\dots$ OR h_k the head *of the rule (head(r));*

$l_1, l_2, \ldots l_m, not \;\; l_{m+1}, not \;\; l_{m+2}, \ldots, l_n$ *is its* body *(body(r)), and pos(r), neg(r) denote, respectively,* $\{l_1, \ldots, l_m\}$ *and* $\{l_{m+1}, \ldots, l_n\}$.

The informal reading of the rule (in terms of the reasoning of a rational agent about its own beliefs) is "if you believe $l_1, \ldots, l_m$ and have no reason to believe $l_{m+1}, \ldots, l_n$, then believe one of $h_1, \ldots, h_k$." The connective "not" is called *default negation*.

A rule such that $k = 0$ is called *constraint*, and is considered a shorthand of:

$$false \leftarrow \text{not } \; false, l_1, l_2, \ldots l_m, \text{not } \; l_{m+1}, \text{not } \; l_{m+2}, \ldots, l_n.$$

Definition 2.2.2. *A* basic A-Prolog program *is a pair* $\langle \Sigma, \Pi \rangle$*, where* Σ *is a signature and* Π *is a set of basic rules.*

In this dissertation we often denote programs of basic A-Prolog (and its extensions) by their second element. The corresponding signature is denoted by $\Sigma(\Pi)$. The terms, atoms and literals of a program Π are denoted respectively by $terms(\Pi)$, $atoms(\Pi)$ and $literals(\Pi)$.

Notice that the definition of the syntax of basic A-Prolog does not allow the use of variables. To simplify the presentation, in the rest of this dissertation we assume that programs containing variables (denoted by capital letters) are shorthands for the sets of their ground instantiations, obtained by substituting the variables with all the terms of appropriate type from the signature of the program. The approach is justified for the so called closed domains, i.e. domains satisfying the domain closure assumption [61] that all objects in the domain of discourse have names in the language of the program. Semantics of basic A-Prolog for open domains can be found in [7, 39].

The semantics of basic A-Prolog is defined in two steps. The first step consists in giving the semantics of programs that do not contain default negation. We will begin by introducing some terminology.

An atom is in *normal form* if it is an arithmetic atom or if it is a plain atom and its arguments are either non-arithmetic terms or simple arithmetic terms. Notice that atoms that are not in normal form can be mapped into atoms in normal form by applying the standard rules of arithmetic. For example, $p(4+1)$ is mapped into $p(5)$. For this reason, in the following definition of the semantics of basic A-Prolog, we assume that all literals are in normal form unless otherwise stated.

A literal l is *satisfied* by a consistent set of plain literals S (denoted by $S \models l$) if:

- l is an arithmetic literal and is true according to the standard arithmetic interpretation;
- l is a plain literal and $l \in S$.

If l is not satisfied by S, we write $S \not\models l$. An expression not l, where l is a plain literal, is satisfied by S if $S \not\models l$. A set of literals is satisfied by S if each element of the set is satisfied by S.

We say that a consistent set of plain literals S is *closed under a program* Π *not containing default negation* if, for every rule

$$h_1 \text{ OR } h_2 \text{ OR } \ldots \text{ OR } h_k \leftarrow l_1, l_2, \ldots l_m$$

of Π such that the body of the rule is satisfied by S, $\{h_1, h_2, \ldots, h_k\} \cap S \neq \emptyset$.

Definition 2.2.3 (Answer Set of a program without default negation). *A consistent set of plain literals, S, is an* answer set of a program Π not containing default negation *if S is closed under all the rules of* Π *and S is set-theoretically minimal among the sets satisfying the first property.*

Programs without default negation and whose rules have at most one literal in the head are called *definite*. It can be shown that definite programs have at most one answer set. The answer set of a definite program Π is denoted by $ans(\Pi)$.

The second step of the definition of the semantics consists in reducing the computation of answer sets of basic A-Prolog programs to the computation of the answer sets of programs without default negation, as follows.

Definition 2.2.4 (Reduct of a basic A-Prolog program). *Let* Π *be an arbitrary basic A-Prolog program. For any set S of plain literals, let* Π^S *be the program obtained from* Π *by deleting:*

- *each rule, r, such that $neg(r) \setminus S \neq \emptyset$;*

- *all formulas of the form not l in the bodies of the remaining rules.*

Definition 2.2.5 (Answer Set of a basic A-Prolog program). *A set of plain literals, S, is an* answer set of a basic A-Prolog program Π *if it is an answer set of Π^S.*

An interesting extension [30] of basic A-Prolog consists in the introduction of constructs that simplify representation and reasoning with sets of terms and with functions from such sets to natural numbers.

In this dissertation, we extend basic A-Prolog by adding to it *s-atoms* from [30], which allow to concisely represent subsets of sets of atoms. The resulting language will be called *A-Prolog.* Its syntax is defined as follows.

Definition 2.2.6. *A* s-atom *is a statement of the form:*

$$\{\overline{X} \ : \ p(\overline{X})\} \subseteq \{\overline{X} \ : \ q(\overline{X})\} \tag{2.2}$$

where $\overline{X}$ is the list of all free variables occurring in the corresponding plain *atom.*

Informally, the statement says that p is a subset of q. In A-Prolog, literals and s-atoms are disjoint sets. Literals and s-atoms are called *extended literals.* Rules are defined as follows.

Definition 2.2.7. *A* rule *(of A-Prolog) is a statement of the form (2.1), where l_i's are as before, and either (1) $k = 1$ and h_1 is a s-atom, or (2) all h_i's are plain literals.*

The reader may have noticed that, like in [30], negated atoms, $\neg p$, are not allowed to occur in s-atoms. However, differently from there, we allow negated atoms to occur in the head of the rules as well as in their bodies.

Notice that the combination of sets with classical and default negations introduces some subtleties. Consider the following informal argument. Suppose we are given a statement $\{\overline{X} \ : \ p(\overline{X})\} \subseteq \{\overline{X} \ : \ q(\overline{X})\}$ and we know $q(a)$ and $\neg q(b)$, but have no information about $q(c)$. Clearly, $p(a)$ satisfied the condition. But can we about $\neg p(b)$? And what about $p(c)$ or $\neg p(c)$?

To restrict ourselves to cases in which the meaning of s-atoms is unambiguous, we give the following definition of A-Prolog program.

Definition 2.2.8. *An* A-Prolog program *is pair* $\langle \Sigma, \Pi \rangle$, *where* Σ *is a signature,* Π *is a set of A-Prolog rules, and* for every atom $r(\overline{X})$ that occurs in the scope of an s-atom, Π contains the rule:

$$\neg r(\overline{X}) \leftarrow not\ r(\overline{X}).$$

(which encodes the Closed World Assumption on $r(\overline{X})$*).*

Thanks to this restriction, the meaning of s-atoms in our programs is unambiguous. Going back to the previous example, and assuming the Close World Assumption for p and q is part of the program, it can be shown that, for every x, $p(x)$ if $q(x)$ and $\neg p(x)$ otherwise.

To simplify the presentation, in the rest of this dissertation we will leave the Closed World Assumption implicit in the case of predicates that never occur in the scope of classical negation.

To define the semantics of A-Prolog, we introduce the following terminology. Let Σ be a signature and S be a set of plain literals from Σ. A s-atom (2.2) from Σ is true in S if, for any sequence $\bar{t}$ of ground terms from Σ, either $p(\bar{t}) \notin S$ or $q(\bar{t}) \in S$.

The following definition is similar to the notion of reduct introduced earlier.

Definition 2.2.9 (Set-Elimination). *Let* Π *be an arbitrary A-Prolog program. For any consistent set* S *of plain literals, the set-elimination of* Π *with respect to* S *(denoted by* $se(\Pi, S)$*) is the program obtained from* Π *by:*

- *removing from* Π *all the rules whose bodies contain s-atoms not satisfied by* S*;*
- *removing all remaining s-atoms from the bodies of the rules;*

- *replacing rules of the form $l_s \leftarrow \Gamma$, where l_s is an s-atom not satisfied by S, by rules $\leftarrow \Gamma$;*

- *replacing each remaining rule*

$$\{\overline{X} \;:\; p(\overline{X})\} \subseteq \{\overline{X} \;:\; q(\overline{X})\} \leftarrow \Gamma$$

  by a set of rules of the form $p(\bar{t}) \leftarrow \Gamma$ for each $p(\bar{t})$ from S.

Finally we are ready to define the notion of answer set of an A-Prolog program.

Definition 2.2.10 (Answer Set of an A-Prolog program). *A consistent set of plain literals S from the signature of program Π is an* answer set *of Π if it is an answer set of $se(\Pi, S)$.*

A-Prolog rules of the form

$$\{\overline{X} \;:\; p(\overline{X})\} \subseteq \{\overline{X} \;:\; q(\overline{X})\} \leftarrow \Gamma \tag{2.3}$$

are called *selection rules*. It can be noted that selection rules are closely related to the choice rules

$$m\{p(\overline{X}) : q(\overline{X})\}n \leftarrow \Gamma \tag{2.4}$$

introduced in [69, 56]. Proposition 5 of [30] makes this connection precise. Adapted to the language used here, the proposition states the following.

Proposition 2.2.1. *For every program Π such that:*

1. *Π contains a rule*

$$\{p(\overline{X}) : q(\overline{X})\} \leftarrow \Gamma;$$

2. *no other rule of Π contains p in the head,*

let Π^{++} be the program obtained from Π by replacing the choice rule with selection rule (2.3). Then, S is an answer set of Π iff S is an answer set of Π^{++}.

One limitation of our definition of A-Prolog with respect to the language of [69, 56] is that it does not allow the specification of bounds, i.e. of the lower and upper number of elements of the subset defined by $\{\overline{X} : p(\overline{X})\} \subseteq \{\overline{X} : q(\overline{X})\}$. For simple bounds such as those used in this dissertation (we use only lower or upper bounds of 1) we will use simple constraints, and avoid the introduction of the f-atoms from [30]. For example, imposing a maximum limit of 1 on the cardinality of the set (assuming that the arity of p is 1) can be achieved by means of the constraint:

$$\leftarrow p(X_1), p(X_2), X_1 \neq X_2.$$

Imposing a lower bound of 1 is equally easy:

$$\leftarrow \text{not } \mathit{some\_p}.$$
$$\mathit{some\_p} \leftarrow p(X).$$

To simplify the notation, from now on we use the statement:

$$m\{p(\overline{X}) : q(\overline{X})\}n \leftarrow \Gamma.$$

where m and n, if present, are 1, as an abbreviation of (2.3) together with the appropriate constraints to limit the cardinality of p.

The next section describes the high-level language used in the formalization of domain descriptions. This language is intended to be used by humans when formalizing the domain description. We will later see how the high-level specification is translated into A-Prolog in order to be used by the reasoning algorithms of the agent.

2.3 Action Language $\mathcal{AL}$

Action languages [34] are formalisms used to talk about the effects of actions. In these languages, the domain is represented by a transition diagram, i.e. a directed graph with nodes corresponding to the states of the domain, and arcs corresponding to transitions from one state to another. Arcs are labeled by actions, according to the intuition that state transitions are caused by the execution of actions. In our agent architecture, the agent is given in input a description of the domain that the agent uses to perform its reasoning tasks (e.g. to determine whether a sequence of actions achieves the desired effect).

Action languages are rather simple languages with respect to both syntax and semantics. They are good candidates for the high-level specification of the domain model: their relative simplicity and the clear intuitive reading of the statements usually allow the designer to have good confidence in the correctness of the specification that he came up with.

The formalism used in this dissertation to describe the domain is called $\mathcal{AL}$, and essentially follows the description given in [8]. The main difference with [8] is the introduction of the distinction between properties that depend on time and properties that do not. (This distinction is used in the reasoning component that performs inductive learning.)

Language $\mathcal{AL}$ is parametrized by an *action signature*, defining the alphabet used in the language. As in [8], $\mathcal{AL}$ is divided in three components: an *action description language*, a *history description language*, and a *query language*. The action description language, $\mathcal{AL}_d$, is used to describe the actions and their effects. The history description language, $\mathcal{AL}_h$, specifies the history of the domain. The query language, $\mathcal{AL}_q$, encodes queries about the domain. The next sections describe the various components of $\mathcal{AL}$.

2.3.1 Action Signature of $\mathcal{AL}_d$

An *action signature* of $\mathcal{AL}_d$ is a tuple $\langle C, V, P_S, P_F, N_A, TS \rangle$, where:

- C is a set of *symbols for constants*;
- V is a set of *symbols for variables*;
- P_S is a set of *symbols for static predicates*;
- P_F is a set of *symbols for fluent predicates*;
- N_A is a set of *symbols for action names*;

- TS is a set of non-negative integers $0, 1, 2, \ldots$.

Intuitively, static predicates are predicates whose truth does not depend on the state of the domain; fluent predicates are predicates whose truth depends on the state of the domain; the integers from TS are used to denote *steps* in the evolution of the domain.

A *term* is either a constant or a variable. A *fluent* is a statement

$$f(t_1, \ldots, t_n)$$

where f is a fluent predicate and t_i's are terms. A *fluent literal* is either a fluent or its classical negation, $\neg f(t_1, \ldots, t_n)$. A *static* is a statement

$$r(t_1, \ldots, t_n)$$

possibly prefixed by $\neg$, where r is a static predicate and t_i's are terms. An *AL-literal* is either a fluent literal or a static. An *elementary action* is a statement

$$a(t_1, \ldots, t_n)$$

where a is an action name and t_i's are terms. A *compound action* is a set $\{a_1, \ldots, a_n\}$ of elementary actions. Intuitively, the occurrence of a compound action corresponds to the simultaneous occurrence of the elementary actions it consists of. To simplify notation, we will sometimes use an elementary action, a_e, in place of the corresponding singleton, $\{a_e\}$. Unless otherwise specified, in this dissertation the term *action* will refer to elementary actions.

A *ground* fluent literal is a fluent literal that contains no variables. Similarly we define ground statics, ground AL-literals, and ground elementary actions. A ground compound action is a set of ground elementary actions. AL-literals and actions are called *tokens*.

Given an action signature, Σ, we denote its constants, variables, terms, fluents, fluent literals, statics, AL-literals, and actions by, respectively, $const(\Sigma)$, $var(\Sigma)$, $term(\Sigma)$, $fluent(\Sigma)$, $lit(\Sigma)$, $static(\Sigma)$, $pcond(\Sigma)$, and $action(\Sigma)$.

2.3.2 Syntax of $\mathcal{AL}_d$

A *dynamic law* of $\mathcal{AL}_d$ is a statement:

$$d : a \text{ causes } l \text{ if } \quad c_1, \ldots, c_n \tag{2.5}$$

where:

- d is a constant, used to name the dynamic law;
- a is a compound action;
- l is a fluent literal;
- c_i's are AL-literals.

Informally, a dynamic law says that, if action a were to be executed in a state in which $c_1, \ldots, c_n$ hold, fluent literal l would be caused to hold in the resulting state.

The name of the law is not used to define the semantics of the language, and can thus be omitted [1].

A *state constraint* of $\mathcal{AL}_d$ is a statement:

$$s : \text{ caused } l \text{ if } \quad c_1, \ldots, c_n \tag{2.6}$$

where s is the name of the state constraint, and l, c_i's are as before. Informally, a state constraint says that, in every state, the truth of $c_1, \ldots, c_n$ is sufficient to cause the truth of l. *The keyword caused is optional and can be omitted.*

An *impossibility condition* of $\mathcal{AL}_d$ is a statement:

$$b : a \text{ impossible\_if } \quad c_1, \ldots, c_n \tag{2.7}$$

where b is the name of the impossibility condition, and a, c_i's are as before. Informally, an impossibility condition states that action a cannot be performed in any state in which $c_1, \ldots, c_n$ hold.

[1] In the chapter on the encoding of $\mathcal{AL}$ into A-Prolog, we will assume that a name has been specified for each law, as this allows to simplify the presentation. The assumption however is not essential and can be easily lifted.

We use the term *law* to refer to dynamic laws, state constraints, and impossibility conditions. The *body* of a law is the set of its *preconditions* $c_1, \dots, c_n$. The *head* of a law (2.5) or (2.6) is fluent literal l. The *trigger* of a law (2.5) or (2.7) is action a. The name, body, head, and trigger of law w will be denoted by $name(w)$, $body(w)$, $head(w)$, and $trigger(w)$ respectively. The set of the fluent literals that occur in $body(w)$ is denoted by $body_l(w)$. The set of statics from $body(w)$ is $body_r(w)$. To simplify the presentation, we define these functions so that, if w is a state constraint, $trigger(w) = \emptyset$, and, if w is an impossibility condition, $head(w) = \epsilon$ (we assume that ϵ does not occur in the action signature).

The *static knowledge base* is a complete set of statics. Intuitively, the static knowledge base lists the statics that are true.

Definition 2.3.1 (Action Description). *An* action description *of $\mathcal{AL}$ is a tuple $\langle \Sigma, L, K \rangle$, where:*

- *Σ is an action signature;*
- *L is a set of laws;*
- *K is a static knowledge base.*

Given an action description, AD, we denote its signature, laws, and static knowledge base by, respectively, $sig(AD)$, $law(AD)$, and $stat\_kb(AD)$.

In this dissertation, we define static knowledge bases by means of consistent A-Prolog programs with a unique answer set. This approach allows an elegant and concise encoding of rather complex static knowledge bases. To simplify the presentation, we allow action descriptions to be specified by tuples $\langle \Sigma, L, \Pi_K \rangle$, where Σ is an action signature, L is a set of laws, and Π_K is a consistent A-Prolog program with a unique answer set. Such a tuple denotes the action description $\langle \Sigma, L, K \rangle$, where K is the answer set of Π_K.

2.3.3 Semantics of $\mathcal{AL}_d$

The semantics of an action description, AD, is given by defining the transition diagram, $trans(AD)$, that corresponds to AD. In the rest of this section, we consider laws containing variables as an abbreviation for the collection of their ground instances. For this reason, unless otherwise stated, we *restrict our attention to ground tokens*. To define precisely the semantics of $\mathcal{AL}_d$, we need the following terminology and notation.

We say that a fluent literal, l, *holds in a set of fluent literals* S if $l \in S$. A static, r, *holds in* AD if $r \in stat\_kb(AD)$. The notion is extended to sets of fluent literals and sets of statics as usual.

Definition 2.3.2 (Satisfaction of AL-literals). *A set of AL-literals,* $C = \{c_1, \ldots, c_n\}$*, is* satisfied by a set of fluent literals S (under AD) *if the fluent literals of* C *hold in* S *and the statics of* C *hold in* AD*.*

Definition 2.3.3 (Closedness). *A set* S *of fluent literals is* closed under a state constraint, w, of AD *if* $head(w)$ *holds in* S *whenever* $body(w)$ *is satisfied by* S*. A set* S *of fluent literals is closed under a set,* Z*, of state constraints if it is closed under each element of* Z*.*

Definition 2.3.4 (Set of Consequences). *The set* $Cn_Z(S)$ *of* consequences *of* S *under* Z *is the smallest set of fluent literals that contains* S *and is closed under* Z*.*

Definition 2.3.5 (State). *A* state *is a complete and consistent set of fluent literals (i.e. for any fluent,* f*, either* f *or* $\neg f$ *belongs to the set, but not both), closed under the state constraints of* AD*.*

Given a ground compound action, a, a state, σ, and a dynamic law, d, of the form (2.5), we say that a fluent literal, l', is a *direct effect of* a *in* σ *w.r.t.* d if:

- $a \supseteq trigger(d)$;
- $l' = head(d)$;

- *body*(d) is satisfied by σ.

Definition 2.3.6 (Set of Direct Effects). *By $E(a, \sigma)$ we denote the set of all ground fluent literals that are direct effects of a in σ w.r.t. some dynamic law of AD.*

Definition 2.3.7 (Impossibility of Actions). *We say that ground action a is* impossible in state σ *if there exists an impossibility condition, b, such that: (*i*) $a \supseteq trigger(b)$, and (*ii*) body(b) is satisfied by σ. We say that a is executable in σ when a is not impossible in σ,*

Definition 2.3.8 (Transition Diagram). *The* transition diagram*, $trans(AD)$, encoded by an action description, AD, is the directed graph, $\langle N, R\rangle$, such that:*

- *N is the collection of all states of AD;*
- *R is the set of all triples $\langle \sigma, a, \sigma' \rangle$, where σ, σ' are states and a is a compound action, such that a is executable in σ, and*

  $$\sigma' = Cn_Z(E(a, \sigma) \cup (\sigma \cap \sigma')) \tag{2.8}$$

  where Z is the set of all state constraints of AD. State σ' is called successor state *of σ.*

The argument of Cn_Z in (2.8) is the union of the set, $E(a, \sigma)$, of the direct effects of a, with the set, $\sigma \cap \sigma'$, of the facts that are "preserved by inertia". The application of Cn_Z adds the "indirect effects" to this union.

Notice that it is possible for the transition diagram to contain no arcs labeled by an action, a, even though dynamic laws for a are specified in the action description. Consider the action description:

$$a_e \text{ causes } p.$$
$$a_e \text{ causes } \neg p.$$

For any pair of states σ, σ', $Cn_Z(E(a_e, \sigma) \cup (\sigma \cap \sigma'))$ contains $\{p, \neg p\}$. Since the set if inconsistent, (2.8) is never satisfied, and the corresponding transition diagram consists of two nodes, $\{p\}$, $\{\neg p\}$ and no arcs.

Definition 2.3.9 (Deterministic Action Descriptions). *We call an action description* deterministic *if, for any state, σ, and action, a, there is at most one successor state σ'.*

The above definition of $trans(AD)$ is based on results from [50, 49], which are the product of a long investigation on the nature of causality. (See for instance, [42, 74].) Finding this definition required a good understanding of the nature of causal effects of actions in the presence of complex interrelations between fluents. An additional level of complexity is added by the need to specify what is not changed by actions. The latter, known as the *frame problem*, is often reduced to the problem of finding a concise and accurate representation of the inertia axiom – a default which says that *things normally stay as they are* [37]. The search for such a representation substantially influenced AI research during the last twenty years. An interesting account of history of this research together with some possible solutions can be found in [68].

2.3.4 History Description Language, $\mathcal{AL}_h$

To describe the history of the domain, we use integers from set TS in the action signature to denote the step (in the evolution of the domain) that each observation refers to. Intuitively, step s corresponds to the state of the domain reached after the execution of s compound actions.

2.3.4.1 Syntax of $\mathcal{AL}_h$

Observations about the domain have two possible forms:

- $hpd(a, s)$
- $obs(l, s)$

where a is a ground action, l is a ground fluent literal, and s is a step. The first statement informally says that action a was observed to occur at step s. The second statement intuitively states that fluent literal l was observed to be true at step s. We call the former *observations about (occurrences of) actions*, and the latter *observations about (the truth of) fluent literals.*

We often use consistent A-Prolog programs with a unique answer set to compactly encode sets of observations. The set of observations described by such an A-Prolog program, Π, consists of the set of statements $hpd(a, s)$, $obs(l, s)$ from the answer set of Π. When there is no ambiguity, we allow A-Prolog programs to be specified in place of sets of observations.

Definition 2.3.10 (Recorded History). *A* (recorded) history *of the domain is a pair $\langle H, cT\rangle$, where H is a set of observations and cT is a step, denoting the current time step, satisfying the following conditions:*

- *$cT > s$ for every $hpd(a, s) \in H$;*
- *$cT \geq s$ for every $obs(l, s) \in H$.*

We often denote a history, $\langle H, cT\rangle$, by H^{cT}. To simplify the presentation, the notation $e \in H^{cT}$ is used as an abbreviation of "e belongs to the set of observations, H, of history H^{cT}."

Definition 2.3.11 (Domain Description). *A* domain description *is a pair $\langle AD, H^{cT}\rangle$, where AD is an action description and H^{cT} is a recorded history.*

This definition completes the description of the syntax of $\mathcal{AL}_h$. Its semantics is intuitively based on the set of paths in the transition diagram that match the observations, as follows.

2.3.4.2 Semantics of $\mathcal{AL}_h$

Definition 2.3.12 (State-Action Sequence). *A* state-action sequence *is a sequence $\pi = \langle \sigma_0, a_0, \sigma_1, \ldots, a_{n-1}, \sigma_n\rangle$ such that σ_i's are states and a_i's are compound actions.*

We say that the *length* of π is the number of states in it (in this case, $n + 1$). The length of π is denoted by $length(\pi)$.

To define the semantics of $\mathcal{AL}_h$, we also need the following notation. Given a state-action sequence $\pi = \langle \sigma_0, a_0, \sigma_1, \ldots, a_{n-1}, \sigma_n \rangle$ and a non-negative integer k, $\sigma(\pi, k)$ denotes state σ_k from π, and $act(\pi, k)$ denotes action a_k. To simplify the presentation, we define $act(\pi, length(\pi)) = \emptyset$.

Definition 2.3.13 (Path). *A state-action sequence, π, is a* path in *$trans(AD)$ if for every $0 \le i < length(\pi)$, $\langle \sigma(\pi, i), act(\pi, i), \sigma(\pi, i+1) \rangle$ is an arc from $trans(AD)$.*

Definition 2.3.14 (Model of a History). *A path, π, in $trans(AD)$ is a* model of H^{cT} (with respect to AD) *if $length(\pi) = cT + 1$ and, for every $0 \le s \le cT$:*

- *$act(\pi, s) = \{a \mid hpd(a, s) \in H^{cT}\}$;*
- *if $obs(l, s) \in H^{cT}$, then $l \in \sigma(\pi, s)$.*

In the sections on the reasoning algorithms of the agent, it will be often important to be able to check if a history has at least one model. The next definition elaborates on this idea.

Definition 2.3.15 (Consistency). *We say that H^{cT} is* consistent (with respect to *AD) if it has a model. A domain description $D = \langle AD, H^{cT} \rangle$ is* consistent *if H^{cT} is consistent with respect to AD.*

Next, we describe the query component of $\mathcal{AL}$.

2.3.5 Query Language, $\mathcal{AL}_q$

Various reasoning tasks can be essentially reduced to answering queries about properties of the domain. Our query language, $\mathcal{AL}_q$, consists of statements of the form:

- $h(\{l_1, \ldots, l_n\}, s)$
- $h\_after(\{l_1, \ldots, l_n\}, \langle a_1, a_2, \ldots, a_k \rangle)$

where l_i's are fluent literals, s is a step, and a_i's are compound actions. Intuitively, the query $h(\{l_1, \ldots, l_n\}, s)$ checks whether all the fluent literals are expected to hold at step s of the evolution of the domain. The query $h\_after(\{l_1, \ldots, l_n\}, \langle a_1, a_2, \ldots, a_k\rangle)$ tests whether the sequence of actions $\langle a_1, \ldots, a_k\rangle$ can be executed at the current step of the evolution of the domain, and if $l_1, \ldots, l_n$ are expected to hold after the execution of $\langle a_1, \ldots, a_k\rangle$. The following definitions formalize the intuition.

Definition 2.3.16 (Entailment Relation). *For every domain description $D = \langle AD, H^{cT}\rangle$:*

- *Model M* entails *a query $h(\{l_1, \ldots, l_n\}, s)$ if:*
  - *$0 \leq s \leq cT$;*
  - *for every $1 \leq i \leq n$, $l_i \in \sigma(M, s)$.*
- *Model M* entails *a query $h\_after(\{l_1, \ldots, l_n\}, \langle a_0, a_1, \ldots, a_k\rangle)$ if there exists a path, π, in $trans(AD)$ such that:*
  - *π contains $cT + k + 1$ states;*
  - *$\sigma(\pi, s) = \sigma(M, s)$ for $0 \leq s \leq cT$;*
  - *$act(\pi, s) = act(M, s)$ for $0 \leq s < cT$;*
  - *$act(\pi, cT + i) = a_i$ for $0 \leq i \leq k$;*
  - *for every $0 \leq i \leq n$, $l_i \in \sigma(\pi, cT + k + 1)$.*
- *The history, H^{cT},* entails *a query Q if, for every model M of H^{cT}, M entails Q.*

To simplify the notation, when the set of fluent literals that occurs in a query is a singleton, we represent it by its only element (e.g., we write $h(l, s)$ instead of $h(\{l\}, s)$). The entailment relation between a model, M, and a query, Q, is denoted by $M \models_{AD} Q$; the entailment relation between a domain description, $D = \langle AD, H^{cT}\rangle$, and a query, Q, is denoted by $H^{cT} \models_{AD} Q$.

2.4 The Agent's Control Loop

Our goal in this dissertation is to define an architecture for highly autonomous, rational agents, where the reasoning algorithms:

- are based on the ASP paradigm;
- share the same domain description;
- reduce reasoning to checking for consistency of the domain description and to computing the entailment relation;
- are provably sound and complete.

The architecture is based on the control loop in Figure 2.1, called *Observe-Think-Act loop.*

1. observe the world;
2. interpret the observations;
3. select a goal;
4. plan;
5. execute part of the plan.

Figure 2.1: The Observe-Think-Act Loop

Every step of the loop corresponds to either an input/output operation (**steps 1** and **5**) or to the execution of a reasoning algorithm. In the rest of this dissertation, the term *reasoning component* will be used as a synonym of *reasoning algorithm.* Also, by *core* of a reasoning algorithm we mean the set of CR-Prolog programs used in the algorithm.

Although it is possible to formalize a more complex architecture, e.g. where the execution of the reasoning algorithms depends on the available computational resources and other constraints, in this dissertation we will restrict our attention to the simple observe-think-act loop above. This will allow us to focus on the theoretical aspects of the design of the reasoning algorithms, and will simplify the proofs of the properties of the architecture.

In the next chapter, we give an overall description the intended behavior of the agent by means of a running example. The reasoning algorithms will be analyzed in detail in the later chapters.

CHAPTER III
AGENT BEHAVIOR: AN EXAMPLE

"The true delight is in the finding out rather than in the knowing."
Isaac Asimov (1920-1992)

Let us begin by assuming that, when the agent loop is first executed, the agent is initially given in input a domain description, $D_0 = \langle AD, H^0 \rangle$, and a partially ordered set of *goals*, G, where by *goal* we mean a finite set of fluent literals that the agent has to make true. The partial ordering encodes the relative importance of the goals. By cT we denote the current step in the evolution of the domain, and by Σ the action signature of AD. Also, we denote the *current* domain description by $D = \langle AD, H^{cT} \rangle$ (the domain description can be modified by the reasoning algorithms).

In the architecture, we divide the elementary actions specified by the signature of AD in *agent actions* and *exogenous actions.* Agent actions are those actions that the agent can perform. Exogenous actions are performed by nature or by other agents. The set of agent actions is denoted by $action_{ag}(\Sigma)$, and the set of exogenous actions by $action_{ex}(\Sigma)$. The two sets of actions are assumed to be disjoint. Compound agent actions are compound actions whose elements belong to $action_{ag}(\Sigma)$. Similarly, compound exogenous actions are subsets of $action_{ex}(\Sigma)$. Although exogenous actions may sometimes occur undetected by the agent, the agent operates under the assumption of being able to observe all exogenous actions. This assumption can be withdrawn if observations force the agent to conclude that some exogenous actions occurred undetected in the past.

To explain the observe-think-act loop, we refer to the simple electrical circuit shown in Figure 3.1, where switch sw_1 controls bulb b_1 and switch sw_2 controls bulb b_2. The circuit is powered by a battery, $batt$.

The agent can change the position of a switch, sw, by performing action $flip(sw)$. If $batt$ is malfunctioning, he can also replace it by performing action $replace(batt)$.

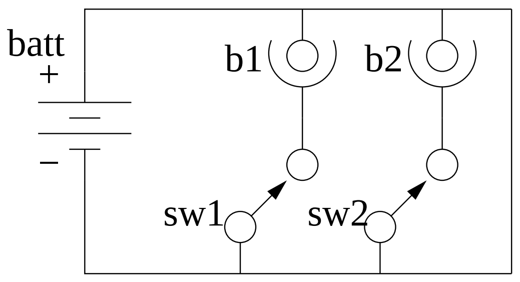


Figure 3.1: Circuit $\mathcal{AC}$

Finally, from time to time a bulb, b, may blow up – which is represented by the occurrence of action $blow\_up(b)$.

The state of the world is described by fluents:

- $closed(SW)$: switch SW is closed;
- $lit(B)$: blub B is lit;
- $ab(B)$: bulb B is malfunctioning, i.e. blown up;
- $ab(batt)$: $batt$ is malfunctioning.

In the following examples, we assume that the agent is given an action description stating that:

- flipping a switch inverts its position;
- action $blow\_up(b)$ causes switch b to blow up;
- when switch sw_i is closed, bulb b_i is lit, unless it is blown up or $batt$ is malfunctioning;
- when sw_i is open, b_i is off;
- action $replace(batt)$ replaces the battery with a functioning one.

At **step 1** of the *observe-think-act loop*, the agent gathers observations about the current state of the world. All the observations up to the current moment are recorded in H^{cT}. Notice that the observations gathered by the agent are *possibly incomplete.*

Example 3.1.1. *Suppose all switches are currently open, and the state of the bulbs and battery cannot be observed. Then, the agent at* ***step 1*** *may gather observations:*

$$\{\neg closed(sw_1), \neg closed(sw_2)\}.$$

After gathering the observations, the agent checks if they meet its expectations about the current state of the world (**step 2**). More precisely, the agent's expectations are met by its observations when:

$$\langle AD, H^{cT}\rangle \text{ is consistent.}$$

Our agent is designed to deal with inconsistency arising from:

- the undetected occurrence of some *exogenous actions*; or
- incompleteness or incorrectness of the action description.

Example 3.1.2. *Suppose the agent:*

1. *observes*
$$\{\neg closed(sw_1), \neg closed(sw_2), \neg ab(b_1), \neg ab(b_2),$$
$$\neg lit(b_1), \neg lit(b_2), \neg ab(batt)\}$$
2. *performs* $flip(sw_1)$
3. *observes*
$$\{closed(sw_1), \neg lit(b_1)\}$$

The observation $\neg lit(b_1)$ *is unexpected, as the execution of action* $flip(sw_1)$ *normally causes bulb* b_1 *to become lit. A possible explanation is that action* $blow\_up(b_1)$ *occurred together with* $flip(sw_1)$*. In fact,* $blow\_up(b_1)$ *would cause* b_1 *to become blown-up, and thus prevent it from becoming lit.*

Inconsistencies due to undetected exogenous actions are dealt with by finding which exogenous actions are likely to have occurred, undetected.

More precisely, the agent finds a recorded history, H^{cT}_{*}, such that:

- $H^{cT} \subseteq H^{cT}_{*}$;
- $H^{cT}_{*} \setminus H^{cT}$ is a subset of $\{hpd(a,s) \mid a \in action_{ex}(\Sigma) \wedge 0 \leq s < cT\}$;
- $\langle AD, H^{cT}_{*} \rangle$ is consistent.

Intuitively, the task is accomplished by viewing the history as a symptom, and reducing the problem of explaining the observations to that of computing the diagnoses of the symptom. Since, in general, several diagnoses can explain the symptom, the agent uses criteria (e.g., set-theoretic minimality) and tests (i.e., gathers further observations) to determine which diagnosis is the most plausible.

Example 3.1.3. *Going back to Example 3.1.2, the unexpected observation, $\neg lit(b_1)$, can be explained by assuming that $blow\_up(b_1)$ occurred, undetected, together with $flip(sw_1)$. To ensure that this is the correct diagnosis, the agent will then check if $ab(b_1)$ is true.*

If the occurrence of exogenous actions is not sufficient to explain the unexpected observations, the agent tries to update the action description to match the observations. The update is obtained by adding and removing laws, and by modifying the body of the existing laws.

More formally, the agent finds an action description, AD', such that

$$\langle AD', H^{cT} \rangle \text{ is consistent.}$$

Again, in general, several sets of modifications can be used to restore consistency. Hence, the agent is designed to perform tests to determine which set is likely to give the best results (in terms of generality of the modification, and expected correctness of the predictions in other states).

Example 3.1.4. *Suppose the agent:*

1. *observes*
$$\{closed(sw_1), \neg closed(sw_2), \neg ab(b_1), \neg ab(b_2),$$
$$lit(b_1), \neg lit(b_2), \neg ab(batt)\}$$

2. *performs* $flip(sw_2)$

3. *observes*
$$\{closed(sw_2), \neg lit(b_1), \neg lit(b_2)\}$$

Observations $\neg lit(b_1), \neg lit(b_2)$ *are unexpected. The agent tries to explain them initially by assuming that some exogenous actions occurred, undetected. The explanation consists of the assumption that actions* $blow\_up(b_1), blow\_up(b_2)$ *occurred when* $flip(sw_2)$ *was performed.*

Next, the agent tests the explanation. Notice that the occurrence of the two exogenous actions would cause bulbs b_1 *and* b_2 *to be blown. Checking whether* $ab(b_1), ab(b_2)$ *are true allows the agent to test if the explanation is correct.*

*Suppose the observations show that the two bulbs are not blown. This proves that the explanation is incorrect. Since there are no other possible explanations based on the occurrence of exogenous actions, the agent assumes that the action description is inaccurate. A possible correction is the addition of a law saying that "*if sw_1 and sw_2 are both closed, then $batt$ becomes malfunctioning.*" Notice that intuitively this corresponds to assuming that some sort of overloading occurs in the battery when both switches are closed.*

The corrected domain description would entail that $ab(batt)$ *is currently true. To validate the explanation, the agent tests whether* $ab(batt)$ *is actually true.*

After the agent has found a suitable interpretation for the observations, and has updated accordingly the domain description in its memory, the agent selects which goal to achieve (**step 3**) from set G. The selection of the current goal is performed taking into account information such as the partial ordering of goals, the history of the domain, the previous goal, and the action description (e.g., to evaluate how hard/time-consuming it will be to achieve a goal).

The next step consists of computing a plan, i.e. a sequence of actions, to achieve the selected goal (**step 4**).

More precisely, given a goal g from G, the agent finds a sequence $\langle a_1, \ldots, a_k \rangle$ of compound agent actions such that:

$$H^{cT} \models_{AD} h\_after(g, \langle a_1, \ldots, a_k \rangle).$$

In domains involving exogenous actions and/or uncertainty, it is unlikely that the execution of the plan will proceed exactly as the agent expects. (e.g., exogenous actions take the agent to an unforeseen state). That is why, at **step 5**, the agent selects only a part of the plan (e.g., the first compound action, a_1), and executes it.

Next, the agent goes back to observing the world (**step 1**). This allows him to find out whether the actions just executed took him to the expected state, and to compensate for any unexpected effect.

Example 3.1.5. *The following trace of the loop gives an example of the overall behavior of the agent. The agent:*

1. *observes*
$$\{closed(sw_1), \neg closed(sw_2), \neg ab(b_1), \neg ab(b_2),$$
$$lit(b_1), \neg lit(b_2), \neg ab(batt)\}$$
2. *selects the goal of making* $lit(b_2)$ *true*
3. *finds the plan* $\{flip(sw_2)\}$
4. *performs* $flip(sw_2)$
5. *observes*
$$\{closed(sw_2), \neg lit(b_1), \neg lit(b_2)\}$$
6. *explains the observations by adding to the action description a law stating that*

   *"*if sw_1 and sw_2 are both closed, then $batt$ becomes malfunctioning.*"*

7. *finds the plan* $\{flip(sw_1), repair(batt)\}$

8. *performs* $flip(sw_1)$

9. *observes*

$$\neg closed(sw_1)$$

10. *performs* $repair(batt)$

11. *observes*

$$\{closed(sw_2), lit(b_2)\}$$

The reasoning components of the loop are implemented in A-Prolog or an extension of it, called CR-Prolog (see Chapter VII). Simple procedural code fragments connect the various components and perform the input/output steps.

This combination of procedural and declarative languages has several advantages:

- the connection between the reasoning components is made easy by the flexibility of control of procedural languages;
- the small size of the procedural component simplifies the proof of correctness of the implementation;
- the availability of a formal, declarative, semantics for A-Prolog and its extensions simplifies the proofs of the relevant properties of the loop;
- the expressive power of A-Prolog allows for compact, yet easy to understand, implementations of the reasoning components.

The three major reasoning tasks performed by the agent are:

- planning;
- diagnostics;
- inductive learning.

Planning is used by the agent at step 4 of the loop. Diagnostics and inductive learning are used at step 2. Diagnostics is performed by the agent to find occurrences of exogenous actions that explain the observations. Inductive learning, instead, is used to correct the action description to match the observations. The reasoning algorithms are described in detail later in the dissertation.

In the next chapter we describe how $\mathcal{AL}$ is encoded in A-Prolog.

CHAPTER IV
ENCODING OF $\mathcal{AL}$ IN A-PROLOG

Language $\mathcal{AL}$ is designed to be easily written and understood by humans. To be used by the reasoning components of our agent, $\mathcal{AL}$ has to be translated into a form that they can process. In this chapter, we describe a translation of $\mathcal{AL}$ into A-Prolog rules.

We start by describing the encoding of action descriptions of $\mathcal{AL}$ into A-Prolog, and state its properties. Notice that our encoding applies to ground as well as *non-ground* action description. In the similar approaches from the literature (e.g. [3], [46]), the encoding is defined only for ground action descriptions, and non-ground action descriptions are viewed as abbreviations of their ground instances. By providing a direct encoding of non-ground action descriptions, we preserve the connection among the ground instances of non-ground rules, which is important in learning tasks.

After describing the encoding of action descriptions, we show the encoding of domain descriptions. Finally, we explain how the encoding of domain descriptions can be used to compute the entailment relation defined in Section 2.3.5.

To simplify the encoding, we assume that the action descriptions to be translated follow some syntactic restrictions. We say that action descriptions that follow the restrictions are in *normal form.* We will show later that this simplification does not limit the expressive power of the language.

4.1 Normal Form of $\mathcal{AL}$

To describe the notion of normal form, we need to introduce some terminology. Given a token, $m(t_1, \ldots, t_n)$, the tuple $\langle t_1, \ldots, t_n \rangle$ is called *parameter list* of the token, and is denoted by $\varpi(m(t_1, \ldots, t_n))$ (if $n = 0$, the parameter list is the empty sequence $\langle\rangle$). Notice that, since the action description is possibly non-ground, $\varpi(m(t_1, \ldots, t_n))$ may contain variables. To simplify notation, we extend the notion of parameter list to the special symbol ϵ (see Section 2.3.2), so that $\varpi(\epsilon) = \langle\rangle$. We say that a token is *unbound* if its parameter list consists only of variables. The *parameter list* of a compound action, $a = \{a_1, \dots, a_n\}$, is defined as the concatenation of the parameter lists of the elementary actions $a_1, \dots, a_n$ (the parameter list of an empty compound action is $\langle\rangle$). The *parameter list* of a law w (denoted by $\varpi(w)$) is defined as:

$$\varpi(w) = \varpi(head(w)) \circ \varpi(trigger(w)) \circ \varpi(body_l(w))$$

where $\circ$ is denotes concatenation of tuples (notice that $body_r(w)$ is *not* part of the parameter list of w).

A law, w, is in *normal form* if:

1. every fluent literal and action that occurs in w is unbound;
2. $\varpi(w)$ contains at most one occurrence of each variable;
3. for every static, r_i, from the body of w, every element of $\varpi(r_i)$ either occurs in $\varpi(w)$ or is a constant.

An action description is in *normal form* if all its laws are in normal form. Action descriptions in normal form have the following important property:

Theorem 4.1.1. *For every action description in normal form, AD, and every law $w \in AD$, $\varpi(w)$ consists only of variables.*

Proof. *By definition, the parameter list of w is the concatenation of the parameter lists of its actions and fluent literals. Since AD is in normal form, the actions and fluent literals of w are unbound, i.e. their parameter lists consist only of variables. Hence the conclusion.*

$\diamond$

The next theorem proves that restricting the language to action descriptions in normal form does not reduce its expressive power. The theorem is based on the concept of *equivalence* between action descriptions: we say that two action descriptions, AD_1 and AD_2, are *equivalent* if $trans(AD_1) = trans(AD_2)$.

Theorem 4.1.2. *For every action description, there exists an equivalent action description in normal form.*

Proof. *Let AD_1 be an action description that is not in normal form, and w a law, not in normal form, from AD_1. Since w is not in normal form, it must contain at least one of the following:*

1. *a fluent literal or action with at least one constant in its parameter list;*

2. *a variable that occurs more than once, not considering $body_r(w)$;*

3. *a variable that occurs in $body_r(w)$ but does not occur in $\varpi(w)$.*

Now, let us consider the ground instantiation of AD_1, AD_1^* and the set of laws corresponding to the grounding of w_1, w_1^*. Notice that, because of the way grounding is defined, $trans(AD_1) = trans(AD_1^*)$.

It easy to see that the only reason for a ground instance of w not to be in normal form is because some of its fluent literals or actions have some constants in their parameter lists.

Therefore, let us construct AD_2 from AD_1^* as follows. We begin by adding the definition of static eq, denoting the identity relation, to AD_2.[1] *Next, we replace each law, w', of AD_1^* with a law, w_2 obtained by substituting, in the fluent literals and actions of w', each constant c_i with a variable, x_i, so that each variable occurs only once in the parameter list of w_2 (new symbols for variables are added to the signature of AD_2 as needed). Finally, for every variable x_i introduced above, we add to the body of w_2 a static $eq(x_i, c_i)$.*

Clearly, AD_2 still satisfies requirements 2 and 3 of the definition of normal form. Moreover, by construction, all the laws of AD_2 satisfy requirement 1. Hence, AD_2 is in normal form. Moreover, from the fact that $trans(AD_1) = trans(AD_1^*)$, it is not difficult to conclude that $trans(AD_1) = trans(AD_2)$.

◇

[1]For simplicity, we assume that eq does not occur in the signature of AD_1^*.

In the rest of this chapter, we restrict our attention to laws and action descriptions in normal form.

4.2 Encoding of Action Descriptions

In this section, we define a mapping, α, from action descriptions of $\mathcal{AL}$ into sets of rules of A-Prolog. The mapping is initially defined for the elements of the action signature, then for single laws, and finally extended to the entire action description.

We omit an explicit specification of the signature of the A-Prolog program obtained by α. Rather, when writing A-Prolog rules, we will follow the usual convention that words starting with an uppercase letter denote variables, while words starting with a lower case letter denote constants, function symbols, or predicate symbols, depending on the context in which they are used.

We will use the following notation and conventions. Let AD be the action description being translated, and Σ be its signature. For every law, w, of AD we denote the number of elements in $\varpi(w)$ by $|\varpi(w)|$. Finally, given a tuple of terms $\varpi = \langle t_1, \ldots, t_n \rangle$, the position of each t_i in ϖ is denoted by $pos_{\varpi}(t_i)$.

Every token, $y = m(t_1, \ldots, t_n)$, is mapped by α into an A-Prolog term $m(t'_1, \ldots, t'_n)$ such that, for every t'_i:

- if t_i is a constant, $t'_i = t_i$;
- otherwise (recall that t_i is either a constant or a variable), t'_i is the A-Prolog variable X_k, where $k = pos_{\varpi(y)}(t_i)$.

Example 4.2.1. *Consider static $l = \neg g(c_1, x, c_2, y)$, where c_1,$c_2 \in const(\Sigma)$ and x, $y \in var(\Sigma)$. Since its parameter list is $\{c_1, x, c_2, y\}$, $pos_{\varpi(l)}(x) = 2$ and $pos_{\varpi(l)}(y) = 4$. This causes variable x to be mapped into X_2 and y into X_4. Hence, the mapping of l is:*

$$\neg g(c_1, X_2, c_2, X_4).$$

The mapping is extended to compound actions in the usual way: if $a = \{a_1, \ldots, a_n\}$ is a compound action, $\alpha(a) = \{\alpha(a_1), \ldots, \alpha(a_n)\}$.

In the rest of this section, tokens and compound actions will be written in **boldface** to denote the terms obtained by mapping them into A-Prolog. For example, if f is fluent, $\mathbf{f}$ denotes $\alpha(f)$.

Let w be a law of AD, m be an arbitrary token that occurs in w, and $t_1, \ldots, t_{|\varpi(m)|}$ denote the elements of $\varpi(m)$. We define $\Gamma_w(m)$ as the set of A-Prolog atoms:

$$\Gamma_w(m) = \{par(j, P, X_i) \mid 1 \leq i \leq |\varpi(m)| \wedge t_i \in var(\Sigma) \wedge j = pos_{\varpi(w)}(t_i)\}$$

Intuitively, $\Gamma_w(m)$ is the A-Prolog encoding of the association between the variables that occur in m and their position in the parameter list of w; X_i is the A-Prolog variable corresponding to t_i (if t_i is a variable), and j is its position in the parameter list of w.

Example 4.2.2. *Let $\langle u, x, y, z\rangle$ be the parameter list of law w, and $m = \neg g(x, c_1, z, c_2)$ be a static occurring in w, with $c_1, c_2 \in const(\Sigma)$. The A-Prolog variables corresponding to x and z are, respectively, X_1 and X_3. Since $pos_{\varpi(w)}(x) = 2$ and $pos_{\varpi(w)}(z) = 4$, $\Gamma_w(m)$ is:*

$$\{par(2, P, X_1), par(4, P, X_3)\}.$$

4.2.1 Translation of the Laws of $\mathcal{AL}$

Let w be a law of $\mathcal{AL}$, $\underline{w}$ be its name, and η be a mapping from the elements of $body(w)$ into the integers $1, \ldots, |body(w)|$, such that every $p \in body(w)$ is mapped to its position in the body of w (according to the order in which the preconditions are written) [2]. The translation of w into A-Prolog, $\alpha(w)$, consists of:

- A fact $dlaw(\underline{w})$, $slaw(\underline{w})$, or $impcond(\underline{w})$, if w is, respectively, a dynamic law, state constraint, or impossibility condition.

[2] Actually, any enumeration of the elements of $body(w)$ can be used. This particular mapping was chosen because it simplifies the presentation.

- The encoding of the parameter list of w (by Theorem 4.1.1, $\varpi(w)$ consists only of variables), consisting of a fact:

$$parlist(\underline{w}, pars(X_1, \ldots, X_{|\varpi(w)|})).$$

- For every $1 \leq i \leq |\varpi(w)|$, a rule:

$$par(i, pars(X_1, \ldots, X_{|\varpi(w)|}), X_i).$$

- *If w is either a dynamic law or an impossibility condition:* for every elementary action $a_e \in trigger(w)$, a rule

$$action(\underline{w}, P, \mathbf{a_e}) \leftarrow \Gamma_w(a_e).$$

- *If w is either a dynamic law or a state constraint:* a rule

$$head(\underline{w}, P, \mathbf{l}) \leftarrow \Gamma_w(l).$$

  where $l = head(w)$.

- For every AL-literal, p, from $body(w)$, a rule:

$$prec(\underline{w}, P, \eta(p), \mathbf{p}) \leftarrow \Gamma_w(p).$$

Example 4.2.3. *Consider the dynamic law, d_1, in normal form:*

$$\begin{aligned} \underline{d_1} : \{a\} \ causes \ g(x) \ if \quad & l_1(y, z), l_2(v), \\ & eq(x, y), eq(z, c_1), eq(v, c_2). \end{aligned} \tag{4.1}$$

where c_1, c_2 are constants, g, l_i's are fluent literals, a is an elementary action, and eq is a static corresponding to the identity relation. The parameter list of d_1 is $\langle x, y, z, v \rangle$.

The translation of d_1 consists of the A-Prolog rules:

$$dlaw(\underline{d_1}).$$

$$parlist(\underline{d_1}, pars(X_1, \ldots, X_4)).$$

$$par(1, pars(X_1, \ldots, X_4), X_1).$$
$$par(2, pars(X_1, \ldots, X_4), X_2).$$
$$par(3, pars(X_1, \ldots, X_4), X_3).$$
$$par(4, pars(X_1, \ldots, X_4), X_4).$$

$$action(\underline{d_1}, P, a).$$

$$head(\underline{d_1}, P, g(X_1)) \leftarrow par(1, P, X_1).$$

$$prec(\underline{d_1}, P, 1, l_1(X_1, X_2)) \leftarrow par(2, P, X_1), par(3, P, X_2).$$
$$prec(\underline{d_1}, P, 2, l_2(X_1)) \leftarrow par(4, P, X_1).$$
$$prec(\underline{d_1}, P, 3, eq(X_1, X_2)) \leftarrow par(1, P, X_1), par(2, P, X_2).$$
$$prec(\underline{d_1}, P, 4, eq(X_1, c_1)) \leftarrow par(3, P, X_1).$$
$$prec(\underline{d_1}, P, 5, eq(X_1, c_2)) \leftarrow par(4, P, X_1).$$

4.2.2 Translation of Action Descriptions

Let $AD = \langle \Sigma, L, K \rangle$ be an action description of $\mathcal{AL}$. The translation $\alpha(L)$ of L in A-Prolog is the union of the sets of rules resulting from the translation of each law.

The translation, $\alpha(K)$, of K in A-Prolog is:

$$K \cup \{is\_true(r) \leftarrow r \mid r \text{ is a static}\}.$$

Finally, the translation of AD, $\alpha(AD)$, is the set of A-Prolog rules:

$$\alpha(L) \cup \alpha(K) \cup \Pi_{prj},$$

where Π_{prj} contains the definition of relations *static* and $fliteral$ (used to denote statics and fluent literals), together with:

```
% if all the preconditions of a state constraint
% are satisfied, the conclusion must hold.
1.  $h(FL, T) \leftarrow$
                 $slaw(S), parlist(S, P),$
                 $head(S, P, FL),$
                 $all\_prec\_h(S, P, T).$

% if all the preconditions of a dynamic law are
% satisfied, the conclusion must hold.
2.  $h(FL, T + 1) \leftarrow$
                 $dlaw(D), parlist(D, P),$
                 $head(D, P, FL),$
                 $all\_prec\_h(D, P, T),$
                 $all\_actions\_o(D, P, T).$

% the body of an impossibility
% condition must never be satisfied.
3.  $\leftarrow impcond(C),$
        $parlist(C, P),$
        $all\_prec\_h(C, P, T),$
        $all\_actions\_o(C, P, T).$

% all the preconditions of law L, with parameters $P$,
% hold at $T$ if they hold starting from the 1st
% precondition.
4.  $all\_prec\_h(L, P, T) \leftarrow$
                 $all\_prec\_h(D, P, 1, T).$
```

% all the preconditions of law L, with parameters P,

% are true at T, starting from the N^{th} precondition

5. $all\_prec\_h(L, P, N, T) \leftarrow$

   not $has\_prec(L, N)$.

%

6. $all\_prec\_h(L, P, N, T) \leftarrow$

   $prec\_h(L, P, N, T)$,

   $all\_prec\_h(L, P, N + 1, T)$.

% law L has precondition number N

7. $has\_prec(L, N) \leftarrow$

   $prec(L, P, N, R)$.

% a static precondition is true if the corresponding

% static is true.

8. $prec\_h(L, P, N, T) \leftarrow$

   $static(S), prec(L, P, N, S)$,

   $is\_true(S)$.

% a fluent precondition is satisfied at T if the

% corresponding fluent literal holds at T.

9. $prec\_h(L, P, N, T) \leftarrow$

   $fliteral(FL), prec(L, P, N, FL)$,

   $h(FL, T)$.

% the trigger of law L is satisfied at T if all actions occurred.

10. $all\_actions\_o(D, P, T) \leftarrow$ not $actions\_not\_o(D, P, T)$.

% $actions\_not\_o(D, P, T)$: some action did not occur at T.

11. $actions\_not\_o(D, P, T) \leftarrow$
    $action(D, P, A)$,
    not $o(A, T)$.

% the inertia axiom: normally, things stay as
% they are.

12. $h(FL, T + 1) \leftarrow$
    $complement(FL, NegFL)$,
    $h(FL, T1)$,
    not $h(NegFL, T2)$.

% FL and its complement cannot both hold at T.

13. $\leftarrow complement(FL, NegFL)$,
    $h(FL, T), h(NegFL, T)$.

% dynamic laws, state constraints and impossibility
% conditions are laws.

14. $law(W) \leftarrow dlaw(W)$.
15. $law(W) \leftarrow slaw(W)$.
16. $law(W) \leftarrow impcond(W)$.

In the following discussion, given a state, σ, a ground compound action, a, and a non-negative integer, t, $h(\sigma, t)$ denotes $\{h(l, t) \mid l \in \sigma\}$, and $o(a, t)$ denotes $\{o(a_e, t) \mid a_e \in a\}$.

Before we discuss the correctness of the encoding, it is useful to establish a link between the encoding from $\mathcal{AL}$ to A-Prolog presented here and the one from [3]. Let us start by summarizing the relevant notions from [3] (notice that the terminology was slightly changed to avoid confusion).

The mapping θ, from action descriptions of $\mathcal{AL}$ into programs of A-Prolog (corresponding to the α mapping from [3]) is defined for action descriptions satisfying the following restrictions:

- no statics are allowed;
- in dynamic laws and impossibility conditions only singleton compound actions are allowed.

An action description satisfying the above conditions is called θ-compatible. The definition of θ for laws from θ-compatible action descriptions is as follows:

1. Each dynamic law is mapped into a collection of atoms:

$$d\_law(d), head(d, l_0), action(d, a_e),$$
$$prec(d, 1, l_1), \ldots, prec(d, m, l_m), prec(d, m+1, nil),$$

   where l_i's are its fluent literals and a_e its action[3], nil is a constant that does not occur in the action description, and d is the term $\underline{w}(\bar{x})$ where $\underline{w}$ is the name of the law and $\bar{x}$ is the list of variables from the action and the fluent literals of the law.

2. Each state constraint is mapped into a collection of atoms:

$$s\_law(d), head(d, l_0),$$
$$prec(d, 1, l_1), \ldots, prec(d, m, l_m), prec(d, m+1, nil).$$

3. Each impossibility condition is mapped into the rule:

$$\begin{aligned} \leftarrow \quad & h(l_1, T), \ldots, h(l_m, T), \\ & o(a, T). \end{aligned}$$

Given an action description $AD = \langle \Sigma, L, K \rangle$, by $\theta(L)$ we denote the union of the sets of rules resulting from the translation of each law from L.

Finally, $\theta(AD)$ is defined as:

$$\theta(L) \cup \Pi_\theta,$$

[3]The variables occurring in the law are suitably mapped into variables of A-Prolog.

where Π_θ is:

$$
\Pi \left\{
\begin{array}{llll}
1. & h(L,T) & \leftarrow & s\_law(D), \\
 & & & head(D,L), \\
 & & & prec\_h(D,T). \\
2. & h(L,T') & \leftarrow & d\_law(D), \\
 & & & head(D,L), \\
 & & & action(D,A), \\
 & & & o(A,T), \\
 & & & prec\_h(D,T). \\
3. & all\_h(D,N,T) & \leftarrow & prec(D,N,nil). \\
4. & all\_h(D,N,T) & \leftarrow & prec(D,N,P), \\
 & & & h(P,T), \\
 & & & all\_h(D,N',T). \\
5. & prec\_h(D,T) & \leftarrow & all\_h(D,1,T). \\
6. & h(L,T') & \leftarrow & h(L,T), \\
 & & & not\ h(\overline{L},T'). \\
7. & & \leftarrow & h(L,T), h(\overline{L},T).
\end{array}
\right.
$$

Given an action description AD, state σ and compound action a, we denote by $\sigma'_\alpha(AD,\sigma,a)$ the collection of maximal sets of fluent literals σ' such that, for some answer set A of $\alpha(AD) \cup h(\sigma,0) \cup o(a,0)$:

$$h(\sigma',1) \subseteq A.$$

Intuitively this is the set of successor states of σ and a under AD according to encoding α. The set $\sigma'_\theta(AD,\sigma,a)$ is defined similarly.

The following lemma links the α encoding and the θ encoding.

Lemma 4.2.1. *For every θ-compatible action description AD, state σ and compound action a:*

$$\sigma' \in \sigma'_\alpha(AD,\sigma,a) \quad \textit{iff} \quad \sigma' \in \sigma'_\theta(AD,\sigma,a).$$

Proof. (sketch)

The difference between $\alpha(AD)$ and $\theta(AD)$ is purely syntactic. The lemma can be proven using the Splitting Lemma, following the same approach that we used for similar comparisons in Appendix D of [3].

◇

The applicability of this result can be increased by lifting the requirement that action descriptions must be θ-compatible.

To do this, we extend θ to allow arbitrary compound actions in dynamic laws and impossibility conditions. The extension is obtained by allowing sets of $action(d, a_e)$ atoms in the encoding of the laws and by performing trivial changes to Π_θ. We call θ^+ the encoding so obtained.

The restriction on the use of statics in the action description is also not difficult to lift. In fact, since the static knowledge base consists of a collection of statics, it can be shown that any action description AD with a non-empty static knowledge base can be mapped into an action description AD_θ with an empty static knowledge base by:

- turning all static predicates into fluent predicates;
- adding to AD_θ a state constraint:

  $$r(t_1, \ldots, t_n) \text{ caused}$$

  for each static that is true in the static knowledge base of AD. Notice the empty body of the law.

Let us denote the result of applying the above mapping to an arbitrary action description AD by $k(AD)$. Using encoding θ^+ and the above mapping, the previous lemma can be generalized as follows.

Lemma 4.2.2. *For every action description AD, state σ and compound action a:*

$$\sigma' \in \sigma'_\alpha(AD, \sigma, a) \quad \textit{iff} \quad \sigma' \in \sigma'_{\theta+}(k(AD), \sigma, a).$$

Proof. *The proof of the statement is similar to that of Lemma 4.2.1.*

◇

Intuitively, this lemma shows that the existing theoretical results for encodings similar to θ apply also to the encoding used in this dissertation.

The following theorem establishes the correctness of the α encoding with respect to the semantics of $\mathcal{AL}$.

Theorem 4.2.1. *For every action description, AD, $\langle \sigma, a, \sigma' \rangle$ is a state transition in trans(AD) iff there exists some answer set, A, of*

$$\alpha(AD) \cup h(\sigma, 0) \cup o(a, 0)$$

such that $h(\sigma', 1) \subseteq A$.

Proof. *Follows from Lemma 4.2.2 and the similar results from Section D.1 of [3].*

◇

Finally, the next theorem proves that the encoding into A-Prolog can be also used to compute paths in the transition diagram.

Theorem 4.2.2. *For every action description, AD, $\pi = \langle \sigma_0, a_0, \ldots, \sigma_n \rangle$ is a path in trans(AD) iff there exists some answer set, A, of*

$$\alpha(AD) \cup h(\sigma_0, 0) \cup o(a_0, 0) \cup o(a_1, 1) \cup \ldots \cup o(a_{n-1}, n-1)$$

such that, for every $1 \leq i \leq n$, $h(\sigma_i, i) \subseteq A$.

Proof. *Follows from Lemma 4.2.2 and the similar results from Section D.1 of [3].*

◇

4.3 Encoding of Domain Descriptions

In this section, we extend mapping α to domain descriptions of $\mathcal{AL}$. Let $D = \langle AD, H^{cT} \rangle$ be a domain description. Its mapping into A-Prolog is defined as:

$$\alpha(\langle AD, H^{cT} \rangle) = \alpha(AD) \cup H^{cT} \cup \Pi_{ra}$$

where Π_{ra} consists of the following *Reality Axioms*:

% If a fluent literal is observed to be true at the initial step, it is
% true in the initial state of every model of D.
$h(L, 0) \leftarrow obs(L, 0).$

% If an action, A, was observed at step T, A occurs at T in
% every model of D.
$o(A, T) \leftarrow hpd(A, T).$

% It is impossible for a state of a model of D not to match the
% observations.
$\leftarrow obs(L, T), \text{not } h(L, T).$

The next theorem proves the correctness of the encoding of domain descriptions.

Theorem 4.3.1. *For every domain description, $D = \langle AD, H^{cT}\rangle$, such that the initial situation of H^{cT} is* complete *(i.e. for any fluent f of AD, H^{cT} contains either $obs(f, 0)$ or $obs(\neg f, 0)$), M is a model of H^{cT} iff there exists some answer set, A, of $\alpha(D)$ such that, for every $0 \leq s \leq cT$:*

$$\sigma(\pi, s) = \{l \mid h(l, s) \in A\}, \text{ and}$$

$$act(\pi, s) = \{a \mid o(a, s) \in A\}.$$

Proof. *The statement follows from Lemma 4.2.2 above and Theorem 1 from [3].*

$\diamond$

(The theorem is similar to the result from [77] which deals with a different language and uses the definitions from [50].)

The next theorem links consistency of recorded histories with the consistency of $\alpha(D)$.

Theorem 4.3.2. *For every domain description,* $D = \langle AD, H^{cT} \rangle$*,* H^{cT} *is consistent iff* $\alpha(D)$ *is consistent.*

Proof. *The conclusion follows from the application of Lemma 4.2.2, followed by Corollary 1 from [3].*

◇

4.4 Computing the Entailment Relation

Given a domain description, $D = \langle AD, H^{cT} \rangle$, checking whether H^{cT} entails a query can be reduced to membership check on the answer sets of $\alpha(D)$.

Proposition 4.4.1. *For every domain description,* $D = \langle AD, H^{cT} \rangle$ *and step* $0 \leq s \leq cT$*,* H^{cT} *entails* $h(l, s)$ *iff* $h(l, s)$ *belongs to all the answer sets of* $\alpha(D)$*.*

Proposition 4.4.2. *For every domain description,* $D = \langle AD, H^{cT} \rangle$*,* H^{cT} *entails* $h\_after(l, \langle a_1, \ldots, a_k \rangle)$ *iff* $h(l, cT + k + 1)$ *belongs to all the answer sets of* $\alpha(D) \cup \{o(a, cT + i) \mid a \in a_i\}$*.*

The proofs of these propositions can be derived from Proposition 13.6.1 of [9].

CHAPTER V
SUFFICIENT CONDITIONS FOR DETERMINISM OF ACTION DESCRIPTIONS

It is often important to know whether an action description is deterministic. In this chapter, we introduce a sufficient condition for the determinism of action descriptions that can be efficiently checked (the *necessary and sufficient* condition was shown to be coNP-hard in [78]), and show how the condition can be verified using A-Prolog.

5.1 Definition of the Condition

In the following definitions, we focus on ground action descriptions not containing statics. (Non-ground action descriptions are viewed as abbreviations of their ground instances. Ground action descriptions containing statics can be simplified into static-free ones by removing the laws whose statics do not hold, and by removing the remaining statics from the other laws.)

Definition 5.1.1 (Dependency Graph). *The* dependency graph, $dep_C(AD)$, of action description AD with respect to a consistent set of ground fluent literals C, *is a directed graph whose nodes correspond to the ground fluent literals from the signature of AD, and whose arcs, labeled by elements of* $\{1, +\}$, *are defined as follows:*

- *for every ground instance, w, of a state constraint (2.6) such that $body(w)$ is a singleton $\{l\}$, $dep_C(AD)$ contains a 1-arc $\langle head(w), 1, l\rangle$;*
- *for every ground instance, w, of a state constraint (2.6) such that $|body(w)| > 1$, and for every $l_i \in body(w)$ such that $body(w) \setminus \{l_i\} \subseteq C$, $dep_C(AD)$ contains a +-arc $\langle head(w), +, l_i\rangle$.*

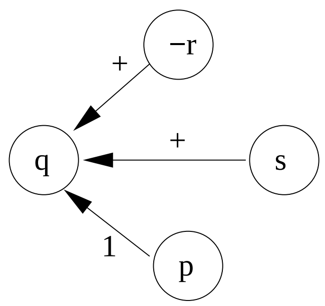
Example 5.1.1. *Consider action description:*

$$q \ \textit{if} \ \neg r, s.$$

$$q \ \textit{if} \ p.$$

$$a \ \textit{causes} \ p.$$

Its dependency graph w.r.t. $\{\neg r, s, p\}$ *is: (here and in the rest of the chapter we omit nodes that have no arcs associated to them)*



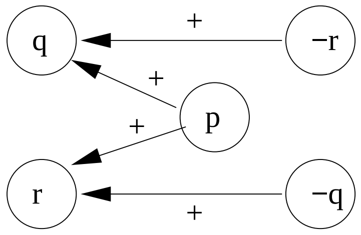
Now consider action description:

$$q \ \textit{if} \ \neg r, p.$$

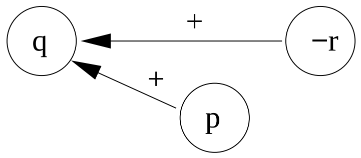
$$r \ \textit{if} \ \neg q, p.$$

$$a \ \textit{causes} \ p.$$

Its dependency graph w.r.t. $\{q, \neg q, r, \neg r, p\}$ *is:*



while its dependency graph w.r.t. $\{p, \neg r\}$ *is:*



Definition 5.1.2 (Dependency Path). *A* dependency path *w.r.t. C in the dependency graph, $dep_C(AD)$, is a sequence*

$$\pi = \langle l_1, t_1, l_2, t_2, \ldots, t_{k-1}, l_k \rangle$$

such that $k > 1$, and for every $1 \leq i < k$, $dep_C(AD)$ contains an arc $\langle l_i, t_i, l_{i+1} \rangle$.

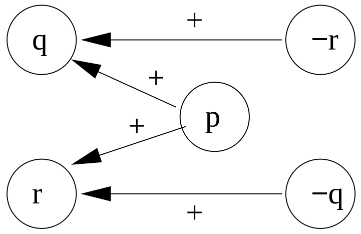
Example 5.1.2. *Consider again the following action description*

$$q \textit{ if } \neg r, p.$$

$$r \textit{ if } \neg q, p.$$

$$a \textit{ causes } p.$$

and its dependency graph w.r.t. $\{q, \neg q, r, \neg r, p\}$,



Dependency paths in this dependency graph are, for example, $\langle q, \neg r \rangle$, $\langle r, \neg q \rangle$, $\langle r, p \rangle$. On the other hand, $\langle r, p, \neg q \rangle$ is not a dependency path because there is no arc from $\neg q$ to p.

Given a dependency path π in $dep_C(AD)$, we denote its first node by π^s and its last node by π^e. Also, $|\pi|$ (the cardinality of π) denotes the number of nodes in π.

We say that a dependency path, π, *contains a* 1*-arc* (respectively, +-arc) if, for some $1 \leq i < |\pi|$, $\langle l_i, 1, l_{i+1} \rangle$ belongs to π (respectively, $\langle l_i, +, l_{i+1} \rangle$ belongs π). Paths that contain at least a +-arc are called *conditional.*

To simplify notation, we will omit the arc labels from arcs and paths when possible (e.g. we denote a path by $\langle l_1, l_2, \ldots, l_k \rangle$).

To prove the main theorems of this chapter, we will need the following definitions.

Definition 5.1.3 (Neg-Seq). *A* dependency sequence through negation *(in short,* neg-seq*) in $dep_C(AD)$ is a non-empty sequence, $\nu = \langle \pi_1, \ldots, \pi_k \rangle$, of dependency paths from $dep_C(AD)$ such that, for every $1 \leq i < k$:*

$$\pi^s_{i+1} = \overline{\pi^e_i}.$$

($\overline{\pi^e_i}$ denotes the complement of π^e_i.)

If, for every *$1 \leq i \leq k$, π_i is conditional, then we say that ν is a* conditional neg-seq*.*

Given a neg-seq, ν, in $dep_C(AD)$, we say that $dep_C(AD)$ *contains* ν.

Example 5.1.3. *In the dependency graph of Example 5.1.2, a neg-seq is*

$$\langle \langle q, \neg r \rangle, \langle r, \neg q \rangle \rangle.$$

Definition 5.1.4 (Neg-Loop). *A* dependency loop through negation *(or* neg-loop*) is a neg-seq, $\nu = \langle \pi_1, \ldots, \pi_k \rangle$, such that $\pi^s_1 = \overline{\pi^e_k}$.*

If every π_i is conditional, then we say that ν is a conditional neg-loop*.*

Example 5.1.4. *The neg-seq shown in Example 5.1.3, $\langle \langle q, \neg r \rangle, \langle r, \neg q \rangle \rangle$, is also a neg-loop.*

Definition 5.1.5 (Safe Dependency Graph). *A dependency graph, $dep_C(AD)$, is* safe *if it does not contain any conditional neg-loop.*

Example 5.1.5. *The dependency graph of Example 5.1.2 is* not *safe, as it contains a neg-loop.*

The following proposition extends results from [8, 9].

Proposition 5.1.1. *Given an arbitrary action description, AD, if, for every set of ground fluent literals, C, from the signature of AD, $dep_C(AD)$ is safe, then AD is deterministic.*

The proof of the proposition is similar to that of Theorem 5.1.1 below.

For practical applications, testing that $dep_C(AD)$ is safe for *every* C may be too complex. In this case, it may be useful to have a stronger condition, based on a notion of dependency graph that does not depend on a particular set of ground fluent literals.

Definition 5.1.6 (Simplified Dependency Graph). *The* simplified dependency graph*, $dep(AD)$, of an action description, AD, is a directed graph whose nodes are the ground fluent literals from the signature of AD, and whose arcs, labeled by elements of* $\{1, +\}$*, are defined as follows:*

- *for every ground instance, w, of a state constraint (2.6) such that $body(w)$ is a singleton $\{l\}$, $dep(AD)$ contains a* 1*-arc $\langle head(w), 1, l\rangle$;*
- *for every ground instance, w, of a state constraint (2.6) such that $|body(w)| > 1$, and for every $l_i \in body(w)$, $dep(AD)$ contains a* +*-arc $\langle head(w), +, l_i\rangle$.*

Example 5.1.6. *Consider again the action description from Example 5.1.1:*

$$q \textit{ if } \neg r, p.$$
$$r \textit{ if } \neg q, p.$$
$$a \textit{ causes } p.$$

Its simplified dependency graph is:

$$\begin{tikzcd}
q & & \neg r \arrow[ll, "+"'] \\
& p \arrow[ul, "+"'] \arrow[dl, "+"] & \\
r & & \neg q \arrow[ll, "+"]
\end{tikzcd}$$

We extend the notions of dependency path, neg-seq, and neg-loop to simplified dependency graphs. Notice that the main difference between the definitions of $dep_C(AD)$ and $dep(AD)$ is in the fact that $dep(AD)$ does not depend on a set of fluent literals. Similarly to what we did for dependency graphs, we define the notion of safe simplified dependency graph.

Definition 5.1.7 (Safe Simplified Dependency Graph). *A simplified dependency graph, $dep(AD)$, is* safe *if it does not contain any conditional neg-loop.*

Lemma 5.1.1. *Let AD be an arbitrary* non-deterministic *action description, and $\langle \sigma_0, a, \sigma_1 \rangle$, $\langle \sigma_0, a, \sigma_2 \rangle$ be two transitions from $trans(AD)$ such that $\sigma_1 \neq \sigma_2$.*

For every fluent literal $l \in \sigma_1 \setminus \sigma_2$ such that $l \notin \sigma_0$, there exists an arc $\langle l, l' \rangle$ in $dep(AD)$ such that $l' \in \sigma_1 \setminus \sigma_2$.

Proof. *Notice that $l \notin E(a, \sigma_0)$. In fact, $E(a, \sigma_0) \subseteq \sigma_2$ by Equation 2.8, and $l \notin \sigma_2$ by hypothesis.*

Moreover, from $l \notin \sigma_0$, it follows that $l \notin \sigma_1 \cap \sigma_0$.

Hence, there exists some state constraint, w, such that:

- $l = head(w)$;
- $body(w) \subseteq \sigma_1$;
- $body(w) \not\subseteq \sigma_2$.

By Definition 5.1.6, for every $l' \in body(w)$, there exists $\langle l, l' \rangle$ in $dep(AD)$. Since $body(w) \subseteq \sigma_1$ and $body(w) \not\subseteq \sigma_2$, $l' \notin \sigma_2$ for some $l' \in body(w)$.

◇

Definition 5.1.8 (S-contained path). *Given a set, S, of fluent literals, a dependency path $\langle l_1, l_2, \ldots, l_k \rangle$ in $dep(AD)$ is an S-contained path if, for every $1 \leq i \leq k$, $l_i \in S$.*

Definition 5.1.9 (S-support of l). *Given a set, S, of fluent literals, and a fluent literal, l, the S-support of l (in short, C_l^S) is the set of all fluent literals that occur in at least one S-contained path starting from l.*

Lemma 5.1.2. *Let AD be an arbitrary* non-deterministic *action description, and $\langle \sigma_0, a, \sigma_1 \rangle$, $\langle \sigma_0, a, \sigma_2 \rangle$ be two transitions from $trans(AD)$ such that $\sigma_1 \neq \sigma_2$.*

For every $l \in \sigma_1 \setminus \sigma_2$ such that $l \notin \sigma_0$, there exists a $(\sigma_1 \setminus \sigma_2)$-contained path in $dep(AD)$ that starts from l.

Proof. *Lemma 5.1.1 guarantees the existence of an arc $\langle l, l' \rangle \in dep(AD)$ such that $l' \in \sigma_1 \setminus \sigma_2$.*

By Definition 5.1.8, $\langle l, l' \rangle$ is a $(\sigma_1 \backslash \sigma_2)$-contained path.

$\diamond$

Lemma 5.1.3. *Let AD be an arbitrary* non-deterministic *action description. For every pair of transitions $\langle \sigma_0, a, \sigma_1 \rangle$, $\langle \sigma_0, a, \sigma_2 \rangle$ from $trans(AD)$ such that $\sigma_1 \neq \sigma_2$ and for every $l \in \sigma_1 \setminus \sigma_2$, the set $\sigma_1 \setminus C_l^{\sigma_1 \backslash \sigma_2}$ is closed under the state constraints of AD.*

Proof. *Let $\delta = \sigma_1 \setminus C_l^{\sigma_1 \backslash \sigma_2}$. Proving the claim by contradiction, let us assume that there exists a state constraint,*

$$\textit{caused } g \textit{ if } g_1, \ldots, g_h$$

such that $\{g_1, \ldots, g_h\} \subseteq \delta$ but $g \notin \delta$.

Obviously, $g \in \sigma_1$. Since $g \notin \delta$, $g \in C_l^{\sigma_1 \backslash \sigma_2}$. By Definition 5.1.9, there exists a $(\sigma_1 \backslash \sigma_2)$-contained path $\langle l, \ldots, g \rangle$ in $dep(AD)$. By Definition 5.1.6, for every $1 \leq i \leq h$, $\langle l, \ldots, g, g_i \rangle$ is a dependency path.

Notice that there exists $g' \in \{g_1, \ldots, g_h\}$ such that $g' \notin \sigma_2$. (Otherwise, it would follow that $g \in \sigma_2$, which contradicts $g \in C_l^{\sigma_1 \backslash \sigma_2}$.) Hence, $g' \in \sigma_1 \setminus \sigma_2$. By Definition 5.1.8 $\langle l, \ldots, g, g' \rangle$ is $(\sigma_1 \setminus \sigma_2)$-contained. By Definition 5.1.9, $g' \in C_l^{\sigma_1 \backslash \sigma_2}$. Hence, $g' \notin \delta$, which contradicts the assumption that $\{g_1, \ldots, g_h\} \subseteq \delta$.

$\diamond$

Lemma 5.1.4. *Let AD be an arbitrary* non-deterministic *action description. For every pair of transitions $\langle \sigma_0, a, \sigma_1 \rangle$, $\langle \sigma_0, a, \sigma_2 \rangle$ from $trans(AD)$ such that $\sigma_1 \neq \sigma_2$, and for every $l \in \sigma_1 \setminus \sigma_2$ such that $l \notin \sigma_0$, there exists a $(\sigma_1 \setminus \sigma_2)$-contained path, $\langle l, l_1, \ldots, l_k \rangle$, such that $l_k \in \sigma_0$.*

Proof. *Proving by contradiction, assume that, for every $(\sigma_1 \setminus \sigma_2)$-contained path $\langle l, l_1, \ldots, l_k \rangle$, $l_i \notin \sigma_0$ for every l_i.*

Let $\delta = \sigma_1 \setminus C_l^{\sigma_1 \backslash \sigma_2}$. Since the existence of a $(\sigma_1 \backslash \sigma_2)$-contained path starting from l is guaranteed by Lemma 5.1.2, $C_l^{\sigma_1 \backslash \sigma_2}$ is not empty. Hence, $\sigma_1 \supset \delta$.

From $E(a, \sigma_0) \subseteq \sigma_1 \cap \sigma_2$ and $C_l^{\sigma_1 \setminus \sigma_2} \subseteq \sigma_1 \setminus \sigma_2$, it follows that δ contains $E(a, \sigma_0)$. The assumption that $l_i \notin \sigma_0$ for every l_i, implies that $C_l^{\sigma_1 \setminus \sigma_2} \cap \sigma_0 = \emptyset$. Therefore, δ also contains $\sigma_1 \cap \sigma_0$.

Summing up, $\delta \supseteq E(a, \sigma_0) \cup (\sigma_1 \cap \sigma_0)$, and, by Lemma 5.1.3, δ is closed under the state constraints of AD. Therefore, $\delta \supseteq Cn_Z(E(a, \sigma_0) \cup (\sigma_1 \cap \sigma_0))$. Since $\sigma_1 \supset \delta$, $\sigma_1 \neq Cn_Z(E(a, \sigma_0) \cup (\sigma_1 \cap \sigma_0))$. Contradiction.

$\diamond$

Lemma 5.1.5. *Let AD be an arbitrary* non-deterministic *action description, and $\langle \sigma_0, a, \sigma_1 \rangle$, $\langle \sigma_0, a, \sigma_2 \rangle$ be two transitions from trans(AD) such that $\sigma_1 \neq \sigma_2$.*

For every fluent literal $l \in \sigma_1 \setminus \sigma_2$ such that $l \notin \sigma_0$, there exists a conditional dependency path π in dep(AD) such that:

$$\pi^s = l \ \wedge \ \pi^e \in \sigma_1 \setminus \sigma_2 \ \wedge \ \pi^e \in \sigma_0. \tag{5.1}$$

Proof. *The existence of π satisfying (5.1) follows directly from the application of Lemma 5.1.4.*

We prove that π is conditional by contradiction. Let us assume that π is not conditional (i.e. contains only 1-arcs), and let l_i denote the i^{th} node of π (hence, $l = l_1$). Since $\langle l_i, l_{i+1} \rangle$ is a 1-arc, for every state σ, if $l_{i+1} \in \sigma$, then $l_i \in \sigma$. Because σ_0 is a state, and $\pi^e \in \sigma_0$, $l_{|\pi|-1} \in \sigma_0$. It is not difficult to prove by induction that $l_1 \in \sigma_0$. Since $l = l_1$, $l \in \sigma_0$. But $l \notin \sigma_0$ by hypothesis. Contradiction.

$\diamond$

Lemma 5.1.6. *Let AD be an arbitrary* non-deterministic *action description, and $\langle \sigma_0, a, \sigma_1 \rangle$, $\langle \sigma_0, a, \sigma_2 \rangle$ be two transitions from trans(AD) such that $\sigma_1 \neq \sigma_2$.*

For every fluent literal $l \in \sigma_1 \setminus \sigma_2$ such that $l \notin \sigma_0$, and for every positive integer k, there exists a conditional neg-seq, $\langle \pi_1, \ldots, \pi_k \rangle$, such that $\pi_1^s = l$.

Proof. *By induction on k.*

Base case: $k = 1$. The conclusion follows directly from Lemma 5.1.5.

<u>Inductive Step</u>: let us assume that the theorem holds for k, and let us prove that it holds for $k+1$.

By Lemma 5.1.5, there exists a conditional path, π_1, such that $\pi_1^s = l$, $\pi_1^e \in \sigma_1 \setminus \sigma_2$, and $\pi_1^e \in \sigma_0$.

Because $\pi_1^e \in \sigma_1 \setminus \sigma_2$, $\overline{\pi_1^e} \in \sigma_2 \setminus \sigma_1$; also, from $\pi_1^e \in \sigma_0$, it follows that $\overline{\pi_1^e} \notin \sigma_0$.

By inductive hypothesis, there exists a conditional neg-seq, $\langle \pi_2, \ldots, \pi_{k+1} \rangle$, of length k, such that $\pi_2^s = \overline{\pi_1^e}$.

By Definition 5.1.3, $\langle \pi_1, \pi_2, \ldots, \pi_{k+1} \rangle$ is a conditional neg-seq. Since its length is $k+1$, and $\pi_1^s = l$, the proof is complete.

◇

Lemma 5.1.7. *For every action description AD, and for every state σ_0 and action a such that a is executable in σ_0, if $E(a, \sigma_0) \subseteq \sigma_0$, then σ_0 is the only successor state of σ_0 under a.*

Proof. *Consider an arbitrary $\langle \sigma_0, a, \sigma_1 \rangle \in trans(AD)$, and let us prove that, under the hypotheses, $\sigma_1 = \sigma_0$.*

Recall that, by Equation 2.8, $\sigma_1 = Cn_Z(E(a, \sigma_0) \cup (\sigma_1 \cap \sigma_0))$. Obviously, $\sigma_1 \cap \sigma_0 \subseteq \sigma_0$. As $E(a, \sigma_0) \subseteq \sigma_0$ by hypothesis, $E(a, \sigma_0) \cup (\sigma_1 \cap \sigma_0) \subseteq \sigma_0$.

Since σ_0 is a state, from Definition 2.3.5 it follows that, for every $X \subseteq \sigma_0$, $Cn_Z(X) \subseteq \sigma_0$. Hence, $Cn_Z(E(a, \sigma_0) \cup (\sigma_1 \cap \sigma_0)) \subseteq \sigma_0$, which implies that $\sigma_1 \subseteq \sigma_0$. Since σ_0, σ_1 are states, $\sigma_1 = \sigma_0$.

◇

Corollary 5.1.1. *For every action description, AD, and for every state σ_0 and action a such that a is executable in σ_0, if a transition $\langle \sigma_0, a, \sigma_0 \rangle$ belongs to $trans(AD)$, then σ_0 is the only successor state of σ_0 under a.*

Proof. *By Equation 2.8, $E(a, \sigma_0) \subseteq \sigma_0$. The application of Lemma 5.1.7 concludes the proof.*

◇

Corollary 5.1.2. *Let AD be an arbitrary* non-deterministic *action description. For every pair of transitions* $\langle\sigma_0, a, \sigma_1\rangle$, $\langle\sigma_0, a, \sigma_2\rangle$ *from* $trans(AD)$ *such that* $\sigma_1 \neq \sigma_2$,

$$\sigma_1 \neq \sigma_0 \text{ and } \sigma_2 \neq \sigma_0.$$

Proof. *By contradiction. If* $\sigma_1 = \sigma_0$, *then, by Corollary 5.1.1,* $\sigma_2 = \sigma_0$. *Hence,* $\sigma_1 = \sigma_2$. *Contradiction.*

◇

Theorem 5.1.1. *For every action description, AD, if dep(AD) is safe, then AD is deterministic.*

Proof. *We prove the theorem by proving the contrapositive of the claim:*

For every action description AD, if AD is non-deterministic, then dep(AD) is not safe.

Since AD is non-deterministic, there exist two transitions, $\langle\sigma_0, a, \sigma_1\rangle \in trans(AD)$ *and* $\langle\sigma_0, a, \sigma_2\rangle \in trans(AD)$, *such that* $\sigma_1 \neq \sigma_2$. *By Corollary 5.1.2, there exists* $l \in \sigma_1 \setminus \sigma_2$ *such that* $l \notin \sigma_0$.

Let n denote the number of ground fluent literals from the signature of AD, and k' *be some positive integer such that* $k' > n$. *Lemma 5.1.6 guarantees the existence of a neg-seq,* $\langle\pi_1, \ldots, \pi_{k'}\rangle$, *such that* $\pi_1^s = l$.

Since $k' > n$, *there exist* $1 \leq i < j \leq k'$ *such that* $\pi_i^s = \pi_j^s$. *By Definition 5.1.3,* $\pi_j^s = \overline{\pi_{j-1}^e}$. *By Definition 5.1.4,* $\langle\pi_i, \pi_{i+1}, \ldots, \pi_{j-1}\rangle$ *is a conditional neg-loop.*

This proves that dep(AD) contains a conditional neg-loop. Therefore, by Definition 5.1.7, dep(AD) is not safe.

◇

Example 5.1.7. *The action description:*

$$\begin{array}{l} caused\ q\ \ if\ \neg r, p \\ caused\ r\ \ if\ \neg q, p \\ a\ \ causes\ p \end{array}$$

is non-deterministic. In fact, there are two successor states for the execution of action a *in state* $\{\neg p, \neg q, \neg r\}$*:*

$$\{p, \neg q, r\} \text{ and } \{p, q, \neg r\}.$$

Notice that the simplified dependency graph is not safe, as it contains the conditional neg-loop:

$$\langle\langle q, \neg r\rangle, \langle r, \neg q\rangle\rangle.$$

Example 5.1.8. *The action description:*

$$\begin{array}{c} \textit{caused } q \textit{ if } \neg q, p \\ a \textit{ causes } p \end{array}$$

is deterministic, and its simplified dependency graph is safe, as there are no arcs out of nodes $\neg q$ *and* p*.*

Example 5.1.9. *The action description:*

$$\begin{array}{c} \textit{caused } q \textit{ if } r, p \\ \textit{caused } r \textit{ if } q, p \\ a \textit{ causes } p \end{array}$$

is deterministic, and its simplified dependency graph is safe, as it does not contain nodes $\neg q$ *and* $\neg r$*.*

Example 5.1.10. *The action description:*

$$\begin{array}{c} \textit{caused } q \textit{ if } r, p \\ \textit{caused } q \textit{ if } \neg r, p \\ a \textit{ causes } p \end{array}$$

is deterministic, and its simplified dependency graph is safe, as there are no arcs out of nodes r *and* $\neg r$*.*

It is important to stress that the condition for determinism of action descriptions is only *sufficient*, as shown by the following example.

Example 5.1.11. *The action description:*

$$\begin{gathered}\textit{caused } q \textit{ if } \neg r, p\\ \textit{caused } r \textit{ if } \neg q, p\\ a \textit{ causes } \neg p\end{gathered}$$

is deterministic. In fact, the execution of action a in any state has the only effect of making p false, if it not already false.

However, the simplified dependency graph is not safe*, as it contains the conditional neg-loop:*

$$\langle\langle q, \neg r\rangle, \langle r, \neg q\rangle\rangle.$$

The next example shows that the condition based on dependency graphs is in some cases more accurate than the one based on simplified dependency graphs.

Example 5.1.12. *The action description:*

$$\begin{gathered}\textit{caused } q \textit{ if } \neg r, \neg p\\ \textit{caused } r \textit{ if } \neg q, p\\ a \textit{ causes } p\end{gathered}$$

is deterministic.

The simplified dependency graph is not safe*, as it contains the conditional neg-loop:*

$$\langle\langle q, \neg r\rangle, \langle r, \neg q\rangle\rangle.$$

On the other hand, for every consistent *set of ground fluent literals, the dependency graph of the action description is* safe.

Finally, the next two examples show that the length of the neg-loop influences the determinism of action descriptions.

Example 5.1.13. *The action description:*

$$\begin{gathered}\textit{caused } q \textit{ if } \neg r, \neg p\\ \textit{caused } r \textit{ if } s, p\\ \textit{caused } s \textit{ if } \neg q, \neg p\\ a \textit{ causes } p\end{gathered}$$

is non-deterministic, and its simplified dependency graph is not safe.

Example 5.1.14. *The action description:*

$$\begin{array}{l} caused\ q\ \ if\ \neg r, \neg p \\ caused\ r\ \ if\ \neg s, p \\ caused\ s\ \ if\ \neg q, \neg p \\ a\ \ causes\ p \end{array}$$

is deterministic, but its simplified dependency graph is not safe.

Although taking into account this parameter in the sufficient conditions is in principle possible, it is beyond the scope of this dissertation.

The next proposition illustrates the relationship between the two conditions.

Proposition 5.1.2. *For every action description, AD, if dep(AD) is safe, then $dep_C(AD)$ is safe for every set of ground fluent literals, C. from the signature of AD.*

In the rest of this dissertation, we will use the term *dependency graph* to denote both dependency graphs with respect to a set of group literals and simplified dependency graphs, as long as it is clear from the context to which definition we are referring.

5.2 Checking for Determinism with A-Prolog

In the previous section we have stated sufficient conditions for the determinism of action descriptions. Here we show how A-Prolog can be used to check the condition corresponding to Theorem 5.1.1.

Definition 5.2.1 (A-Prolog Encoding of dep(AD)). *The A-Prolog encoding, $dep^*(AD)$, of a dependency graph dep(AD) is defined as:*

$$\begin{aligned} dep^*(AD) = \ & \{arc(x_1, x_2) \mid \langle x_1, x_2 \rangle \in dep(AD)\} \cup \\ & \{plus\_arc(x_1, x_2) \mid \langle x_1, +, x_2 \rangle \in dep(AD)\} \end{aligned}$$

Definition 5.2.2 (Conditional Neg-Seq Generator). *The Conditional Neg-Seq Generator, $cnsgen(AD)$, for an action description AD consists of the union of $dep^*(AD)$ with the rules:*

%% path(L1,L2): there is a path between $L1$ *and* $L2$
%%
$path(L_1, L_2) \leftarrow arc(L_1, L_2).$
$path(L_1, L_2) \leftarrow arc(L_1, L_3), path(L_3, L_2).$

%% cond_path(L1,L2): there is a conditional path between $L1$ *and* $L2$
%%
$cond\_path(L_1, L_2) \leftarrow plus\_arc(L_1, L_2).$
$cond\_path(L_1, L_2) \leftarrow plus\_arc(L_1, L_3), path(L_3, L_2).$
$cond\_path(L_1, L_2) \leftarrow arc(L_1, L_3), cond\_path(L_3, L_2).$

%% cond_neg_seq(L1,L2): there is a conditional neg-seq between $L1$ *and* $L2$
%%
$cond\_neg\_seq(L_1, L_2) \leftarrow cond\_path(L_1, L_2).$
$cond\_neg\_seq(L_1, L_2) \leftarrow cond\_path(L_1, L_3), cond\_neg\_seq(\overline{L_3}, L_2).$

(As usual, $\overline{L_3}$ denotes the complement of fluent literal L_3.)

Definition 5.2.3 (Safety Tester). *The Safety Tester, $safe(AD)$, for an action description AD is defined as:*

$$safe(AD) = cnsgen(AD) \cup \{\leftarrow cond\_neg\_seq(L, \overline{L}).\}.$$

To prove the correctness of the results produced by $safe(AD)$, we need the following definitions and lemmas.

Lemma 5.2.1. *For every action description, AD, $cnsgen(AD)$ has at most one answer set.*

Proof. *Straightforward, since $cnsgen(AD)$ contains neither default negation nor disjunction.*

$\diamond$

When an A-Prolog program, Π, has a unique answer set, we denote it by $ans(\Pi)$.

Lemma 5.2.2. *For every action description, AD, if $\langle l_1, \ldots, l_k \rangle \in dep(AD)$, then $path(l_1, l_k) \in ans(cnsgen(AD))$.*

Proof. *By induction on the number of nodes in the path.*

Base case: $k = 2$. If $\langle l_1, l_2 \rangle \in dep(AD)$, then, by Definition 5.2.1, a fact $arc(l_1, l_2)$ occurs in $dep^*(AD)$. Since $ans(cnsgen(AD))$ is closed under the rules of $cnsgen(AD)$, $arc(l_1, l_2) \in ans(cnsgen(AD))$.

Inductive step: assume the theorem holds for k, and prove it for $k+1$.

By Definition 5.1.2, $\langle l_1, l_2 \rangle \in dep(AD)$ and $\langle l_2, \ldots, l_k \rangle \in dep(AD)$. By Definition 5.2.1, $arc(l_1, l_2) \in dep^*(AD)$. By inductive hypothesis, $path(l_2, l_k) \in ans(cnsgen(AD))$. By closure of $ans(cnsgen(AD))$ under the laws of $cnsgen(AD)$, $path(l_1, l_k) \in ans(cnsgen(AD))$.

$\diamond$

Lemma 5.2.3. *For every action description, AD, if $\langle l_1, \ldots, l_k \rangle \in dep(AD)$ is conditional, then $cond\_path(l_1, l_k) \in ans(cnsgen(AD))$.*

Proof. *By induction on the number of nodes in the path.*

Base case: $k = 2$. If $\langle l_1, l_2 \rangle \in dep(AD)$ is conditional, then, by Definition 5.2.1, a fact $plus\_arc(l_1, l_2)$ occurs in $dep^*(AD)$. Since $ans(cnsgen(AD))$ is closed under the rules of $cnsgen(AD)$, $arc(l_1, l_2) \in ans(cnsgen(AD))$.

Inductive step: assume the theorem holds for k, and prove it for $k+1$.

By Definition 5.1.2, $\langle l_1, l_2 \rangle \in dep(AD)$ and $\langle l_2, \ldots, l_k \rangle \in dep(AD)$. Since $\langle l_1, \ldots, l_k \rangle$ is conditional, either $\langle l_1, l_2 \rangle$ is conditional, or $\langle l_2, \ldots, l_k \rangle$ is conditional.

In the first case, by Definition 5.2.1, $plus\_arc(l_1, l_2) \in dep^*(AD)$. By Lemma 5.2.2, $path(l_2, l_k) \in ans(cnsgen(AD))$. By closure of $ans(cnsgen(AD))$ under the laws of $cnsgen(AD)$, $cond\_path(l_1, l_k) \in ans(cnsgen(AD))$.

In the second case, by Definition 5.2.1, $arc(l_1, l_2) \in dep^*(AD)$. By inductive hypothesis, $cond\_path(l_2, l_k) \in ans(cnsgen(AD))$. By closure of $ans(cnsgen(AD))$ under the laws of $cnsgen(AD)$, $path(l_1, l_k) \in ans(cnsgen(AD))$.

◇

Lemma 5.2.4. *For every action description, AD, if $dep(AD)$ contains a conditional neg-seq $\langle \pi_1, \dots, \pi_k \rangle$, then $cond\_neg\_seq(\pi_1^s, \pi_k^e) \in ans(cnsgen(AD))$.*

Proof. *By induction on the number of nodes in the path.*

Base case: $k = 1$. By Definition 5.1.3, π_1 is conditional. By Lemma 5.2.3, $cond\_path(\pi_1^s, \pi_1^e) \in ans(cnsgen(AD))$. Since $ans(cnsgen(AD))$ is closed under the rules of $cnsgen(AD)$, $cond\_neg\_seq(l_1, l_2) \in ans(cnsgen(AD))$.

Inductive step: assume the theorem holds for k, and prove it for $k + 1$.

By Definition 5.1.3, all π_i's are conditional paths. By Lemma 5.2.3, $cond\_path(\pi_1^s, \pi_1^e) \in ans(cnsgen(AD))$. By the inductive hypothesis, $cond\_neg\_seq(\pi_2^s, \pi_k^e) \in ans(cnsgen(AD))$. Notice that, by Definition 5.1.3, $\pi_2^s = \overline{\pi_1^e}$. Hence, by closure of $ans(cnsgen(AD))$ under the rules of $cnsgen(AD)$, $cond\_neg\_seq(l_1, l_k) \in ans(cnsgen(AD))$.

◇

Definition 5.2.4 (Recursive Definition). *Given an arbitrary relation r, a* recursive definition of r *is the set of rules:*

$$
\begin{aligned}
&r(X) \leftarrow \Gamma_1(X).\\
&r(X) \leftarrow \Gamma_2(X).\\
&\dots\\
&r(X) \leftarrow \Gamma_n(X).\\
&r(X_1) \leftarrow \Gamma'(X_1, X_2), r(X_2).
\end{aligned}
$$

where $\Gamma_i(X)$'s and $\Gamma'(X_1, X_2)$ are conjunctions of literals (possibly preceded by default negation).

In the discussion that follows, we say that a conjunction of literals (possibly preceded by default negation) Γ is *satisfied* by a set of literals M (written $M \models \Gamma$) if every literal from Γ that is out of the scope of default negation, belongs to M, and every literal from Γ that is in the scope of default negation, does not belong to M.

Lemma 5.2.5. *For every relation r, every A-Prolog program Π containing a recursive definition of r, and every set P of ground terms from the signature of Π, if there exists an answer set, M, of Π such that:*

1. *for every x and Γ_i, if $M \models \Gamma_i(x)$, then $x \in P$;*

2. *for every x_1,x_2, if $M \models \Gamma'(x_1, x_2)$ and $x_2 \in P$, then $x_1 \in P$;*

then, for every x such that $r(x) \in M$, x belongs to P.

Proof. *Notice that, by the Marek-Subrahmanian Lemma, $r(x_1) \in M$ implies one of the following:*

- *$M \models \Gamma_i(x_1)$ for some $1 \leq i \leq n$; or*

- *there exists x_2 such that $M \models \Gamma'(x_1, x_2)$ and $r(x_2) \in M$.*

If $M \models \Gamma_i(x_1)$, then the conclusion follows from hypothesis (1).

Hence, let us consider the case in which $M \models \Gamma'(x_1, x_2)$ and $r(x_2) \in M$. Again, we can apply the Marek-Subramanian Lemma to $r(x_2)$. This proves that there exists a sequence, $s = \langle x_1, x_2, \ldots, x_k \rangle$ such that:

- *for every $1 \leq i < k$, $\Gamma'(x_i, x_{i+1}) \in M$;*

- *for every $1 \leq i \leq k$, $r(x_i) \in M$.*

Notice that, for some x_j, there exists Γ_i such that $M \models \Gamma_i(x_j)$. (Otherwise, the elements of all the sequences starting from x_1 can be removed from M, and the resulting set, $M' \subset M$, is closed under the rules of Π^M, which causes contradiction).

Hence, $x_j \in P$ by hypothesis (1). From hypothesis (2) and by construction of s, $x_{j-1} \in P$. It is not difficult to prove by induction that $x_1 \in P$.

◇

The previous lemma can be seen as a form of induction on the recursive definition of relation r. In the rest of this dissertation, we will denote the application of the lemma to some relation r by the words "By induction on the (recursive) definition of relation r."

Lemma 5.2.6. *For every action description, AD, if $path(l_1, l_2) \in ans(cnsgen(AD))$, then there exists π in $dep(AD)$ such that $\pi^s = l_1$ and $\pi^e = l_2$.*

Proof. *By induction on the definition of path in $cnsgen(AD)$.*

Base case: we need to prove that

$$arc(l_1, l_2) \in ans(cnsgen(AD)) \rightarrow \exists\ \pi \in dep(AD)\ s.t.\ \pi^s = l_1 \wedge \pi^e = l_2.$$

Notice that $ans(cnsgen(AD)) \supseteq dep^*(AD)$. By Definition 5.2.1, $arc(l_1, l_2) \in dep^*(AD)$ iff $\langle l_1, l_2 \rangle \in dep(AD)$.

Inductive step: assume $arc(l_1, l_3) \in ans(cnsgen(AD))$ and $\langle l_3, \ldots, l_2 \rangle \in dep(AD)$, and prove $\langle l_1, \ldots, l_2 \rangle \in dep(AD)$.

As before, $arc(l_1, l_3) \in ans(cnsgen(AD))$ implies $\langle l_1, l_3 \rangle \in dep(AD)$. By Definition 5.1.2, $\langle l_1, l_3, \ldots, l_2 \rangle \in dep(AD)$.

◇

Lemma 5.2.7. *For every action description, AD, if $cond\_path(l_1, l_2) \in ans(cnsgen(AD))$, then there exists π in $dep(AD)$ such that $\pi^s = l_1$, $\pi^e = l_2$ and π is conditional. .*

Proof. *By induction on the definition of $cond\_path$ in $cnsgen(AD)$.*

Base case: we need to prove that:

$$\begin{aligned}&plus\_arc\,(l_1, l_2) \in ans(cnsgen(AD)) \rightarrow \\ &\qquad \exists\ \pi \in dep(AD)\ s.t.\ \pi^s = l_1 \wedge \pi^e = l_2 \wedge \pi \text{ is conditional,}\end{aligned} \tag{5.2}$$

and:

$$\begin{aligned}&\{plus\_arc\,(l_1,l_2), path(l_3,l_2)\} \subseteq ans(cnsgen(AD)) \rightarrow \\ &\qquad \exists\ \pi \in dep(AD)\ s.t.\ \pi^s = l_1 \wedge \pi^e = l_2 \wedge \pi\ is\ conditional.\end{aligned} \tag{5.3}$$

To prove claim (5.2), notice that $ans(cnsgen(AD)) \supseteq dep^*(AD)$*. By Definition 5.2.1,* $plus\_arc(l_1,l_2) \in dep^*(AD)$ *iff* $\langle l_1,+,l_2\rangle \in dep(AD)$*.*

Claim (5.3) is proven as follows. As before, $plus\_arc(l_1,l_3) \in dep^*(AD)$ *iff* $\langle l_1,+,l_3\rangle \in dep(AD)$*. By Lemma 5.2.6,* $path(l_3,l_2) \in ans(cnsgen(AD))$ *implies that* $\langle l_3,\ldots,l_2\rangle \in dep(AD)$*. Hence,* $\langle l_1,l_3,\ldots,l_2\rangle \in dep(AD)$ *is conditional.*

Inductive step: assume $arc(l_1,l_3) \in ans(cnsgen(AD))$ *and* $\langle l_3,\ldots,l_2\rangle \in dep(AD)$ *is conditional, and prove* $\langle l_1,\ldots,l_2\rangle \in dep(AD)$ *is conditional.*

As before, $arc(l_1,l_3) \in ans(cnsgen(AD))$ *implies* $\langle l_1,l_3\rangle \in dep(AD)$*. By Definition 5.1.2,* $\langle l_1,l_3,\ldots,l_2\rangle \in dep(AD)$ *is conditional.*

◇

Lemma 5.2.8. *For every action description, AD, if* $cond\_neg\_seq(l_1,l_2) \in ans(cnsgen(AD))$*, then* $dep(AD)$ *contains a conditional neg-seq* $\langle \pi_1,\ldots,\pi_2\rangle$ *such that* $\pi_1^s = l_1$ *and* $\pi_2^e = l_2$*. .*

Proof. *By induction on the definition of cond_neg_seq in cnsgen(AD).*

Base case: we need to prove:

$$\begin{aligned}&cond\_path\,(l_1,l_2) \in ans(cnsgen(AD)) \rightarrow \\ &\qquad \exists\ \langle \pi_1,\ldots,\pi_2\rangle \in dep(AD)\ s.t.\ \pi_1^s = l_1 \wedge \pi_2^e = l_2 \\ &\qquad and\ \forall\ i, \pi_i\ is\ conditional.\end{aligned}$$

If $cond\_path(l_1,l_2) \in ans(cnsgen(AD))$*, then by Lemma 5.2.7 there exists* $\pi \in dep(AD)$ *such that* $\pi^s = l_1$*,* $\pi^e = l_2$ *and* π *is conditional. Since* $\langle\pi\rangle$ *is a conditional neg-seq, the claim is proven.*

Inductive step: assume $cond\_path(l_1,l_3) \in ans(cnsgen(AD))$ *and* $dep(AD)$ *contains a conditional neg-seq* $\langle \pi_3,\ldots,\pi_2\rangle \in dep(AD)$ *such that* $\pi_3^s = \overline{l_3}$*,* $\pi_2^e = l_2$*, and prove the existence of a conditional neg-seq* $\langle \pi_1,\ \ldots,\pi_2\rangle$ *in* $dep(AD)$ *such that* $\pi_1^s = l_1$ *and* $\pi_2^s = l_2$*.*

As in the base case, from cond_path$(l_1, l_3) \in$ *ans*$($*cnsgen*$(AD))$ *and Lemma 5.2.7 it follows that* $\langle \pi_1 \rangle$ *is a conditional neg-seq in dep*(AD) *such that* $\pi_1^s = l_1$ *and* $\pi_1^e = l_3$*. By the inductive hypothesis and Definition 5.1.3,* $\langle \pi_1, \pi_3, \ldots, \pi_2 \rangle$ *is a conditional neg-seq in dep*(AD) *such that* $\pi_1^s = l_1$ *and* $\pi_2^e = l_2$*.*

◇

We can now state the main theorem of this section.

Theorem 5.2.1. *For every action description,* AD*, dep*(AD) *is safe iff* $safe(AD)$ *is consistent.*

Proof.
Left-to-Right: by Definition 5.1.7, dep(AD) *does not contain any conditional neg-loop, i.e. any conditional neg-seq from* l *to* $\bar{l}$*. By Lemma 5.2.8, cond_neg_seq*$(l, \bar{l})$ *does not belong to ans*$($*cnsgen*$(AD))$*. Hence, the body of the constraint in* $safe(AD)$ *is never satisfied.*

Right-to-Left: by the Marek-Subramanian Lemma, there exists no l *such that cond_neg_seq*$(l, \bar{l})$ *is in ans*$(safe(AD))$*. By Lemma 5.2.4, no conditional neg-seq from* l *to* $\bar{l}$ *is in dep*(AD)*. By Definition 5.1.4, dep*(AD) *does not contain any conditional neg-loop. Hence, dep*(AD) *is safe.*

◇

The following corollary states the computational complexity of checking the condition.

Corollary 5.2.1. *For every action description,* AD*, checking whether dep*(AD) *is safe is a problem of polynomial complexity.*

Proof. *From Theorem 5.2.1, we know that the test can be reduced to checking the consistency of* $safe(AD)$*, which in turn corresponds to checking whether all the answer sets of cnsgen*(AD) *satisfy the constraint* $\{\leftarrow$ *cond_neg_seq*$(L, \overline{L})\}$*. Notice that*

$cnsgen(AD)$ is stratified. Hence, its unique answer set coincides [1] with the well-founded model of the program. Such model can be computed in polynomial time. Obviously, checking whether such model satisfies the constraint can be done in polynomial time.

$\diamond$

Notice that the encoding, $dep^*(AD)$ of the dependency graph of an action description AD can be computed directly from the encoding, $\alpha(AD)$, of AD into A-Prolog. It is not difficult to show that $dep^*(AD)$ can be extracted from the answer set of the program $\alpha(AD) \cup depgen$, where $depgen$ is:

```
$arc(FL1, FL2) \leftarrow$
        $slaw(L)$,
        $parlist(L, P)$,
        $head(L, P, FL1)$,
        $prec(L, P, N, FL2)$,
        $all\_static(L, P)$.

$plus\_arc(FL1, FL2) \leftarrow$
        $slaw(L)$,
        $parlist(L, P)$,
        $head(L, P, FL1)$,
        $prec(L, P, N1, FL2)$,
        $all\_static(L, P)$,
        $prec(L, P, N2, FL3)$,
        $FL3 \; != FL2$.

$all\_static(L, P, N) \leftarrow$
        $parlist(L, P)$,
        not $has\_prec(L, N)$.

$all\_static(L, P, N1) \leftarrow$
        $parlist(L, P)$,
        $static\_true(L, P, N1)$,
        $N2 = N1 + 1$,
        $all\_static(L, P, N2)$.

$all\_static(L, P) \leftarrow$
        $parlist(L, P)$,
        $all\_static(L, P, 1)$.

$static\_true(L, P, N) \leftarrow$
        $parlist(L, P)$,
        $static(S)$,
        $prec(L, P, N, S)$,
        $is\_true(S)$.

$static\_true(L, P, N) \leftarrow$
        $parlist(L, P)$,
        $fliteral(FL)$,
        $prec(L, P, N, FL)$.
```

CHAPTER VI
REASONING ALGORITHMS

6.1 Planning

Computation of plans in our agent follows the answer set planning approach. The term answer set planning was introduced in [44] to describe a planning technique based on a translation of domain descriptions into logic programs and on reducing planning to finding the answer sets of those logic programs.

The following terminology characterizes the description of planning tasks. A *system* is a pair $SY = \langle \Sigma, Trans \rangle$, where Σ is an action signature, and $Trans$ is a transition diagram defined on Σ.

A *planning domain* is a pair $PD = \langle SY, g \rangle$, where SY is a system and g is a goal (recall that a goal is a finite set of fluent literals that the agent has to make true). A *planning problem* is a pair $\langle PD, H^{cT} \rangle$, where PD is a planning domain and H^{cT} is the recorded history.

To simplify notation, we use a pair $\langle D, g \rangle$ (where $D = \langle AD, H^{cT} \rangle$ is a domain description and g a goal) as an abbreviation of the planning problem $\langle \langle SY, g \rangle, H^{cT} \rangle$ where $SY = \langle sig(AD), trans(AD) \rangle$. In the rest of the discussion, when we talk about a fixed planning domain, we denote the corresponding action signature, domain description, action description and goal by Σ, D, AD, and g respectively.

Hence, planning for a goal g is reduced to finding a sequence $\langle a_1, \ldots, a_k \rangle$ of compound agent actions such that:

$$H^{cT} \models_{AD} h\_after(g, \langle a_1, \ldots, a_k \rangle). \tag{6.1}$$

Such sequence is called *plan*.

Shortest plans can be computed by the following simple planning component.

Algorithm PC_0

Input:

- domain description, $D = \langle AD, H^{cT} \rangle$;
- goal $g = \{l_1, \ldots, l_m\}$.

Output:

- a sequence of agent actions, $\langle a_1, \ldots, a_k \rangle$, satisfying (6.1).

Steps:

1. k := 0
2. find a sequence $\langle a_1, \ldots, a_k \rangle$ that satisfies (6.1)
3. if such a sequence exits, then return it
4. k := k + 1
5. goto 2

The computation at step 2 is reduced to computing an answer set of the program:

$$Plan_0(D, g, k) = \alpha(D) \cup AGEN(k) \cup GOALTEST(g, k)$$

where:

- k is the length of the plan that the agent is looking for;
- $AGEN(k)$ is:

  $$1\{o(A, T) : ag\_action(A)\} \leftarrow cT \leq T < cT + k.$$

  where cT is the current time step, as specified by the history H^{cT}, k is the plan length currently being considered, and ag_action is a relation that is true for agent actions. The rule informally says that each step T of an action sequence contains a non-empty compound action.

- $GOALTEST(\{l_1, \ldots, l_m\}, k)$ is:

$$\begin{aligned} goal\_achieved \leftarrow \quad & h(l_1, cT + k), \\ & \ldots, \\ & h(l_m, cT + k). \end{aligned}$$

$$\leftarrow \text{not } goal\_achieved.$$

  where l_i's are the elements of goal g. The first rule informally says that goal g has been achieved if all l_i's are expected to be true at the end of the execution of the action sequence. The second says that any action sequence computed must be a plan (i.e., it must achieve the goal).

The structure of $Plan_0$ follows the *generate (define) and test* approach described in [25, 43, 29], where $AGEN$ generates candidate plans, $\alpha(D)$ defines the effects of the actions, and $GOALTEST$ tests whether an action sequence achieves the goal.

The next theorem proves that step 2 can be reduced to computing the answer sets of $Plan_0(D, g, k)$.

Theorem 6.1.1. *For every deterministic domain description,* $D = \langle AD, H^{cT} \rangle$, such that H^{cT} contains complete information about the initial state*, for every goal, g, and for every non-negative integer, k:*

a sequence of compound agent actions, $p = \langle a_1, \ldots, a_k \rangle$ *is a plan for g w.r.t. D iff there exists an answer set, A, of* $Plan_0(D, g, k)$*, such that, for every* $1 \leq i \leq k$*,*

$$a_i = \{a \mid o(a, cT + i) \in A \land a \in action_{ag}(sig(AD))\}.$$

Proof. (sketch)
First of all, we need to prove that $Plan_0(D, g, k)$ *finds the plans of length k for g. This follows from the following observations:*

1. *For any future step t, AGEN generates all possible sets of occurrences of the agent's actions at t.*

2. *By Theorem 4.3.1 there is a one-to-one correspondence between the models of* H^{cT} *and the answer sets of* $\alpha(D)$.

3. *Because of* $GOALTEST(g, k)$*, all the answer sets of* $Plan_0(D, g, k)$ *are bound to satisfy the goal at* $cT + k$.

4. *Because of the lower bound in* $AGEN$*, at least one occurrence is generated at each step.*

In fact, (2) can be used to show that the results of the sequences of actions mentioned in (1) are correctly predicted by $Plan_0(D, g, k)$*. (3) guarantees that only sequences of actions that achieve the goal are encoded by the answer sets of* $Plan_0(D, g, k)$*. Next, the assumption that* D *is deterministic allows to conclude that the sequences of actions encoded by the answer sets are plans. Finally, from (4) it follows that the answer sets encode all the plans that are exactly* k *steps long (i.e. they terminate at* $cT + k$*).*

$\diamond$

Finally, the next theorem proves that algorithm PC_0 is sound and complete with respect to the definition of shortest plan.

Theorem 6.1.2. *For every deterministic domain description,* $D = \langle AD, H^{cT} \rangle$, such that H^{cT} contains complete information about the initial state, *and for every goal* g*:*

a sequence of compound actions, p*, is a shortest plan for* g *w.r.t.* D *iff* p *is returned by* $PC_0(D, g)$.

Proof. *Observe that the algorithm computes the answer sets of the sequence of programs* $Plan_0(D, g, 0)$, $Plan_0(D, g, 1)$, $Plan_0(D, g, 2)$, $\ldots$ *and terminates for the smallest* k *such that* $Plan_0(D, g, k)$ *is consistent.*

By theorem 6.1.1, $Plan_0(D, g, k)$ *is consistent iff at least one plan of length* k *exist. Hence, the plan returned is a shortest plan.*

$\diamond$

In this dissertation, we address the following questions:

- Does this approach scale well to medium-size, knowledge-intensive applications?
- How can we specify criteria (different from length minimization) for the selection of best plans?

To answer the first question, we have developed *USA-Advisor*, a medium-size application that computes plans in the presence of malfunctioning components in the Reaction Control System of the Space Shuttle. To address the second question, we used cr-rules and preferences of CR-Prolog. The following section gives more details on the first topic[1], while the use of CR-Prolog is discussed in Chapter VIII.

6.1.1 USA-Advisor

USA-Advisor is a decision support system for the Reaction Control System (RCS) of the Space Shuttle, designed in the context of a project aimed at demonstrating the applicability of A-Prolog, and of answer set planning in particular, to medium-size, knowledge-intensive applications. The project also demonstrated that A-Prolog allows for a modular organization of knowledge, enabling knowledge module reuse, as well as testing (and debugging) of single knowledge modules, rather than of the entire knowledge base as a whole. The techniques used in the development of USA-Advisor are general enough to allow for the modeling of many other physical systems, and for the execution of reasoning tasks other than planning, including fault detection and diagnosis.

The RCS is the system that is primarily used to perform rotational and translational movements during flight. It is a rather complex physical system, that includes 12 tanks, 44 jets, 66 valves, 33 switches, and around 40 computer commands (computer-generated signals). USA-Advisor contains a complete model of the RCS, including wiring and plumbing diagrams.

To understand the functionality of USA-Advisor, let us imagine a Shuttle's flight controller who is considering how to prepare the Shuttle for a maneuver when faced

[1]These results have been published in part in [2]

with a collection of faults present in the RCS (for example, switches and valves can be stuck in various positions, electrical circuits can malfunction in various ways, valves can be leaking, jets can be damaged, etc.). In this situation, the controller needs to find a sequence of actions to set the shuttle ready for the maneuver, i.e. that delivers propellant to an appropriate set of jets. USA-Advisor is designed to facilitate this task. The controller can use it to test if a plan, which he came up with manually, will actually be able to prepare the RCS for the desired maneuver. Most importantly, USA-Advisor can be used to automatically find such a plan.

We now give a brief introduction to the design of the system.

6.1.1.1 System's Design

USA-Advisor consists of a collection of largely independent A-Prolog modules, represented by lp-functions[2], and a graphical Java interface. The interface gives a simple way for the user to enter information about the history of the RCS, its faults, and the task to be performed. The A-Prolog modules are organized in knowledge modules and reasoning modules. Each knowledge module contains a different part of the knowledge about the domain of the RCS, while each reasoning module is responsible for performing a different reasoning task. At the moment there are two possible types of tasks:

- checking if a sequence of occurrences of actions in the history of the system satisfies a given goal, G;
- finding a plan for G of a length not exceeding some number of steps, N.

The set of A-Prolog modules that are used depends on the particular task being performed (e.g. detailed knowledge about electrical circuits is included only in presence of electrical faults). Based on the information provided by the user, the graph-

[2]By an lp-function we mean program Π of A-Prolog with input and output signatures $\sigma_i(\Pi)$ and $\sigma_o(\Pi)$ and a set $dom(\Pi)$ of sets of literals from $\sigma_i(\Pi)$ such that, for any $X \in dom(\Pi)$, $\Pi \cup X$ is consistent, i.e. has an answer set.

ical interface verifies if the input is complete, selects an appropriate combination of modules, assembles everything into an A-Prolog program, Π, and passes Π as an input to an inference engine for computing its answer sets.

The results of the reasoning task are then extracted from the answer sets of Π. The interpretation of the contents of the answer sets depends, again, on the reasoning task being performed. In our approach, the task of verifying the correctness of a sequence of actions is reduced to checking if program Π has at least an answer set. On the other hand, the planning module is designed so that there is a one-to-one correspondence between the plans and the answer sets of Π. Extraction and displaying of the results is performed by the Java interface.

The modules that compose USA-Advisor are:

- the Plumbing Module;
- the Valve Control Module (divided in "Basic Valve Control Module" and "Extended Valve Control Module");
- Circuit Theory Module;
- Planning Module.

We continue with the descriptions of the various modules.

6.1.1.2 Plumbing Module

The Plumbing Module (PM) models the plumbing system of the RCS, which consists of a collection of tanks, jets and pipe junctions connected through pipes. The flow of fluids through the pipes is controlled by valves. The system's purpose is to deliver fuel and oxidizer from tanks to the jets needed to perform a maneuver. The structure of the plumbing system is described by a directed graph, Gr, whose nodes are tanks, jets and pipe junctions, and whose arcs are labeled by valves. The possible faults of the system at this level are leaky valves, damaged jets, and valves stuck in some position.

The purpose of PM is to describe how faults and changes in the position of valves affect the pressure of tanks, jets and junctions. In particular, when fuel and oxidizer flow at the right pressure from the tanks to a properly working jet, the jet is considered ready to fire. In order for a maneuver to be started, all the jets it requires must be ready to fire. The necessary condition for a fluid to flow from a tank to a jet, and in general to any node of Gr, is that there exists a path without leaks from the tank to the node and that all valves along the path are open.

The rules of PM define a function which takes as input the structural description, Gr, of the plumbing system, its current state, including position of valves and the list of faulty components, and determines: the distribution of pressure through the nodes of Gr; which jets are ready to fire; which maneuvers are ready to be performed. In our approach, the state of the plumbing system (as well as of the electrical system shown later) consists of the set of fluents (properties of the domain whose truth depends on time) which are true in that state.

To illustrate the issues involved in the construction of PM, let us consider the definition of fluent $pressurized\_by(N, Tk)$, describing the pressure obtained on a node N by a tank Tk. Some special nodes, the helium tanks, are always pressurized. For all other nodes, the definition is recursive. It says that any node $N1$ is pressurized by a tank Tk if $N1$ is not leaking and is connected by an open valve to a node $N2$ which is pressurized by Tk.

Representation of this definition in most logic programming languages, including Prolog, is problematic, since the corresponding graph can contain cycles. The ability of A-Prolog to express and to reason with recursion allows us to use the following concise definition of pressure on non-tank nodes.

$$\begin{aligned} h(pressurized\_by(N1, Tk), T) \leftarrow & \\ & \text{not } tank\_of(N1, R), \\ & \text{not } h(leaking(N1), T), \\ & link(N2, N1, V), \\ & h(in\_state(V, open), T), \\ & h(pressurized\_by(N2, Tk), T). \end{aligned}$$

The high level of abstraction of A-Prolog is confirmed by the relatively small number of rules present in the knowledge modules of USA-Advisor. For example, the Plumbing Module consists of approximately 40 rules.

6.1.1.3 Valve Control Module

The flow of fuel and oxidizer propellants from tanks to jets is controlled by opening/closing valves along the path. The state of valves can be changed either by manipulating mechanical switches or by issuing computer commands. Switches and computer commands are connected to the valves, they control, by electrical circuits.

The action of flipping a switch Sw to some position S normally puts a valve controlled by Sw in this position. Similarly for computer commands. There are, however, three types of possible failures: switches and valves can be stuck in some position, and electrical circuits can malfunction in various ways. Substantial simplification of the VCM module is achieved by dividing it in two parts, called *basic* and *extended* VCM modules.

At the basic level, it is assumed that all electrical circuits are working properly and therefore are not included in the representation. The extended level includes information about electrical circuits and is normally used when some of the circuits are malfunctioning. In that case, flipping switches and issuing computer commands may produce results that cannot be predicted by the basic representation.

6.1.1.4 Basic Valve Control Module

At this level, the VCM deals with a set of switches, computer commands and valves, and connections among them. The input of the basic VCM consists of the initial positions and faults of switches and valves, and the sequence of actions defining the history of events. The module implements an lp-function that, given this input, returns positions of valves at the current moment of time. This output is used as input to the plumbing module. The possible faults of the system at this level are valves and switches stuck at some position(s).

The following rules show an example of the formalization of the basic VCM. The first is a dynamic causal rule stating that, if a properly working switch Sw is flipped to state S at time T, then Sw will be in this state at the next moment of time.

$$\begin{aligned} h(in\_state(Sw, S), T + 1) \leftarrow \\ & occurs(flip(Sw, S), T), \\ & \text{not } stuck(Sw). \end{aligned}$$

A static connection between switches and valves is expressed by the next rule. This static law says that, under normal conditions, if switch Sw controlling a valve V is in some state S (different from *gpc*[3]) at time T, then V is also in this state at the same time.

$$\begin{aligned} h(in\_state(V, S), T) \leftarrow \\ & controls(Sw, V), \\ & h(in\_state(Sw, S), T), \\ & neq(S, gpc), \\ & \text{not } h(ab_input(V), T), \\ & \text{not } stuck(V), \\ & \text{not } bad\_circuitry(V). \end{aligned}$$

The condition *not bad_circuitry(V)* is used to stop this rule from being applied when the circuit connecting Sw and V is not working properly. (Notice that the

[3]A switch can be in one of three positions: open, closed, or gpc. When it is in gpc, it does not affect the state of the valve.

previous dynamic rule, instead, is applied independently of the functioning conditions of the circuit, since it is related only to the switch itself.) If the switch is in a position, $S1$, different from *gpc*, and a computer command is issued to move the valve to position $S2$, then there is a conflict in case $S1 \neq S2$. This is an abnormal situation, which is expressed by fluent $ab\_input(V)$. When this fluent is true, negation as failure is used to stop the application of this rule. In fact, the final position of the valve can only be determined by using the representation of the electrical circuit that controls it. This will be discussed in the next section.

6.1.1.5 Extended Valve Control Module

The extended VCM encompasses the basic VCM and also includes information about electrical circuits, power and control buses, and the wiring connections among all the components of the system.

The lp-function defined by this module takes as input the same information accepted by the basic VCM, together with faults on power buses, control buses and electrical circuits. It returns the positions of valves at the current moment of time, exactly like the basic VCM.

Since (possibly malfunctioning) electrical circuits are part of the representation, it is necessary to compute the signals present on all wiring connections, in order to determine the positions of valves. The signals present on the circuit's wires are generated by the Circuit Theory Module (CTM), included in the extended VCM. Since this module was developed independently to address a different collection of tasks [5], its use in this system is described in a separate section.

There are two main types of valves in the RCS: solenoid and motor controlled valves. Depending on the number of input wires they have, motor controlled valves are further divided in 3 sub-types. While at the basic VCM there is no need to distinguish between these different types of valves, they must be taken into account at the extended level, since the type determines the number of input wires of the valve. In all cases, the state of a valve is normally determined by the signals present on its input wires.

For the solenoid valve, its two input wires are labeled *open* and *closed*. If the *open* wire is set to 1 and the *closed* wire is set to 0, the valve moves to state open. Similarly for the state closed. The following static law defines this behavior.

$$
\begin{aligned}
& h(in\_state(V, S1), T) \leftarrow \\
& \qquad input(W1, V), \\
& \qquad input(W2, V), \\
& \qquad input\_of\_type(W1, S1), \\
& \qquad input\_of\_type(W2, S2), \\
& \qquad h(value(W1, 1), T), \\
& \qquad h(value(W2, 0), T), \\
& \qquad neq(S1, S2), \\
& \qquad \text{not } stuck(V).
\end{aligned}
$$

The state of all other types of valves is determined in much the same way. The only difference is in the number of wires that are taken into consideration.

The output signals of switches, valves, power buses and control buses are also defined by means of static causal laws.

At this level, the representation of a switch is extended by a collection of input and output wires. Each input wire is associated to one and only one output wire, and every input/output pair is linked to a position of the switch. When a switch is in position S, an electrical connection is established between input Wi and output Wo of the pair(s) corresponding to S. Therefore, the signal present on Wi is transferred to Wo, as expressed by the following rule.

$$
\begin{aligned}
& h(value(Wo, X), T) \leftarrow \\
& \qquad h(in\_state(Sw, S), T), \\
& \qquad connects(S, Sw, Wi, Wo), \\
& \qquad h(value(Wi, X), T).
\end{aligned}
$$

The VCM consists of 36 rules, not including the rules of the Circuit Theory Module.

6.1.1.6 Circuit Theory Module

The Circuit Theory Module (CTM) is a general description of components of electrical circuits. It can be used as a stand-alone application for simulation, computation of the topological delay of a circuit, detection of glitches, and abduction of the circuit's inputs given the desired output.

The CTM is employed in this system to model the electrical circuits of the RCS, which are formed by digital gates and other electrical components, connected by wires. Here, we refer to both types of components as *gates*. The structure of an electrical circuit is represented by a directed graph E where gates are nodes and wires are arcs. A gate can possibly have a propagation delay D associated with it, where D is a natural number (zero indicates no delay). All signals present in the circuit are expressed in 3-valued logic (0, 1, u). If no value is present on a wire at a certain moment of time T then it is said to be unknown (u) at T.

This module describes the normal and faulty behavior of electrical circuits with possible propagation delays and 3-valued logic.

In CTM, *input wires* of a circuit are defined as the wires coming from switches, valves, computer commands, power buses and control buses. *Output wires* are those that go to valves. The CTM is an lp-function that takes as input the description of a circuit C, the values of signals present on its input wires, the set of faults affecting its gates, and determines the values on the output wires of C at the current moment of time.

We allow for standard faults from the theory of digital circuits [40, 54]. A gate G malfunctions if its output, or at least one of its input pins, are permanently stuck on a signal value. The effect of a fault associated to a gate of the direct graph E only propagates forward.

CTM contains two sets of static rules. One of them allows for the representation of the normal behavior of gates, while the other expresses their faulty behavior. To illustrate how the normal behavior of gates is described in the CTM, let us consider the case of the Tri-State gate. This type of component has two input wires, of which one is labeled *enable*. If this wire is set to 1, the value of the other input is transferred to the output wire. Otherwise, the output is undefined. The following rule describes the normal behavior of the Tri-State gate when it is enabled.

$$\begin{aligned}
&h(value(W,X),T+D) \leftarrow \\
&\qquad delay(G,D), \\
&\qquad input(W1,G), \\
&\qquad input(W2,G), \\
&\qquad type\_of\_wire(W2,G,enable), \\
&\qquad neq(W1,W2), \\
&\qquad h(value(W1,X),T), \\
&\qquad h(value(W2,1),T), \\
&\qquad output(W,G), \\
&\qquad \text{not } is\_stuck(W,G).
\end{aligned}$$

It is interesting to discuss how faults are treated when they occur on the input wire of a gate. Let us consider the case of a gate G with an input wire stuck at value X. This wire is represented as two unconnected wires, W and $stuck\_wire(W)$, corresponding to the normal and faulty sections of the wire. The faulty part is stuck at value X, while the value of W is computed by normal rules depending upon its connection to the output of other gates. Rules for gates with faulty inputs use $stuck\_wire(W)$ as input wire. The example below is related to a Tri-State gate with the non-enable wire stuck to X.

$$\begin{aligned} h(value(W,X),T+D) \leftarrow & \\ & delay(G,D), \\ & input(stuck\_wire(W1),G), \\ & input(W2,G), \\ & type\_of\_wire(W2,G,enable), \\ & neq(W1,W2), \\ & h(value(stuck\_wire(W1),X),T), \\ & h(value(W2,1),T), \\ & output(W,G), \\ & \text{not } is\_stuck(W,G). \end{aligned}$$

Notice that condition *not is_stuck(W,G)* prevents the above rules from being applied when the output wire is stuck. Whenever an output wire is stuck at X, the corresponding rule guarantees that its signal value is always X.

The behavior of a circuit is said *normal* if all its gates are functioning correctly. If one or more gates of a circuit malfunction then the circuit is called *faulty*.

The description of faulty electrical circuit(s) is included as part of the RCS representation. However, it is not necessary to add the description of normal circuits controlling a valve(s) since the program can reason about effects of actions performed on that valve through the basic VCM. This allows for an increase in efficiency when computing models of the program.

The Circuit Theory Module contains approximately 50 rules.

6.1.1.7 Planning module

This module establishes the search criteria used by the program to find a plan, i.e. a sequence of actions that, if executed, would achieve the goal. The modular design of USA-Advisor allows for the creation of a variety of such modules.

The structure of the Planning Module (PlM) follows the generate and test approach described in [25, 43]. Since the RCS contains more than 200 actions, with rather complex effects, and may require very long plans, this standard approach needs to be substantially improved. This is done by addition of various forms of heuristic, domain-dependent information[4]. In particular, the generation part takes advantage of the fact that the RCS consists of three, largely independent, subsystems. A plan for the RCS can therefore be viewed as the composition of three separate plans that can operate in parallel. Generation is implemented using the following rule:

$$\begin{aligned} 1\{occurs(A,T) : action\_of(A,R)\}1 \leftarrow \\ & subsystem(R), \\ & \text{not } goal(T,R). \end{aligned}$$

This rule states that exactly one action for each subsystem of the RCS should occur at each moment of time, until the goal is reached for that subsystem.

In the RCS, the common task is to prepare the shuttle for a given maneuver. The goal of preparing for such a maneuver can be split into several subgoals, each setting some jets, from a particular subsystem, ready to fire. The overall goal can therefore be stated as a composition of the goals of individual subsystems containing the desired jets, as follows:

$$\begin{aligned} goal \leftarrow \\ & goal(T1, left_rcs), \\ & goal(T2, right_rcs), \\ & goal(T3, fwd_rcs). \end{aligned}$$

The plan testing phase of the search is implemented by the following constraint

$$\leftarrow \text{not } goal.$$

which eliminates the models that do not contain plans for the goal.

Splitting into subsystems allows us to improve the efficiency of the module substantially.

[4]Notice that the addition does not affect the generality of the algorithm.

The module also contains other domain-dependent as well as domain-independent heuristics. The reasons for adding such heuristics are two-fold: first, to eliminate plans which are correct but unintended, and second, to increase efficiency. A-Prolog allows for a concise representation of these heuristics as constraint rules. This can be demonstrated by means of the following examples.

Some heuristics are instances of domain-independent heuristics. They express common-sense knowledge like "under normal conditions, do not perform two different actions with the same effect." In the RCS, there are two different types of actions that can move a valve V to a state S: a) flipping to state S the switch, Sw, that controls V, or b) issuing the (specific) computer command CC capable of moving V to S. In A-Prolog we can write this heuristic as follows

$$\begin{aligned}\leftarrow\ & occurs(flip(Sw, S), T),\\ & controls(Sw, V),\\ & occurs(CC, T1),\\ & commands(CC, V, S),\\ & \text{not } bad\_circuitry(V).\end{aligned}$$

More domain-dependent rules embody common-sense knowledge of the type "do not pressurize nodes which are already pressurized." In the RCS, some nodes can be pressurized through more than one path. Clearly, performing an action in order to pressurize a node already pressurized will not invalidate a plan, but this involves an unnecessary action. The following constraint eliminates models where more than one path to pressurize a node $N2$ is open.

$$\begin{aligned}\leftarrow\ & link(N1, N2, V1),\\ & link(N1, N2, V2),\\ & neq(V1, V2),\\ & h(in\_state(V1, open), T),\\ & h(in\_state(V2, open), T),\\ & \text{not } stuck(V1, open),\\ & \text{not } stuck(V2, open).\end{aligned}$$

As mentioned before, some heuristics are crucial for the improvement of the planner's efficiency. One of them states that "a normally functioning valve connecting nodes $N1$ and $N2$ should not be open if $N1$ is not pressurized." This heuristic clearly prunes a significant number of unintended plans. It is represented by a constraint that discards all plans in which a valve V is opened before the node, preceding it, is pressurized.

$$\begin{aligned}\leftarrow\ & link(N1, N2, V),\\ & h(in\_state(V, open), T),\\ & \text{not } h(pressurized\_by(N1, Tk), T),\\ & \text{not } has\_leak(V),\\ & \text{not } stuck(V).\end{aligned}$$

The efficiency improvement offered by domain-dependent heuristics has not been studied mathematically. However, experiments showed impressive results. In the case of tasks involving a large number of faults, for example, the introduction of some of the most effective heuristics reduced the time required to find a plan from hours to seconds.

6.2 Unexpected Observations

An unexpected observation is an observation that contradicts the agent's expectations about the state of the domain. In the observe-think-act loop, unexpected observations are detected when the domain description is inconsistent; to remove the inconsistency, the agent needs to find an explanation of the discrepancy between the observations and its expectations. More precisely, unexpected observations are explained by:

- hypothesizing that the current domain description does not adequately model the domain, and
- finding how the domain description can be modified to match the observations.

In principle, unexpected observations may be due to problems in the recorded history, in the action description, or in both. In the first case, we say that the recorded history is *problematic*. In the second case, the action description is problematic. In the third case, both are problematic. We call the first task *diagnosis*, the second *learning*, and the third *mixed interpretation task*.

To simplify the study, we focus only on the first two cases. We also restrict attention to agents operating physical devices and capable of testing and repairing the device components. We call the domains corresponding to this scenario *physical systems*.

A physical system is represented by a triple $PS = \langle \Sigma, Trans, Cm \rangle$, where:

- Σ is an action signature;
- $Trans$ is a transition diagram defined on Σ;
- Cm is a set of constants from Σ, called *components* of PS.

We assume that Σ contains a fluent predicate ab. Intuitively, $ab(c)$ (where $c \in Cm$) states that component c is malfunctioning [62].

In this dissertation we make the assumption that environments are *non-intrusive*, i.e. do not normally interfere with the agent's work. This assumption is particularly important when the agent needs to perform tests to determine whether a possible explanation of unexpected observations is indeed correct.

In our formalization of the interpretation of unexpected observations, an important role is played by the *actual evolution of the system*, a sequence $W = \langle \tau_0, a_0, \tau_1, \ldots, a_{k-1}, \tau_k, a_k, \ldots \rangle$ where a_i's are compound actions and τ_i's (called *moments*) are consistent and complete sets of literals from the action signature of the system. Intuitively, W describes the sequence of steps that the system actually went through. Notice that the only direct knowledge that the agent has about W comes from the observe step in the observe-think-act loop, and by the recordings of its own actions. We denote the moment and action at step i of W by $\tau(W, i)$ and $act(W, i)$.

An *interpretation domain* is a pair $\langle PS, W\rangle$, where PS is a physical system and W is its actual evolution. An *interpretation problem* is a pair $\langle ID, H^{cT}\rangle$, where ID is an interpretation domain and H^{cT} is the recorded history.

To simplify notation, we often use a pair $\langle D, W\rangle$ (where $D = \langle AD, H^{cT}\rangle$ is a domain description) as an abbreviation of the interpretation problem $\langle\langle PS, W\rangle, H^{cT}\rangle$ where $PS = \langle sig(AD), trans(AD), Cm\rangle$ and Cm is implicitly defined by $sig(AD)$. In the rest of the discussion, when we talk about a fixed interpretation domain, we denote the corresponding action signature and components of the physical system by Σ and Cm respectively.

The introduction of the above terminology allows us to precisely characterize diagnostic, learning and mixed tasks. Given an interpretation problem $P_I = \langle D, W\rangle$ where $D = \langle AD, H^{cT}\rangle$:

- if W is a path from $trans(AD)$, and for every $0 \leq i < cT$:
  - $act(W, i) \supseteq \{a_e \mid hpd(a_e, i) \in H^{cT}\}$, and
  - $act(W, i) \setminus \{a_e \mid hpd(a_e, i) \in H^{cT}\} \subseteq action_{ex}(\Sigma)$

  then P_I is a *diagnostic problem*, and the corresponding reasoning task is a diagnostic task (the moments of W are thus states, and can be denoted by expressions of the form $\sigma(W, i)$);

- if W is not a path from $trans(AD)$ and, for every $0 \leq i < cT$,

$$act(W, i) = \{a_e \mid hpd(a_e, i) \in H^{cT}\},$$

  then P_I is a *learning problem* and the corresponding reasoning task is a learning task;

- if W is not a path from $trans(AD)$, and for every $0 \leq i < cT$:
  - $act(W, i) \supseteq \{a_e \mid hpd(a_e, i) \in H^{cT}\}$, and
  - $act(W, i) \setminus \{a_e \mid hpd(a_e, i) \in H^{cT}\} \subset action_{ex}(\Sigma)$

where for some i the first inclusion is strict, then P_I is a *mixed interpretation problem* and the corresponding reasoning task is a mixed interpretation task.

Central to all of the above reasoning tasks is the notion of *symptom*, which we now introduce.

In the rest of the discussion, O_n^m ($n \leq m$) denotes a set of statements of $\mathcal{AL}_h$, corresponding to observations made by the agent between steps n and m. Given an interpretation problem, $P_I = \langle D, W \rangle$, a pair $\langle H^n, O_n^{cT} \rangle$ (where $H^n \cup O_n^{cT} = H^{cT}$) is called a *configuration* of P_I. We say that a configuration

$$\mathcal{S} = \langle H^n, O_n^{cT} \rangle \tag{6.2}$$

is a *symptom* of the system's malfunctioning if H^n is consistent and $H^n \cup O_n^{cT}$ is not.

6.2.1 Diagnosis

In this section, we discuss how answer set programming can be used to perform diagnostic tasks.

The underlying assumption that guides the agent during diagnosis is that the environment is *observable*, i.e. the agent normally observes all of the domain occurrences of exogenous actions. The agent is, however, aware of the fact that these assumptions can be contradicted by observations. As a result the agent is ready to observe and to take into account occasional occurrences of exogenous actions.

The following example will be used throughout the section.

Example 6.2.1. *Consider a system PS consisting of an agent operating an analog circuit $\mathcal{AC}_{diag}$ from Figure 6.1. We assume that switches s_1 and s_2 are mechanical components which cannot become damaged. Relay* r *is a magnetic coil. If not damaged, it is activated when s_1 is closed, causing s_2 to close. Undamaged bulb* b *emits light if s_2 is closed. For simplicity of presentation we consider the agent capable of performing only one action, close(s_1). The environment can be represented by two damaging exogenous actions: brk, which causes* b *to become faulty, and srg (power surge),*

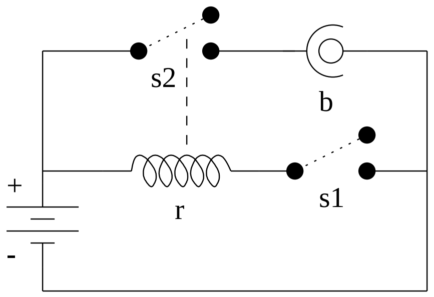


Figure 6.1: Circuit $\mathcal{AC}_{diag}$

which damages r *and also* b *assuming that* b *is not protected. Suppose that the agent operating this device is given a goal of lighting the bulb. He realizes that this can be achieved by closing the first switch, performs the operation, and discovers that the bulb is not lit. The goal of this section is to develop methods for modeling the agent's behavior after this discovery.*

The following is a description of system PS from Example 6.2.1:

$$Objects \begin{cases} r, b : \text{component} \\ s_1, s_2 : \text{switch} \end{cases} \qquad Fluents \begin{cases} active(r) \\ on(b) \\ prot(b) \\ closed(SW) \\ ab(X) \end{cases}$$

$$\begin{matrix} Agent \\ Actions \end{matrix} \begin{cases} close(s_1) \end{cases} \qquad \begin{matrix} Exogenous \\ Actions \end{matrix} \begin{cases} brk \\ srg \end{cases}$$

- *Laws describing normal functioning of PS:*

$$\begin{array}{l}
close(s_1) \text{ causes } closed(s_1) \\
\quad \text{caused } active(r) \text{ if } closed(s_1), \neg ab(r) \\
\quad \text{caused } closed(s_2) \text{ if } active(r) \\
\quad \text{caused } on(b) \text{ if } closed(s_2), \neg ab(b) \\
\quad \text{caused } \neg on(b) \text{ if } \neg closed(s_2) \\
close(s_1) \text{ impossible\_if } closed(s_1)
\end{array}$$

- *Information about system's abnormal behavior:*

$$\begin{array}{ll}
brk \text{ causes } ab(b) & \\
 & \text{caused } \neg on(b) \text{ if } ab(b) \\
srg \text{ causes } ab(r) & \\
 & \text{caused } \neg active(r) \text{ if } ab(r) \\
srg \text{ causes } ab(b) \text{ if } \neg prot(b) &
\end{array}$$

Now consider a history, Γ_1, of PS:

$$\Gamma_1 \begin{cases} hpd(close(s_1), 0). & obs(\neg ab(b), 0). \\ obs(\neg closed(s_1), 0). & obs(\neg ab(r), 0). \\ obs(\neg closed(s_2), 0). & obs(prot(b), 0). \end{cases}$$

Γ_1 says that, initially, the agent observed that s_1 and s_2 were open, both the bulb, b, and the relay, r, were not to be damaged, and the bulb was protected from surges. Γ_1 also contains the observation that action $close(s_1)$ occurred at step 0.

6.2.1.1 Basic Definitions

Our definition of candidate diagnosis of a symptom (6.2) is based on the notion of *explanation* from [3]. According to that terminology, an explanation, E, of symptom (6.2) is a collection of statements

$$E = \{hpd(a_e, s) \mid 0 \leq s < n \text{ and } a_e \in action_{ex}(\Sigma)\} \tag{6.3}$$

such that $H^n \cup O_n^{cT} \cup E$ is consistent. We also introduce the notion of possible fault-set of an explanation E, which intuitively corresponds to the set of components of PS that may be damaged by actions from E. More precisely:

Definition 6.2.1. *A set $\Delta_E \subseteq Cm$ is a* possible fault-set *of explanation E if there exists a model M of $H^n \cup O_n^{cT} \cup E$ such that:*

$$\Delta_E = \{c \mid M \models_{AD} h(ab(c), cT)\}.$$

Definition 6.2.2. *A* candidate diagnosis *of symptom (6.2) is a pair*

$$cD = \langle E, \Delta_E \rangle,$$

where E is an explanation of the symptom and Δ_E is a possible fault-set of E.

We denote the elements of a candidate diagnosis, cD, by $E(cD)$ and $\Delta(cD)$ respectively.

Definition 6.2.3. *We say that Di is a* diagnosis *of a symptom $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$ if Di is a candidate diagnosis of $\mathcal{S}$ such that all components of $\Delta(Di)$ are faulty, i.e., for every $c \in \Delta(Di)$, $ab(c) \in \sigma(W, cT)$.*

6.2.1.2 Computing candidate diagnoses

Now we show how the need for diagnosis can be determined and candidate diagnoses found by the techniques of answer set programming.

Now let $DP = \langle D, W \rangle$ be a diagnostic problem, $\mathcal{S}$ be a configuration of DP of the form (6.2), and let

$$Conf(\mathcal{S}) = \alpha(AD) \cup H^n \cup O_n^{cT} \cup R \tag{6.4}$$

where

$$R \begin{cases} h(f, 0) & \leftarrow \quad \text{not } h(\neg f, 0). \\ h(\neg f, 0) & \leftarrow \quad \text{not } h(f, 0). \end{cases}$$

for every fluent f from the signature of D. The rules of R are sometimes called the *awareness axioms*. They guarantee that initially the agent considers all possible values of the domain fluents. (If the agent's information about the initial state of the system is complete these axioms can be omitted.) The following theorem forms the basis for our diagnostic algorithms.

Theorem 6.2.1. *Let $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$ where H^n is consistent. Then configuration $\mathcal{S}$ is a symptom of system's malfunctioning iff program $Conf(\mathcal{S})$ has no answer set.*

Proof. *The conclusion follows from Lemma 4.2.2, and from Corollary 1 from [3].*

$\diamond$

To diagnose the system, PS, we construct a program, DM, defining the *explanation space* of our diagnostic agent – a collection of sequences of exogenous events which could happen (unobserved) in the system's past and serve as possible explanations of the unexpected observations. We call such programs *diagnostic modules* for PS.

The simplest diagnostic module, DM_0, consists of the rule:

$$\{o(A,T) : ex\_action(A)\} \leftarrow 0 \leq T < n.$$

where *ex_action* is a relation that is true for exogenous actions.

Finding the candidate diagnoses of symptom $\mathcal{S}$ can be reduced to finding the answer sets of the *diagnostic program*

$$D_0(\mathcal{S}) = Conf(\mathcal{S}) \cup DM_0. \tag{6.5}$$

The link between answer sets and candidate diagnoses is described by the following definition.

Definition 6.2.4. *Let D be a physical domain description, $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$ be a symptom of the system's malfunctioning, X be a set of ground literals, and E and δ be sets of ground atoms. We say that $\langle E, \Delta \rangle$ is* determined by X *if*

$$E = \{hpd(a,t) \mid o(a,t) \in X \wedge a \in action_{ex}(\Sigma)\}, \text{ and}$$

$$\Delta = \{c \mid obs(ab(c), cT) \in X\}.$$

Theorem 6.2.2. *Let $\langle D, W \rangle$ be a diagnostic problem, $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$ be a symptom of the system's malfunctioning, and E and Δ be sets of ground atoms. Then,*

$$\langle E, \Delta \rangle \text{ is a candidate diagnosis of } \mathcal{S}$$

iff

$$\langle E, \Delta\rangle \text{ is determined by an answer set of } D_0(\mathcal{S}).$$

Proof. *The conclusion follows from Lemma 4.2.2, and from Theorem 2 from [3].*

$\diamond$

The theorem justifies the following simple algorithm for computing candidate diagnosis of a symptom $\mathcal{S}$:

```
function $Candidate\_Diag$($\mathcal{S}$: symptom);
 Input: a symptom $\mathcal{S} = \langle H^n, O_n^{cT}\rangle$.
 Output: a candidate diagnosis of the symptom, or $\langle\emptyset,\emptyset\rangle$ if no candidate
     diagnosis could be found.
 var $E$ : history;
     $\Delta$ : set of components;

   if $D_0(\mathcal{S})$ is consistent then
     select an answer set, $X$, of $D_0(\mathcal{S})$;
     compute $\langle E, \Delta\rangle$ determined by $X$;
   else
     $E := \emptyset$; $\Delta := \emptyset$;
   end

   return $\langle E, \Delta\rangle$;
 end
```

Given a symptom $\mathcal{S}$, the algorithm constructs the program $D_0(\mathcal{S})$ and passes it as an input to the answer set solver. If no answer set is found the algorithm returns $\langle\emptyset,\emptyset\rangle$. Otherwise the algorithm returns a pair $\langle E, \Delta\rangle$ extracted from some answer set X of the program. By Theorem 6.2.2 the pair is a candidate diagnosis of $\mathcal{S}$. Notice that the set E extracted from an answer set X of $D_0(\mathcal{S})$ cannot be empty and hence the answer returned by the function is unambiguous. (Indeed, using the Splitting Set Theorem [45, 76] we can show that the existence of answer set of $D_0(\mathcal{S})$ with empty E will lead to existence of an answer set of $Conf(\mathcal{S})$, which, by Theorem 6.2.1, contradicts $\mathcal{S}$ being a symptom.)

The algorithm can be illustrated by the following example.

Example 6.2.2. Let us again consider system PS from Example 6.2.1. According to Γ_1 initially the switches s_1 and s_2 are open, all circuit components are ok, s_1 is closed by the agent, and b is protected. It is predicted that b will be *on* at 1. Suppose that, instead, the agent observes that at step 1 bulb b is *off*, i.e. $O_1 = \{obs(\neg on(b), 1)\}$. Intuitively, this is viewed as a symptom $\mathcal{S}_0 = \langle \Gamma_1, O_1 \rangle$ of malfunctioning of PS. Program $Conf(\mathcal{S}_0)$ has no answer sets and therefore, by Theorem 6.2.1, $\mathcal{S}_0$ is indeed a symptom. Diagnoses of $\mathcal{S}_0$ can be found by computing the answer sets of $D_0(\mathcal{S}_0)$ and extracting the necessary information from the computed answer sets. It is easy to check that, as expected, there are three candidate diagnoses:

$$\begin{aligned}
D_1 &= \langle \{o(brk, 0)\}, \{b\} \rangle \\
D_2 &= \langle \{o(srg, 0)\}, \{r\} \rangle \\
D_3 &= \langle \{o(brk, 0), o(srg, 0)\}, \{b, r\} \rangle
\end{aligned}$$

which corresponds to our intuition. Theorem 4.3.1 guarantees correctness of this computation.

The basic diagnostic module D_0 can be modified in many different ways. For instance, a simple modification, $D_1(\mathcal{S})$, which eliminates some candidate diagnoses containing actions unrelated to the corresponding symptom can be constructed as follows. First, let us introduce some terminology. Let REL be the following program:

$$REL \left\{ \begin{array}{llll}
1. & rel(A, L) & \leftarrow & dlaw(D), \\
 & & & parlist(D, P), \\
 & & & head(D, P, L), \\
 & & & action(D, P, A). \\
2. & rel(A, L) & \leftarrow & law(D), \\
 & & & parlist(D, P), \\
 & & & head(D, P, L), \\
 & & & prec(D, P, N, Pr), \\
 & & & rel(A, Pr). \\
3. & rel(A_2, L) & \leftarrow & rel(A_1, L), \\
 & & & impcond(D), \\
 & & & parlist(D, P), \\
 & & & action(D, P, A_1), \\
 & & & prec(D, P, N, Pr), \\
 & & & rel(A_2, \overline{Pr}). \\
4. & rel(A) & \leftarrow & obs(L, T), \\
 & & & T \geq n, \\
 & & & rel(A, L). \\
5. & & \leftarrow & T < n, \\
 & & & o(A, T), \\
 & & & ex\_action(A), \\
 & & & not\ hpd(A, T), \\
 & & & not\ rel(A).
\end{array} \right.$$

and

$$DM_1 = DM_0 \cup REL \cup \alpha(D).$$

The new diagnostic module, D_1 is defined as

$$D_1(\mathcal{S}) = Conf(\mathcal{S}) \cup DM_1.$$

(It is not difficult to see that this modification is *safe*, i.e. D_1 will not miss any useful predictions about the malfunctioning components.) The difference between $D_0(\mathcal{S})$ and $D_1(\mathcal{S})$ can be seen from the following example.

Example 6.2.3. Let us expand the system PS from Example 6.2.1 by a new component, c, unrelated to the circuit, and an exogenous action a which damages this component. It is easy to see that diagnosis $\mathcal{S}_0$ from Example 6.2.1 will still be a symptom of malfunctioning of a new system, S_a, and that the basic diagnostic module applied to S_a will return diagnoses $(D_1) - (D_3)$ from Example 6.2.2 together with new diagnoses containing a and $ab(c)$, e.g.

$$D_4 = \langle \{o(brk, 0), o(a, 0)\}, \{b, c\} \rangle.$$

Diagnostic module D_1 will ignore actions unrelated to $\mathcal{S}$ and return only $(D_1) - (D_3)$.

It may be worth noticing that the distinction between *hpd* and *o* allows exogenous actions, including those unrelated to observations, to actually happen in the past. Constraint (5) of program REL only prohibits generating such actions in our search for diagnosis.

There are many other ways of improving quality of candidate diagnoses by eliminating some redundant or unlikely diagnoses, and by ordering the corresponding search space. For instance, even more unrelated actions can be eliminated from the search space of our diagnostic modules by considering relevance relation *rel* depending on time. This can be done by a simple modification of program REL which is left as an exercise to the reader. The diagnostic module D_1 can also be further modified by limiting its search to recent occurrences of exogenous actions. This can be done by

$$D_2(\mathcal{S}) = Conf(\mathcal{S}) \cup DM_2,$$

where DM_2 is obtained by replacing atom $0 \leq T < n$ in the bodies of rules of DM_0 by $n - m \leq T < n$. The constant m determines the time interval in the past that an agent is willing to consider in its search for possible explanations. To simplify our discussion in the rest of the dissertation we *assume that* $m = 1$. Finally, the rule

$$\leftarrow \quad k\{o(A, n-1)\}.$$

added to DM_2 will eliminate all diagnoses containing more than k actions. Of course the resulting module D_3 as well as D_2 can miss some candidate diagnoses and deepening of the search and/or increase of k may be necessary if no diagnosis of a symptom is found. There are many other interesting ways of constructing efficient diagnostics modules. In particular, it is possible to use CR-Prolog and its preferences in the encoding of exogenous actions to express the relative likelihood of such actions. We will come back to this topic in Chapter VIII.

6.2.1.3 Finding a diagnosis

Suppose now the agent has a candidate diagnosis cD of a symptom $\mathcal{S}$. Is it indeed a diagnosis? To answer this question the agent should be able to test components of $\Delta(cD)$. Assuming that *no exogenous actions occur during testing* a diagnosis can be found by executing the following algorithm, $Find\_Diag(\mathcal{S})$:

```
function $Find\_Diag$( var $\mathcal{S}$: symptom);
  Input: a symptom $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$.
  Output: a diagnosis of the symptom, or $\langle \emptyset, \emptyset \rangle$ if no diagnosis
    could be found. Upon successful termination of the loop $O_n^{cT}$
    is updated in order to incorporate the results of the tests
    performed during the search for a diagnosis.
  var $O$, $E$ : history;
      $\Delta$, $\Delta_0$ : set of components;
      $diag$ : bool;

    $O$ := the collection of observations of $O_n^{cT}$;
    repeat
```

```
        $\langle E, \Delta \rangle := Candidate\_Diag(\ \langle H^n, O_n^{cT} \rangle\ );$
        **if** $E = \emptyset$ { no diagnosis could be found }
           **return** $\langle E, \Delta \rangle$;
        diag := $true$;    $\Delta_0 := \Delta$;
        **while** $\Delta_0 \neq \emptyset$ **and** diag **do**
           select $c \in \Delta_0$;    $\Delta_0 := \Delta_0 \setminus \{c\}$;
           **if** $observe(cT, ab(c)) = ab(c)$ **then**
              $O := O \cup obs(ab(c), cT)$;
           **else**
              $O := O \cup obs(\neg ab(c), cT)$;
              diag := $false$;
           **end**
        **end** {while}
     **until** diag;
     set the collection of observations of $O_n^{cT}$ equal to $O$;
     **return** $\langle E, \Delta \rangle$.
  **end**
```

The properties of *Find_Diag* are described by the following theorems.

Theorem 6.2.3. *For every physical domain description, D, and symptom $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$,*

$$Find\_Diag(\mathcal{S}) \text{ terminates.}$$

Proof. *The conclusion follows from Lemma 4.2.2, and from part 1 of Theorem 3 from [3] (the proof is given in Lemma 9 of the extended version of [3], available online at* `http://www.krlab.cs.ttu.edu/Papers`*).*

◇

Theorem 6.2.4. *For every physical domain description, D, symptom $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$:*

if $\langle E, \Delta \rangle = Find\_Diag(\mathcal{S})$, then

- *if* $\Delta \neq \emptyset$, *then*

$$\langle E, \Delta \rangle \textit{ is a diagnosis of } \mathcal{S};$$

- *otherwise,* $\mathcal{S}$ *has no diagnosis.*

Proof. *The conclusion follows from Lemma 4.2.2, and from part 2 of Theorem 3 from [3].*

◇

To illustrate the algorithm, consider the following example.

Example 6.2.4. Consider the system PS from Example 6.2.1 and a history Γ_1 in which b is not protected, all components of PS are ok, both switches are open, and the agent closes s_1 at step 0. At step 1, he observes that the bulb b is not lit, considers $\mathcal{S} = \langle \Gamma_1, O_1 \rangle$ where $O_1 = \{obs(\neg on(b), 1)\}$ and calls function $Need\_Diag(\mathcal{S})$ which searches for an answer set of $Conf(\mathcal{S})$. There are no such sets, the diagnostician realizes he has a symptom to diagnose and calls function $Find\_Diag(\mathcal{S})$. Let us assume that the first call to $Candidate\_Diag$ returns

$$PD_1 = \langle \{o(srg, 0)\}, \{r, b\} \rangle$$

Suppose that the agent selects component r from Δ and determines that it is not faulty. Observation $obs(\neg ab(r), 1)$ will be added to O_1, $diag$ will be set to $false$ and the program will call $Candidate\_Diag$ again with the updated symptom $\mathcal{S}$ as a parameter. $Candidate\_Diag$ will return another possible diagnosis

$$PD_2 = \langle \{o(brk, 0)\}, \{b\} \rangle$$

The agent will test bulb b, find it to be faulty, add observation $obs(ab(b), 1)$ to O_1 and return PD_2. If, however, according to our actual evolution, W, the bulb is still ok, the function returns $\langle \emptyset, \emptyset \rangle$. No diagnosis is found and the agent (or its designers) should start looking for a modeling error.

6.2.1.4 Diagnostics and repair

Now let us consider a scenario which is only slightly different from that of the previous example.

Example 6.2.5. Let Γ_1 and observation O_1 be as in Example 6.2.4 and suppose that the program's first call to $Candidate\_Diag$ returns PD_2, b is found to be faulty, $obs(ab(b), 1)$ is added to O_1, and $Find\_Diag$ returns PD_2. The agent proceeds to have b repaired but, to his disappointment, discovers that b is still not on! Intuitively this means that PD_2 is a wrong diagnosis - there must have been a power surge at 0.

For simplicity we assume that, similar to testing, repair occurs in well controlled environment, i.e. *no exogenous actions happen during the repair process.* The example shows that, *in order to find a correct explanation of a symptom, it is essential for an agent to repair damaged components and observe the behavior of the system after repair.* To formally model this process we introduce a special agent action, $repair(c)$, for every component c of the physical system. The effect of this action will be defined by the causal law:

$$repair(c) \text{ causes } \neg ab(c).$$

The diagnostic process will be now modeled by the following algorithm: (Here $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$ and $\{obs(f_i, k)\}$ is a collection of observations the agent makes to test his repair at moment k.)

```
function $Diagnose(\mathcal{S})$ : boolean;
  Input: a symptom $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$.
  Output: false if no diagnosis can be found. Otherwise
    repairs the system, updates the second element of $\mathcal{S}$, and returns true.
  var  $E$ : history;
       $\Delta$ : set of components;

    $E = \emptyset$;
    while $Need\_Diag(\langle H^n \cup E, O_n^{cT} \rangle)$ do
        $\langle E, \Delta \rangle = Find\_Diag(\langle H^n, O_n^{cT} \rangle)$;
        if $E = \emptyset$ then return false
        else
            $Repair(\Delta)$;
            $O := O \cup \{hpd(repair(c), m) : c \in \Delta\}$;
            $cT := cT + 1$;
            $O := O \cup \{obs(f_i, cT)\}$;
        end
    end
    return true;
end
```

Example 6.2.6. To illustrate the above algorithm let us go back to the agent from Example 6.2.5 who just discovered diagnosis $PD_2 = \langle \{o(brk, 0)\}, \{b\} \rangle$. He will repair the bulb and check if the bulb is lit. It is not, and therefore a new observation is recorded as follows:

$$O_1 := O_1 \cup \{hpd(repair(b), 1), obs(\neg on(b), 2)\}$$

$Need\_Diag(\mathcal{S})$ will detect a continued need for diagnosis, $Find\_Diag(\mathcal{S})$ will return PD_1, which, after new repair and testing will hopefully prove to be the right diagnosis.

The diagnosis produced by the above algorithm can be viewed as a reasonable interpretation of discrepancies between the agent's predictions and actual observations. To complete our analysis of step 1 of the agent's acting and reasoning loop we need to explain how this interpretation can be incorporated in the agent's history. If the diagnosis discovered is unique then the answer is obvious – O_n^{cT} is simply added to H^n, together with the corresponding occurrences of exogenous actions. If however faults of the system components can be caused by different sets of exogenous actions the situation becomes more subtle. The possible use of CR-Prolog to avoid forcing the agent to commit to a particular diagnosis will be the subject of future investigation.

Notice that although the above algorithms always return reasonable diagnoses, often they find too many of them. To narrow the search to “best” diagnoses, a way to specify preferences among them had to be developed. In the initial steps of our investigation on the specification of preferences, we found no natural, general way to specify preferences of this sort using A-Prolog. This led us to the development of CR-Prolog (see Chapter VII). The use of CR-Prolog for diagnosis is described later, in Chapter VIII.

6.2.2 Learning

In the previous section we have described how problematic recorded histories are detected and corrected. In this section, we focus on problematic action descriptions, i.e. on the case when the action description fails to model some aspect of the domain. We will show how answer set programming can be used to perform learning tasks.

Recall that, by definition, in learning tasks the agent's knowledge about occurrences of exogenous actions is assumed to be *complete*. When the action description is hypothesized to be problematic, the agent finds a new action description, which accounts for the unexpected observations and remains consistent with the rest of the recorded history. *We assume that all action descriptions in this section are in normal form* (see Section 4.1). Although the results presented here are not difficult to extend to the general case, focusing on action descriptions in normal form will help simplify the presentation.

Let us begin by introducing some terminology. *Modification statements* of action description AD are expressions:

- $dlaw(w)$, or
  $slaw(w)$
  (where w is an unused constant from the action signature of AD) meaning that w is the name of a new dynamic law or state constraint;

- $head(w, l)$
  (where l is an *unbound fluent literal* from the signature of AD and w is the name of a new law) meaning that the head of w is l;

- $action(w, a_e)$
  (where a_e is an *unbound elementary action* and w is the name of a new dynamic law) meaning that one element of the trigger of w is a_e;

- $prec(w, p)$
  (where p is an *unbound fluent literal or a static* and w is the name of a law) meaning that p is a precondition of w.

A collection of modification statements Mod is *valid* if: (recall that every w that occurs in statements $dlaw(w)$ or $slaw(w)$ is a fresh constant by definition of modification statement)

- for every w, only one between $dlaw(w)$ and $slaw(w)$ occurs in Mod;
- $head(w, l) \in Mod$ for some l iff either $dlaw(w) \in Mod$ or $slaw(w) \in Mod$;
- for every w, at most one statement $head(w, l)$ occurs in Mod;
- $action(w, a_e) \in Mod$ for some a_e iff $dlaw(w) \in Mod$;
- for every w, at most one statement $action(w, a_e)$ occurs in Mod;

Intuitively, these requirements ensure that: (1) the definition of every new law contains exactly one head (and one trigger, for dynamic laws), and that (2) heads and triggers are defined only for new laws.

Notice that the trigger of new laws is allowed to contain only one elementary action. The restriction is not difficult to lift, but will simplify the presentation.

Definition 6.2.5. *The* update *of action description AD with respect to a valid collection of modification statements Mod, is an action description AD' obtained from AD as follows:*

- *for every w, l such that $slaw(w) \in Mod$ and $head(w,l) \in Mod$, AD' contains a new state constraint w whose head is l;*

- *for every w, l, a_e such that $dlaw(w)$, $head(w,l)$ and $action(w,a_e)$ belong in Mod, AD' contains a new dynamic law w whose head is l and whose trigger is a_e;*

- *for every $prec(w,p) \in Mod$, p is added to the body of w in AD'.*

The update of AD with respect to Mod is denoted by $upd(AD, Mod)$.

Definition 6.2.6. *A valid collection of modification statements, Mod, is a* modification of an action description (in normal form) *AD* for symptom *$\mathcal{S} = \langle H^n, O_n^{cT}\rangle$ if:*

- *$upd(AD, Mod)$ is in normal form, and*

- *$H^n \cup O_n^{cT}$ is consistent w.r.t. $upd(AD, Mod)$.*

Similarly to diagnosis, we introduce the notion of possible fault-set of a modification Mod (with respect to AD and $\mathcal{S}$), which intuitively corresponds to the set of components of PS that may be damaged according to the predictions of $upd(AD, Mod)$ with respect to the history in $\mathcal{S}$.

Definition 6.2.7. *A set $\Delta \subseteq Cm$ is a* possible fault-set *of modification Mod with respect to action description AD and symptom $\mathcal{S} = \langle H^n, O_n^{cT}\rangle$ if there exists a model M of $H^n \cup O_n^{cT}$ such that:*

$$\Delta = \{c \mid M \models_{upd(AD,Mod)} h(ab(c), cT)\}.$$

Notice that modifications can have multiple possible fault-sets, e.g. if the updated action descriptions are non-deterministic. We are now ready to characterize candidate corrections and corrections of an action description.

Definition 6.2.8. *A* candidate correction of action description AD w.r.t. symptom $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$ *is a pair*

$$cC = \langle Mod, \Delta \rangle,$$

where Mod is a modification of AD for $\mathcal{S}$ and Δ is a possible fault-set of Mod.

The elements of candidate correction cC are denoted by $Mod(cC)$ and $\Delta(cC)$ respectively.

We say that a candidate correction cC has *weight* n if the parameter list of one law of $upd(AD, Mod(cC))$ contains n new variables, and the parameter list of no law contains more than n variables.

Definition 6.2.9. *A candidate correction cC of action description AD w.r.t. symptom $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$ is a* correction of AD w.r.t. $\mathcal{S}$ *if all the components of $\Delta(cC)$ are faulty, i.e. for every $c \in \Delta(cC)$, $ab(c) \in \tau(W, cT)$.*

The following example illustrates the definitions.

Example 6.2.7. *Let us examine more closely the learning process described in Example 3.1.4. First of all, we will need to formalize the action description for that* *domain. A possible formalization, that we denote by AD_l, is as follows.*

%% $flip(SW)$ *closes* SW *if it is open, and vice-versa*

$d_1 : flip(SW)$ *causes* $closed(SW)$ *if* $\neg closed(SW)$.

$d_2 : flip(SW)$ *causes* $\neg closed(SW)$ *if* $closed(SW)$.

%% $blow\_up(B)$ *breaks bulb* B

$d_3 : blow\_up(B)$ *causes* $ab(B)$.

%% B *is lit if the corresponding* SW *is closed,*

%% *unless* B *or batt are broken*

$s_1 : on(B)$ *if* $controls(SW, B), closed(SW), \neg ab(B), \neg ab(batt)$.

%% B *is not lit if the corresponding* SW *is open...*

$s_2 : \neg on(B)$ *if* $controls(SW, B), \neg closed(SW)$.

%% *...or if* B *or batt are malfunctioning*

$s_3 : \neg on(B)$ *if* $ab(B)$.

$s_4 : \neg on(B)$ *if* $ab(batt)$.

%% *The battery is always replaced with a working one*

$d_4 : replace(batt)$ *causes* $\neg ab(batt)$.

Recall that the agent:

1. *observes*

$$\{closed(sw_1), \neg closed(sw_2), \neg ab(b_1), \neg ab(b_2),$$
$$lit(b_1), \neg lit(b_2), \neg ab(batt)\}$$

2. *performs* $flip(sw_2)$

3. *observes*

$$\{closed(sw_2), \neg lit(b_1), \neg lit(b_2)\}$$

Observations $\neg lit(b_1), \neg lit(b_2)$ are unexpected, and no diagnosis can be found that explains them.

A few possible modifications of the action description are:

- ***Modification 1*** *consists of the statements:*

  $$\begin{aligned}&slaw(s_1^*),\\ &head(s_1^*, ab(x_1)),\ prec(s_1^*, eq(x_1, batt)),\\ &prec(s_1^*, closed(x_2)),\ prec(s_1^*, eq(x_2, sw_1)),\\ &prec(s_1^*, closed(x_3)),\ prec(s_1^*, eq(x_3, sw_2))\end{aligned}$$

  where x_i's are variables from the signature of AD. The statements correspond to a new state constraint:

  $$s_1^* : ab(x_1)\ \textit{if}\ x_1 = batt, closed(x_2), x_2 = sw_1, closed(x_3), x_3 = sw_2,$$

  which is the normal form version of:

  $$s_1^* : ab(batt)\ \textit{if}\ closed(sw_1), closed(sw_2).$$

- ***Modification 2*** *consists of the statements:*

  $$\begin{aligned}&slaw(s_2^*),\\ &head(s_2^*, ab(x_1)),\ prec(s_2^*, eq(x_1, batt)),\\ &prec(s_2^*, closed(x_2)),\ prec(s_2^*, closed(x_3)),\\ &prec(s_2^*, neq(x_2, x_3))\end{aligned}$$

  which correspond to a new state constraint (sw' and sw'' are variables):

  $$s_2^* : ab(batt)\ \textit{if}\ closed(sw'), closed(sw''), sw' \neq sw''.$$

  Notice that this law is more general than the one encoded by modification 1.

- ***Modification 3*** *consists of the statements:*

  $$\begin{aligned}&slaw(s_3^*),\\ &head(s_3^*, ab(x_1)),\\ &prec(s_3^*, closed(x_1)),\ prec(s_3^*, eq(x_1, sw_1)),\\ &prec(s_3^*, closed(x_2)),\ prec(s_3^*, eq(x_2, sw_2))\end{aligned}$$

which correspond to a new state constraint (b is a variable):

$$s_3^* : ab(b) \ \textit{if} \ closed(sw_1), closed(sw_2)$$

saying that closing both switches causes the bulbs to break.

- ***Modification 4*** *consists of the statements:*

$$\begin{aligned}&dlaw(d_1^*),\\ &head(d_1^*, ab(x_1)), \ prec(d_1^*, eq(x_1, batt)),\\ &action(d_1^*, flip(x_2)), \ prec(d_1^*, eq(x_2, sw_2)),\\ &prec(d_1^*, closed(x_3)), \ prec(d_1^*, eq(x_3, sw_2))\end{aligned}$$

  which correspond to a new dynamic law:

$$d_1^* : flip(sw_2) \ \textit{causes} \ ab(batt), \neg closed(sw_2),$$

  saying that closing sw_2 causes the battery to malfunction.

Notice that all the modifications shown explain the observations by concluding either that batt is malfunctioning or that both bulbs are broken.

Now, assume that, in the actual evolution of the system, only batt is faulty (e.g. it overloaded). Hence, of the above set, only modifications 1, 2 and 4 correspond to corrections, because their predictions are true in the actual evolution. Modification 3 does not correspond to a correction because it incorrectly predicts $ab(b_1)$ and $ab(b_2)$.

Let us now focus on how candidate corrections and corrections of symptoms are computed using A-Prolog.

6.2.2.1 Computing candidate corrections

As for diagnosis, to determine if a configuration $\mathcal{S}$ is a symptom, we test whether $Conf(\mathcal{S})$ is consistent (see Theorem 6.2.1).

Once a symptom has been identified, we correct the action description of system PS by constructing a program, LM, that defines the *modification space* of the learning agent (a collection of modifications of the original action description which serve to justify the unexpected observations). We call such programs *learning modules* for PS.

Intuitively, finding candidate corrections is reduced to computing the answer sets of the program consisting of the learning module together with the domain description. In fact, the *learning module* is conceptually very similar to the diagnostic module. However, the actual task is made more complex than that of the diagnostic module because of the more elaborate structure of the object that the module updates. This results in the need for extra information about the structure of the action description to be provided to the learning module.

Therefore, we assume the existence of a function χ, external to the A-Prolog module, that extends mapping α with the needed information.

Notice that the use of χ *does not invalidate our claim that all the reasoning components of the agent use the same knowledge.* It is easy to show that χ can be used in place of α both in the planning module and in the diagnostic module.[5]

Let us start by defining the arguments of χ. Recall (Chapter IV) that, for every law in normal form w, $\varpi(w)$ contains at most one occurrence of each variable from w. Consequently, every time a precondition is added to the body of w, the parameter list of w needs to be expanded accordingly.

For this reason, χ is defined to take as arguments both an action description AD and a non-negative integer n, where n is the number of new variables to be added to the arguments of each term $pars(X_1, \dots, X_k)$ from $\alpha(AD)$. The value of $\chi(AD, m)$ is a modified version of $\alpha(AD)$, where m new variables have been added to the arguments of each term formed by $pars$ that occurs in $\alpha(AD)$, and the mapping defined by predicate par is expanded accordingly. The value of $\chi(AD, m)$ also includes the definitions of other relations, as described below.

For each law w, two facts are included in $\chi(AD, m)$ to help the algorithm keep

[5]To prove the statement, it is sufficient to observe that the added set of rules is stratified, and that the relations defined by such rules do not belong to the signature of $\alpha(AD)$.

track of the number of preconditions and variables initially used by each law:

- $initially(has\_precs(L, n_L))$, informally read "initially, law L has n_L preconditions,"
- $initially(has\_vars(L, i_L))$, read "initially, law L has i_L variables in its parameter list."

Recall that that the tokens occurring in the modification statements can be unbound. When implementing the learning task, we need a way to denote unbound tokens, i.e. sets of ground tokens, in A-Prolog. Hence, we introduce *token names*, which can denote both single ground tokens and (certain) sets of tokens. Token names are identified by relations *lit_name* (for AL-literals) and *act_name* (for actions), defined as follows.

Relation *lit_name* is defined in χ as follows:

- for every fluent predicate f, both f and $\neg f$ are AL-literal names[6] (denoting the corresponding unbound fluent literals);
- for every static predicate r with arity n, $r(t_1, \ldots, t_n)$ is an AL-literal name, where each t_i is either a constant or a special term $v(j)$, j being an integer denoting a position in some parameter list. Similarly, $\neg r(t_1, \ldots, t_n)$ is an AL-literal name, for all possible assignments of t_i's. (Intuitively, the special terms $v(j)$ denote variables of $\mathcal{AL}$, while the other terms denote themselves.)

Relation *act_name* is defined similarly to the way *lit_name* is defined for fluent literals. Since later we will also need to distinguish between names of fluent predicates and names of statics, we include in χ the extension of relation *static*, such that *static* holds for statics as well as for their names (the extension of relation $fliteral$ can be obtained by applying the closed world assumption, if needed).

The association between token names and tokens is encoded in χ by a relation $denotes(TNAME, I, L, P, TOK)$ which says that token name $TNAME$, when

[6]For simplicity, here we do not allow the use of the same fluent predicate with different arities.

considering the I^{th} variable of law L's parameter list P, denotes token TOK. Finally, relation $needs\_variables(TNAME, N)$ states that token name $TNAME$ needs N fresh variables in the parameter list of a law to which it is added (hence, all statics have a corresponding value of 0).

Example 6.2.8. *Consider an action description AD_{d_1} containing only the dynamic law d_1:*

$$\underline{d_1} : \{a\} \textit{ causes } g(x) \textit{ if } \quad l_1(y, z), l_2(v), \\ eq(x, y), eq(z, c_1), eq(v, c_2).$$

where c_1, c_2 are constants, g, l_i's are fluent literals, a is an elementary action, and eq is a static corresponding to the identity relation. The parameter list of d_1 is $\langle x, y, z, v\rangle$. In this case, $\chi(AD_{d_1}, 2)$ is as follows.

The definition of parlist and par in $\chi(AD_{d_1}, 2)$ consists of the atoms (we use Y_i for the variables added by χ to make them easily recognizable):

$$parlist(\underline{d_1}, pars(X_1, \ldots, X_4, Y_1, Y_2)).$$

$$par(1, pars(X_1, \ldots, X_4, Y_1, Y_2), X_1).$$
$$\ldots$$
$$par(4, pars(X_1, \ldots, X_4, Y_1, Y_2), X_4).$$
$$par(5, pars(Y_1, \ldots, X_4, Y_1, Y_2), Y_1).$$
$$par(6, pars(Y_2, \ldots, X_4, Y_1, Y_2), Y_2).$$

Relation lit_name is defined in $\chi(AD_{d_1}, 2)$ *by:*

$lit\_name(g).$	$lit\_name(\neg g).$
$lit\_name(l_1).$	$\ldots$
$lit\_name(l_2).$	$lit\_name(\neg l_3).$
$lit\_name(l_3).$	

$lit\_name(eq(v(1), v(1))).$	$lit\_name(\neg eq(v(1), v(1))).$
$lit\_name(eq(v(1), v(2))).$	$lit\_name(\neg eq(v(1), v(2))).$
$\ldots$	$\ldots$
$lit\_name(eq(v(4), v(4))).$	$lit\_name(\neg eq(v(4), v(4))).$

$lit\_name(eq(v(1), c_1)).$	$lit\_name(\neg eq(v(1), c_1)).$
$lit\_name(eq(v(1), c_2)).$	$\ldots$
$\ldots$	$lit\_name(\neg eq(c_2, v(2))).$
$lit\_name(eq(c_2, v(2))).$	

$lit\_name(eq(c_1, c_1)).$	$lit\_name(\neg eq(c_1, c_1)).$
$lit\_name(eq(c_1, c_2)).$	$\ldots$
$lit\_name(eq(c_2, c_1)).$	$lit\_name(\neg eq(c_2, c_2)).$
$lit\_name(eq(c_2, c_2)).$	

$\ldots$

Relation static is extended by:

$$static(eq(v(1), v(1))).$$
$$static(eq(v(1), v(2))).$$
$$\ldots$$
$$static(\neg eq(v(1), v(1))).$$
$$\ldots$$

Relation act_name is defined by:

$$act\_name(a).$$

Relation needs_variables is defined by:

$$needs\_variables(g, 4).$$
$$needs\_variables(l_1, 2).$$
$$\ldots$$
$$needs\_variables(eq(v(1), v(1)), 0).$$
$$\ldots$$
$$needs\_variables(eq(v(1), c_1), 0).$$

$$needs\_variables(a, 0).$$
$$\ldots$$

Relation denotes can be defined quite compactly using rules, rather than facts. The *following rules show the definition of denotes for AL-literal name l_1 in $\chi(AD_{d_1}, 2)$.*

$$\begin{aligned} denotes(l_1, I, L, P, l_1(X_1, X_2)) \leftarrow & \\ & parlist(L, P), \\ & par(I, P, X_1), \\ & par(I+1, P, X_2). \end{aligned}$$

$$\begin{aligned} denotes(\neg l_1, I, L, P, \neg l_1(X_1, X_2)) \leftarrow & \\ & parlist(L, P), \\ & par(I, P, X_1), \\ & par(I+1, P, X_2). \end{aligned}$$

The next rules show the definition of denotes for the AL-literal names of the form $eq(v(V_1), v(V_2))$.

$$\begin{aligned} denotes(eq(v(V_1), v(V_2)), I, L, P, eq(X_1, X_2)) \leftarrow & \\ & parlist(L, P), \\ & par(V_1, P, X_1), \\ & par(V_2, P, X_2). \end{aligned}$$

$$\begin{aligned} denotes(\neg eq(v(V_1), v(V_2)), I, L, P, \neg eq(X_1, X_2)) \leftarrow & \\ & parlist(L, P), \\ & par(V_1, P, X_1), \\ & par(V_2, P, X_2). \end{aligned}$$

The definition of denotes for action a is straightforward, since a has no arguments.

$$denotes(a, I, L, P, a).$$

On the other hand, if we were to define denotes for some action b with 3 arguments, *the corresponding rule would be:*

$$\begin{aligned}
denotes(b, I, L, P, b(X_1, X_2, X_3)) \leftarrow \\
&parlist(L, P), \\
&par(I, P, X_1), \\
&par(I+1, P, X_2), \\
&par(I+2, P, X_3).
\end{aligned}$$

In the rest of the discussion, we use the term precondition to refer to both AL-literals used as preconditions of laws and their names when the type of the object is clear from the context.

Given a learning module LM, finding the candidate corrections of AD for symptom $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$ is reduced to finding the answer sets of the following *learning program* for an appropriate value of parameter n:

$$L_0(AD, \mathcal{S}, m) = \chi(AD, m) \cup H^n \cup O_n^{cT} \cup R \cup LM.$$

Let us now see how the learning programs are constructed. We start by presenting a simple learning module, LM_0, which is capable of adding new preconditions to the laws, but cannot add new laws. Recall from Chapter IV that, for every law in normal form w, the fluent literals in $body(w)$ are unbound, and the statics contain either constants or variables from $\varpi(w)$. These are the only two types of tokens that we consider in the learning modules for addition to the laws.

In the module, we adopt the convention that variables beginning with N are used for indexes of preconditions and variables beginning with I are indexes of variables in a parameter list. The selection of the AL-literals that need to be added to the bodies of the laws of AD is performed in LM_0 by the following selection rule.

% Any precondition $LNAME$ can be missing from position
% N of the body of any law L.
%
$$\{missing\_prec(LNAME, N, L) : \\ prec\_name\_and\_pos(LNAME, N, L)\} \leftarrow law(L).$$

% $LNAME$ is an AL-literal name and N is
% a valid position for $LNAME$ in the body of L
$$\begin{aligned} prec\_name\_and\_pos(LNAME, N, L) \leftarrow\ & lit\_name(LNAME), \\ & legal\_prec\_pos(N, L). \end{aligned}$$

% N is a legal position for preconditions added to L if it is greater than
% the number of preconditions initially used by the law.
$$\begin{aligned} legal\_prec\_pos(N, L) \leftarrow\ & initially(has\_precs(L, N')), \\ & N > N'. \end{aligned}$$

% No two missing preconditions can be in the same position.
$$\begin{aligned} \leftarrow\ & missing\_prec(LNAME1, N, L), \\ & missing\_prec(LNAME2, N, L), \\ & LNAME1 \neq LNAME2. \end{aligned}$$

% No "holes" are allowed in the assignment of positions.
$$\begin{aligned} \leftarrow\ & missing\_prec(LNAME1, N1, L), \\ & no\_precondition\_at\_pos(L, N2), \qquad \text{% defined below} \\ & missing\_prec(LNAME3, N3, L), \\ & N2 = N1 + 1, \\ & N3 = N2 + 2. \end{aligned}$$

% There is a precondition at position N if one precondition

% was assigned that position.

$\neg no\_precondition\_at\_pos(L, N) \leftarrow \quad missing\_prec(LNAME, N, L).$

% Closed world assumption on no_precondition_at_pos.

$no\_precondition\_at\_pos(L, N) \leftarrow \quad \text{not } \neg no\_precondition\_at\_pos(L, N).$

The actual update of the action description is obtained by the rules shown below.

% If $LNAME$ denotes a missing precondition from the N^{th} position of the body

% of L, then $PREC$ is the N^{th} precondition of L under parameter list P, where

% $PREC$ is the instance of $LNAME$ using variables from the I^{th} position of P.

% (num_vars_upto_prec is defined below)

$$\begin{aligned} prec(L, P, N, PREC) \leftarrow \quad & missing\_prec(LNAME, N, L), \\ & num\_vars\_upto\_prec(I, N-1, L), \\ & denotes(LNAME, I+1, L, P, PREC). \end{aligned}$$

% If, before any modification of AD, L has N preconditions and I variables

% then I is the number of variables used by the first N preconditions of L.

$$\begin{aligned} num\_vars\_upto\_prec(I, N, L) \leftarrow \quad & initially(has\_precs(L, N)), \\ & initially(has\_vars(L, I)). \end{aligned}$$

% If $LNAME$ is the (missing) N^{th} precondition of L, the number of

% variables used by the first $N-1$ preconditions is $I1$, and $LNAME$

% uses NV new variables, then $I1 + NV$ is the number of variables

% used by the first N preconditions of L.

$$\begin{aligned} num\_vars\_upto\_prec(I2, N, L) \leftarrow \quad & missing\_prec(LNAME, N, L), \\ & num\_vars\_upto\_prec(I1, N-1, L), \\ & needs\_variables(LNAME, NV), \\ & I2 = I1 + NV. \end{aligned}$$

It is not difficult to extend LM_0 to also perform addition of new laws. Consider a learning module LM_1, which extends LM_0 by allowing the addition of dynamic laws and state constraints. Recall that, for simplicity, the trigger of new dynamic laws is only allowed to contain *one* elementary action.

For LM_1, we assume that the signature of $\chi(AD, m)$ is extended with a sufficient number of fresh constants, used to denote the laws being added. Furthermore, $\chi(AD, m)$ is extended to include:

- facts $new\_law\_name(w)$ for each new constant used for laws;
- the extension of relation *denotes* to link action names to the corresponding ground elementary actions.

LM_1 is defined as:

$$LM_1 = LM_0 \cup ADD\_LAWS.$$

Set of rules ADD_LAWS is logically divided in two parts. The first part of ADD_LAWS is responsible for the selection of names for the laws that need to be added to the action description, for determining their type, the name of the fluent literals used for their heads, and the name of the elementary action used in the trigger (where applicable).

% Any fresh constant L can denote a law that
% is missing from the action description.
%
$\{missing\_law(L) : new\_law\_name(L)\}.$

% Each missing law is either a dynamic law or a state constraint.
%
$dlaw(L)$ OR $slaw(L) \leftarrow missing\_law(L).$

% Any AL-literal $LNAME$ can be the head of any missing law...
$1\{missing\_head(LNAME, L) :$
$\quad lit\_name(LNAME)\}1 \leftarrow missing\_law(L).$
% ...as long $LNAME$ denotes a fluent literal.
$\leftarrow missing\_head(LNAME, L), static(LNAME).$

% Any action $ANAME$ can be the trigger of any missing
% dynamic law.
%
$1\{missing\_act(ANAME, L) :$
$\quad act\_name(ANAME)\}1 \leftarrow missing\_law(L),$
$\quad\quad dlaw(L).$

The second part of ADD_LAWS maps the selections made by the first part into the actual encoding of the head and trigger of each law. Moreover, it defines relations $initially(has\_precs(L, n_L))$ and $initially(has\_vars(L, i_L))$ for the new laws, as these relations are necessary to allow the addition of preconditions in LM_0.

% If $LNAME$ denotes the missing head of L, then LIT
% is the head of L under parameter list P, where LIT is
% denoted by $LNAME$ when using variables from the first
% position of P.

$$\begin{aligned} \mathit{head}(L, P, LIT) \leftarrow \quad & \mathit{missing\_head}(LNAME, L), \\ & \mathit{denotes}(LNAME, 1, L, P, LIT). \end{aligned}$$

% If $ANAME$ denotes the missing trigger of L, then ACT
% is the trigger of L under parameter list P, where ACT is
% denoted by $ANAME$ when using variables from the I^{th}
% position of P.

$$\begin{aligned} \mathit{action}(L, P, ACT) \leftarrow \quad & \mathit{missing\_act}(ANAME, L), \\ & \mathit{missing\_head}(LNAME, L), \\ & \mathit{needs\_variables}(LNAME, I), \\ & \mathit{denotes}(ANAME, I+1, L, P, ACT). \end{aligned}$$

% The initial number of preconditions in any missing law is 0.

$$\mathit{initially}(\mathit{has\_precs}(L, 0)) \leftarrow \quad \mathit{missing\_head}(LNAME, L).$$

% The initial number of variables in any missing non-dynamic
% law is the number of variables in its head.

$$\begin{aligned} \mathit{initially}(\mathit{has\_vars}(L, I)) \leftarrow \quad & \mathit{missing\_head}(LNAME, L), \\ & \mathit{slaw}(L), \\ & \mathit{needs\_variables}(LNAME, I). \end{aligned}$$

% The initial number of variables in any missing dynamic law

% is the number of variables in its head and trigger.

$$
\begin{aligned}
initially(has\_vars(L, I)) \leftarrow \;& missing\_head(LNAME, L),\\
& missing\_act(ANAME, L),\\
& needs\_variables(LNAME, I1),\\
& needs\_variables(ANAME, I2),\\
& I = I1 + I2.
\end{aligned}
$$

Candidate corrections can thus be computed by means of the learning program:

$$L_1(AD, \mathcal{S}, m) = \chi(AD, m) \cup H^n \cup O_n^{cT} \cup R \cup LM_1.$$

To better understand how $L_1(AD, \mathcal{S}, n)$ works, consider the following example.

Example 6.2.9. *Let us see how the candidate corrections corresponding to the modifications shown in Example 6.2.7 are computed by means of $L_1(AD_l, \mathcal{S}, n)$ (for a suitable value of n).*

- ***Candidate correction 1.*** *It is not difficult to show that one of the answer sets of $L_1(AD_l, \mathcal{S}, n)$ contains the following atoms, encoding modification 1 (the modification is also encoded by other answer sets, corresponding to all the pos-*

sible ways of ordering the preconditions),

$$
\begin{array}{l}
\%\ \textit{Law: } s_1^* : ab(batt)\ \textit{if}\ closed(sw_1), closed(sw_2) \\
\\
\%\ \textit{Definition of the new law} \\
missing\_law(s_1^*) \\
slaw(s_1^*) \\
\\
\%\ \textit{Head of the law: } ab(X_1)\textit{, where...} \\
missing\_head(ab, s_1^*) \\
\%\ \textit{... } X_1 = batt\ \textit{as stated by the first precondition} \\
missing\_prec(eq(v(1)), batt), 1, s_1^*) \\
\\
\%\ \textit{precondition 2: } closed(X_2) \\
missing\_prec(closed, 2, s_1^*) \\
\%\ \textit{precondition 3: } closed(X_3) \\
missing\_prec(closed, 3, s_1^*) \\
\%\ \textit{precondition 4: } X_2 = sw_1 \\
missing\_prec(eq(v(2), sw_1), 4, s_1^*) \\
\%\ \textit{precondition 5: } X_3 = sw_2 \\
missing\_prec(eq(v(3), sw_2), 5, s_1^*)
\end{array}
$$

The answer set also contains the atom $h(ab(batt), 1)$*, encoding the corresponding fault-set* $\{batt\}$*.*

- ***Candidate correction 2****. Another answer set of* $L_1(AD, \mathcal{S}, n)$ *encodes mod-*

ification 2,

> *% Law:* $s_2^* : ab(batt)$ *if* $closed(SW_1), closed(SW_2), SW_1 \neq SW_2$
>
> *% Definition of the new law*
> $missing\_law(s_2^*)$
> $slaw(s_2^*)$
>
> *% Head of the law:* $ab(X_1)$*, where...*
> $missing\_head(ab, s_2^*)$
> *% ...* $X_1 = batt$ *as stated by the first precondition*
> $missing\_prec(eq(v(1)), batt), 1, s_2^*)$
>
> *% precondition 2:* $closed(X_2)$
> $missing\_prec(closed, 2, s_2^*)$
> *% precondition 3:* $closed(X_3)$
> $missing\_prec(closed, 3, s_2^*)$
> *% precondition 4:* $X_2 \neq X_3$
> $missing\_prec(neq(v(2), v(3)), 4, s_2^*)$

together with the encoding of the fault-set $\{batt\}$.

- ***Candidate correction 3***. *Another answer set of* $L_1(AD, \mathcal{S}, n)$ *encodes mod-*

ification 3,

% *Law:* $s_3^* : ab(B)$ *if* $closed(sw_1), closed(sw_2)$

% *Definition of the new law*

$missing\_law(s_3^*)$

$slaw(s_3^*)$

% *Head of the law:* $ab(X_1)$

$missing\_head(ab, s_3^*)$

% *precondition 1:* $closed(X_2)$

$missing\_prec(closed, 1, s_3^*)$

% *precondition 2:* $closed(X_3)$

$missing\_prec(closed, 2, s_3^*)$

% *precondition 3:* $X_2 = sw_1$

$missing\_prec(eq(v(2), sw_1), 3, s_3^*)$

% *precondition 4:* $X_3 = sw_2$

$missing\_prec(eq(v(3), sw_2), 4, s_3^*)$

together with the encoding of the fault-set $\{b_1, b_2\}$.

- ***Candidate correction 4****. Another answer set of* $L_1(AD, \mathcal{S}, n)$ *encodes mod-*

ification 4,

> *% Law:* $d_1^* : flip(sw_2)$ *causes* $ab(batt), \neg closed(sw_2)$
>
> *% Definition of the new law*
> $missing\_law(d_1^*)$
> $dlaw(d_1^*)$
>
> *% Head of the law:* $ab(X_1)$*, where...*
> $missing\_head(ab, d_1^*)$
> *% ...* $X_1 = batt$ *as stated by the first precondition*
> $missing\_prec(eq(v(1)), batt), 1, d_1^*)$
>
> *% trigger:* $flip(X_2)$*, where...*
> $missing\_act(flip, d_1^*)$
> *% ...* $X_2 = sw_2$ *as stated by the second precondition*
> $missing\_prec(eq(v(1)), sw_2), 2, d_1^*)$
>
> *% precondition 3:* $\neg closed(X_3)$
> $missing\_prec(\neg closed, 3, d_1^*)$
> *% precondition 4:* $X_3 = sw_2$
> $missing\_prec(eq(v(3), sw_2), 3, d_1^*)$

together with the encoding of the fault-set $\{batt\}$.

The following algorithm uses $L_1(AD, \mathcal{S}, m)$ to compute candidate corrections of weight up to m.

```
function $Candidate\_Correction(D$: domain description, $\mathcal{S}$: symptom, $m$: integer)
  Input: a domain description $D = \langle AD, H^{cT} \rangle$;
         a symptom $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$;
         the maximum weight, $m$, of the candidate correction returned.
  **Output**: a candidate correction of $\mathcal{S}$ of weight up to $m$, or
            $\langle\emptyset,\emptyset\rangle$ if none could be found.
  **var** $Mod$ : **set of modification statements**;
      $\Delta$ : **set of components**;

      **if** $L_1(AD, \mathcal{S}, m)$ is consistent **then**
        select an answer set, $X$, of $L_1(AD, \mathcal{S}, n)$
        extract $\langle Mod, \Delta\rangle$ from $X$
        **return** $\langle Mod, \Delta\rangle$
      **end**

    **return** $\langle\emptyset,\emptyset\rangle$
  **end**
```

The extraction of $\langle Mod, \Delta\rangle$ from X (with respect to an original action description AD) is accomplished in two parts: Δ is the set $\{c \mid h(ab(c), cT) \in X\}$; essentially, Mod is obtained from X, with respect to the original action description AD, by mapping the token names to the corresponding tokens. As the reader will notice, most of the work consists in maintaining the parameter lists of the laws appropriately updated.

```
**function** $Extract\_Mod(X$: **answer set of** $L_1(AD, \mathcal{S}, n), AD$: **action description**)
  **Output**: a candidate correction of $AD$ corresponding to $X$
  **var** $Mod$ : **set of modification statements**;
      $plist$ : **a list of variables from** $\Sigma(AD)$;

    $Mod := \emptyset$;
    **for every** $missing\_law(w) \in X$ **do**
      $plist := \emptyset$;
      if $slaw(w) \in X$ then $Mod := Mod \cup \{slaw(w)\}$;
      if $dlaw(w) \in X$ then $Mod := Mod \cup \{dlaw(w)\}$;
      let $p$ be such that $missing\_head(p, w) \in X$;
      select a list $\overline{x}$ of variables from $\Sigma(AD)$, not occurring in $plist$,
                  of length equal to the arity of $p$:
        $Mod := Mod \cup \{head(w, p(\overline{x}))\}$;
        $plist := plist \circ \overline{x}$;
      in a similar way, extract $action(w, a(\overline{x}))$ from $missing\_act(a, w)$;
      $Mod := Mod \cup Extract\_prec(X, AD, w, plist)$;
    end { for }

    for every $missing\_prec(w) \in X$ such that $missing\_law(w) \notin X$ do
      $plist :=$ the parameter list of $w$ in $AD$;
      $Mod := Mod \cup Extract\_prec(X, AD, w, plist)$;
    end { for }
    return $Mod$
  end
```

The extraction of the statements $prec(w, p)$ is performed as follows.

```
function $Extract\_prec(X$: answer set of $L_1(AD, \mathcal{S}, n), AD$: action description,
                    $w$: a law name, $plist$: a parameter list)
  Output: a set of statements $prec(w, p)$ extracted from $X$
  var $Mod$ : a set of modification statements;
      $plist'$ : a list of variables from $\Sigma(AD)$;
      $n$ : an integer;

  $Mod := \emptyset$;   $plist' := plist$;
  $n := 0$;   { used to scan the $missing\_prec$ atoms in the intended order }
```

```
    while $missing\_prec(l, n, w) \in X$ do
      if $l$ is the name of a fluent literal then
        select a list $\overline{x}$ of variables from $\Sigma(AD)$, not occurring in $plist'$,
                of length equal to the arity of $l$;
        $Mod := Mod \cup \{prec(w, l(\overline{x}))\}$;
        $plist' := plist' \circ \overline{x}$;
      end { if }
      $n := n + 1$;
    end { while }

    { Notice that, at this stage, the order of the variables added to $plist'$ is
       as expected by the $missing\_prec$ atoms corresponding to statics.}
    for every $missing\_prec(r(\overline{x}), n, w) \in X$ where $r(\overline{x})$ is a static name do
      obtain $\overline{x'}$ from $\overline{x}$ by replacing every special term $v(i)$ in $\overline{x}$
              with the $i^{th}$ variable from $plist'$;
      $Mod := Mod \cup \{prec(w, r(\overline{x'}))\}$;
    end { for }
    return $P$
  end
```

6.2.2.2 Finding a correction

After finding a candidate correction cC, the agent has to verify whether cC is indeed a correction. To do this this, the agent needs to test the components of $\Delta(cC)$.

As for diagnosis, we assume that *no exogenous actions occur during testing.* Hence, a correction (of weight up to m) can be found by executing the following algorithm, $Find\_Correction(\mathcal{S}, m)$:

function $Find\_Correction(D$: **domain description, var** $\mathcal{S}$: **symptom**, m: **integer**)

```
Input: a domain description $D = \langle AD, H^{cT} \rangle$;
          a symptom $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$;
          the maximum weight, $m$, of the correction returned.
Output: a correction of the symptom of weight up to $m$, or
          $\langle \emptyset, \emptyset \rangle$ if none could be found.
          Upon successful termination of the loop $O_n^{cT}$
          is updated in order to incorporate the results of the tests
          performed during the search for a correction.
var $O$ : history;
    $cC$ : candidate correction;
    $\Delta_0$ : set of components;
    $corr\_found$ : bool;

  $O$ := the collection of observations of $O_n^{cT}$;
  repeat
    $cC := Candidate\_Correction(D,\ \langle H^n, O_n^{cT} \rangle, m)$;
    if $cC = \langle \emptyset, \emptyset \rangle$ { no correction could be found }
      return $\langle \emptyset, \emptyset \rangle$;
    corr_found := $true$;     $\Delta_0 := \Delta(cC)$;
    while $\Delta_0 \neq \emptyset$ and corr_found do
      select $c \in \Delta_0$;     $\Delta_0 := \Delta_0 \setminus \{c\}$;
      if $observe(cT, ab(c)) = ab(c)$ then
        $O := O \cup obs(ab(c), cT)$;
      else
        $O := O \cup obs(\neg ab(c), cT)$;
        corr_found := $false$;
      end
    end {while}
  until corr_found;
```

```
        set the collection of observations of $O_n^{cT}$ equal to $O$;
        return $cC$
    end
```

The properties of *Find_Correction* are described by the following propositions.

Proposition 6.2.1. *For every physical domain description, D, symptom $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$, and integer m,*

$$Find\_Correction(D, \mathcal{S}, m) \text{ terminates.}$$

Proof. *Essentially identical to the proof of Lemma 9 from the extended version of [3].*

◇

As we mentioned earlier, the trigger of the new dynamic laws introduced by LM_1 is allowed to contain only one elementary action. For this reason, we restrict our statement of soundness and completeness of algorithm *Find_Correction* to *simple corrections*, i.e. corrections that contain only one statement $action(w, a_e)$ for each new dynamic law w (*simple collections of modification statements*, mentioned below, are defined accordingly).

Proposition 6.2.2. *For every physical domain description, D, symptom $\mathcal{S} = \langle H^n, O_n^{cT} \rangle$, and integer m:*

if $cC = Find\_Correction(D, \mathcal{S}, m)$, then

- *if $\Delta(cC) \neq \emptyset$, then*

$$cC \text{ is a correction of } \mathcal{S} \text{ of weight up to } m;$$

- *otherwise, $\mathcal{S}$ has no* simple correction *of weight up to m.*

Proof. *The proof of this proposition combines the techniques from Theorems 6.1.1 and 6.1.2 in Section 6.1 with those from Theorem 6.2.4 in Section 6.2.1. The key* *idea is that the selection rules in the learning module generate atoms that encode all possible valid* simple *collections of modification statements.*

◇

To illustrate the algorithm, consider the following example.

Example 6.2.10. *Consider the scenario and the candidate corrections from Example 6.2.9. Suppose the first call to* $Candidate\_Correction$ *returns candidate correction 3. Recall that the corresponding fault-set is* $\Delta = \{b_1, b_2\}$.

Now, let us assume that the agent selects component b_1 *from* Δ *and determines that it is not faulty. Observation* $obs(\neg ab(b_1), 1)$ *is added to the recorded history,* $corr\_found$ *is set to* $false$ *and the program calls* $Candidate\_Correction$ *again.*

Suppose that, this time, $Candidate\_Correction$ *returns candidate correction 1. The fault-set of candidate correction 1 is* $\{batt\}$*. The agent will test batt, find it to be faulty, add the observation* $obs(ab(batt), 1)$ *to the recorded history, and return candidate correction 1.*

If, however, according to our actual evolution, W*, the bulb is still ok, the algorithm will eventually return* $\langle \emptyset, \emptyset \rangle$*, meaning that no correction was found (it is easy to see that any candidate correction in this scenario must have either* $\{batt\}$ *or* $\{b_1, b_2\}$ *as fault-sets).*

Similarly to diagnosis, it is often important to limit the corrections found by the reasoning algorithm to a set of "best" correction. CR-Prolog's cr-rules and preferences can be used for this purpose to specify correction selection criteria. We will discuss this topic in Section 8.3.

The next chapter introduces syntax and semantics of CR-Prolog.

CHAPTER VII
CR-PROLOG

CR-Prolog [4, 6] is an extension of A-Prolog resulting from the introduction of consistency-restoring rules and preferences over them. CR-Prolog is used in this dissertation for the formalization of the reasoning components because it allows to elegantly formalize various sophisticated reasoning tasks. We start by describing *basic CR-Prolog*, which, similarly to basic A-Prolog (see Section 2.2), does not include s-atoms.

7.1 Syntax of basic CR-Prolog

The syntax of basic CR-Prolog is defined as follows:

Definition 7.1.1. *A* basic regular rule *is a statement of the form:*

$$r : h_1 \text{ OR } h_2 \text{ OR } \ldots \text{ OR } h_k \leftarrow l_1, l_2, \ldots l_m, not\ l_{m+1}, not\ l_{m+2}, \ldots, l_n. \tag{7.1}$$

where r is a term representing the name of the rule, $l_1, \ldots, l_m$ are literals, and h_i's and $l_{m+1}, \ldots, l_n$ are plain literals.

Basic regular rules have the same informal reading as the basic rules of A-Prolog.

Definition 7.1.2. *A* basic consistency-restoring rule *(or* cr-rule*) is a statement of the form:*

$$r : h_1 \text{ OR } h_2 \text{ OR } \ldots \text{ OR } h_k \overset{+}{\leftarrow} l_1, l_2, \ldots l_m, not\ l_{m+1}, not\ l_{m+2}, \ldots, l_n. \tag{7.2}$$

where r, h_i's and l_i's are as before.

The intuitive reading of a basic cr-rule is "if you believe $l_1, \ldots, l_m$ and have no reason to believe $l_{m+1}, \ldots, l_n$, then you *may possibly* believe one of $h_1, \ldots, h_k$." The implicit assumption is that this possibility is used as little as possible.

Definition 7.1.3. *A* basic CR-Prolog program *is a pair $\langle \Sigma, \Pi \rangle$, where Σ is a signature and Π is a set of basic regular rules and basic cr-rules.*

Given a basic CR-Prolog program, Π, the *regular part* of Π is the set of its basic regular rules, and is denoted by $reg(\Pi)$. The set of basic cr-rules of Π is denoted by $cr(\Pi)$.

Example 7.1.1. *Consider the following program:*

$$\begin{aligned}&r_1 : p \text{ OR } q \overset{+}{\leftarrow} not\ r.\\&s.\end{aligned}$$

The regular part of the program, consisting of fact s, is consistent. Hence, the is no reason to apply the cr-rule, and the agent is only forced to believe s.

Example 7.1.2. *Now consider the program:*

$$\begin{aligned}&r_1 : p \text{ OR } q \overset{+}{\leftarrow} not\ r.\\&s.\\&\leftarrow not\ p, not\ q.\end{aligned}$$

This time, the regular part of the program is inconsistent. The cr-rule can be applied to restore consistency, and the agent is forced to believe either $\{s, p\}$ or $\{s, q\}$.

It is also possible to have cases when different cr-rules can be applied, like in the following example.

Example 7.1.3.

$$\begin{aligned}&r_1 : p \overset{+}{\leftarrow} not\ r.\\&r_2 : q \overset{+}{\leftarrow} not\ r.\\&s.\\&\leftarrow not\ p, not\ q.\end{aligned}$$

Again, the regular part of the program is inconsistent. Consistency can be restored by applying either r_1 or r_2, or both. Since cr-rules should be applied as little as possible, the last case is not considered. Hence, the agent is forced to believe either $\{s, p\}$ or $\{s, q\}$.

When different cr-rules are applicable, it is possible to specify preferences on which one should be applied by means of atoms of the form

$$prefer(r_1, r_2),$$

where r_1, r_2 are names of cr-rules. The atom informally says "do not consider solutions obtained using r_2 unless no solution can be found using r_1." The next example shows the effect of the introduction of preferences in the program from Example 7.1.3.

Example 7.1.4.

$$\begin{array}{l} r_1 : p \overset{+}{\leftarrow} not\ r. \\ r_2 : q \overset{+}{\leftarrow} not\ r. \\ prefer(r_1, r_2). \\ s. \\ \leftarrow not\ p, not\ q. \end{array}$$

The preference prevents the agent from applying r_2 unless no solution can be found using r_1. We have seen already that r_1 is sufficient to restore consistency. Hence, the agent has only one set of beliefs, $\{s, p\}$

Notice that our reading of the preference atom $prefer(r_1, r_2)$ rules out solutions in which r_1 and r_2 are applied simultaneously, as the use of r_2 is allowed only if no solution is obtained by applying r_1.

7.2 Semantics of basic CR-Prolog

In this section, we define the semantics of basic CR-Prolog. In the following discussion, Π_1 denotes a basic CR-Prolog program, Π_0 the regular part of Π_1, and R the cr-rules of Π_1. Also, for every $R' \subseteq R$, $\alpha(R')$ denotes the set of regular rules obtained from R' by replacing every connective $\overset{+}{\leftarrow}$ with $\leftarrow$. Notice that the regular part of any basic CR-Prolog program is a basic A-Prolog program.

The following definition defines the transitive closure of relation $prefer$, which will be used later in the definition of the semantics.

Definition 7.2.1. For every set of literals, S, from the signature of Π_1, and every r_1,r_2 from R, $pref_S(r_1, r_2)$ is true iff

$$prefer(r_1, r_2) \in S, \text{ or}$$

$$\exists r_3 \in R\ prefer(r_1, r_3) \in S \wedge pref_S(r_3, r_2).$$

The semantics of CR-Prolog is given in three steps. Intuitively, in the first step we look for combinations of cr-rules that restore consistency. Preferences are not considered, with the exception that solutions deriving from the simultaneous use of two cr-rules between which a preference exists are discarded.

Definition 7.2.2. $\langle S_1, R_1 \rangle$ is a *view* of Π_1 if:

1. S_1 is an answer set of $\Pi_0 \cup \alpha(R_1)$, and

2. for every r_1, r_2 such that $pref_{S_1}(r_1, r_2)$, $\{r_1, r_2\} \not\subseteq R_1$, and

3. for every $R_2 \subset R_1$, S_1 is not an answer set of $\Pi_0 \cup \alpha(R_2)$.

The second consists in selecting the best views with respect to the preferences specified. Notice that the definitions are made more complex by the fact if dynamic preferences are specified, different views can contain different preferences. The intuition here is that we consider only preferences on which there is agreement in the views under consideration.

Definition 7.2.3. For every pair of views of Π_1, $\langle S_1, R_1 \rangle$ and $\langle S_2, R_2 \rangle$, $\langle S_1, R_1 \rangle$ *dominates* $\langle S_2, R_2 \rangle$ if there exist r_1, r_2 such that $r_1 \in R_1$, $r_2 \in R_2$, and $pref_{S_1 \cap S_2}(r_1, r_2)$.

Definition 7.2.4. A view, $\langle S_1, R_1 \rangle$, is a candidate answer set of Π_1 if, for every view $\langle S_2, R_2 \rangle$ of Π_1, $\langle S_2, R_2 \rangle$ does not dominate $\langle S_1, R_1 \rangle$.

Finally, we select the candidate answer sets that are obtained by applying a minimal set of cr-rules.

Definition 7.2.5. A set of literals, S_1, is an *answer set* of Π_1 if:

1. there exists $R_1 \subseteq R$ such that $\langle S_1, R_1 \rangle$ is a candidate answer set of Π_1, and

2. for every candidate answer set, $\langle S_2, R_2 \rangle$, of Π_1, $R_2 \not\subset R_1$.

7.3 CR-Prolog

Following the approach used in Section 2.2, we extend basic CR-Prolog by allowing s-atoms in the head of both regular and cr-rules. We call the resulting language CR-Prolog.

Definition 7.3.1. *A* regular rule *of CR-Prolog is a statement of the form (7.1), where r and l_i's are as before, and either (1) $k = 1$ and h_1 is a s-atom, or (2) all h_i's are plain literals.*

Definition 7.3.2. *A* cr-rule *of CR-Prolog is a statement of the form (7.2), where r, l_i's and h_i's are like in Definition 7.3.1.*

Definition 7.3.3. *A* CR-Prolog program *is a pair $\langle \Sigma, \Pi \rangle$, where Σ is a signature and Π is a set of regular rules and cr-rules.*

As for basic CR-Prolog programs, $reg(\Pi)$ and $cr(\Pi)$ denote the regular part and the set of cr-rules of Π. Notice that the regular part of a CR-Prolog program is an A-Prolog program (rather than a basic A-Prolog program).

The semantics of CR-Prolog is a straightforward extension of the semantics of basic CR-Prolog, obtained by considering program $\Pi_0 \cup \alpha(R_1)$ in Definition 7.2.2 an A-Prolog program, rather than a basic A-Prolog program.

CHAPTER VIII
CR-PROLOG BASED REASONING ALGORITHMS

8.1 CR-Prolog and the Selection of Best Plans

In several cases, “best” plans are selected based on criteria different from just the minimization of their length. An interesting case is when we are given a set of requirements that plans should satisfy *if at all possible* (e.g., “if at all possible, do not skip lunch”). Such requirements are referred to as *soft requirements* [2]. In our approach, the satisfaction of soft requirements is checked for in the test phase of the search.

To the best of our knowledge, there is no general, elegant way to encode soft requirements using A-Prolog. For this reason, their encoding will be based on cr-rules and preferences statements of CR-Prolog.

In its simplest form, a soft requirement is encoded by a constraint and a cr-rule. The body of the constraint contains:

- the encoding of the condition that plans should satisfy, according to the soft requirement; the encoding is such that, if the requirement is not met, the body of the constraint is *satisfied*;
- a condition (the *inhibitor*) that allows to stop the application of the constraint, in case the soft requirement has to be violated.

For example, a possible constraint for the soft requirement “if at all possible, do not skip lunch” is:

$$\leftarrow skip(lunch), \text{not } allowed(skip(lunch)).$$

which informally says that it is *normally* impossible to skip lunch.

The cr-rule is used to say that, under some conditions, the constraint *may possibly* be inhibited, but its inhibition should be a rare occurrence. The cr-rule for the soft requirement above is:

$$allowed(skip(lunch)) \overset{+}{\leftarrow} .$$

which intuitively says that one may be possibly allowed to skip lunch.

If plans exist that do not violate the requirement, the cr-rule is not used. However, if no such plan exists, the cr-rule is used to conclude that skipping lunch is allowed. This inhibits the constraint, and allows the computation of plans violating the requirement.

For another example, consider the encoding of the soft requirement "it is normally impossible to skip lunch; however, if you had a big breakfast, you may possibly be allowed to skip it," which consists of the rules:

$$\begin{array}{l} \leftarrow \mathit{skip}(\mathit{lunch}), \text{not } \mathit{allowed}(\mathit{skip}(\mathit{lunch})). \\ \mathit{allowed}(\mathit{skip}(\mathit{lunch})) \stackrel{+}{\leftarrow} \mathit{had}(\mathit{big\_breakfast}). \end{array}$$

The cr-rule informally says that, if one had a big breakfast, he may possibly be allowed to skip lunch.

When several soft requirements are specified, one is often interested in ranking them in order of preference, so that the most preferred soft requirements are the ones that are less likely to be violated. Preferences statements of CR-Prolog provide a convenient way to encode such preferences. For example, consider the two soft requirements:

- if at all possible, do not skip lunch;
- if at all possible, do not skip dinner;

together with the preference "skipping lunch is preferred over skipping dinner." The soft requirements can be encoded as before:

$$\begin{array}{rl} & \leftarrow \mathit{skip}(\mathit{lunch}), \text{not } \mathit{allowed}(\mathit{skip}(\mathit{lunch})). \\ \mathit{skip}_l: & \mathit{allowed}(\mathit{skip}(\mathit{lunch})) \stackrel{+}{\leftarrow} . \\ & \\ & \leftarrow \mathit{skip}(\mathit{dinner}), \text{not } \mathit{allowed}(\mathit{skip}(\mathit{dinner})). \\ \mathit{skip}_d: & \mathit{allowed}(\mathit{skip}(\mathit{dinner})) \stackrel{+}{\leftarrow} . \end{array}$$

The preference is encoded by the following rule:

$$prefer(skip_l, skip_d).$$

which says that (if one has to skip either dinner or lunch) skipping dinner should be considered only if skipping lunch is not possible. It is important to stress that preference statements of CR-Prolog allow to encode more complex criteria than the one above, e.g. dynamic preferences such as "if you had a big breakfast, it is better for you to skip lunch than skipping dinner; otherwise, skipping dinner is preferred." Such preference can be encoded in CR-Prolog with the rules:

$$prefer(skip_l, skip_d) \leftarrow had(big\_breakfast).$$
$$prefer(skip_d, skip_l) \leftarrow \text{not } had(big\_breakfast).$$

The algorithm for the planning component that can deal with soft requirements is based on the observation that computing the plans (with k compound agent actions) satisfying a set of soft requirements can be reduced to finding the answer sets of the program:

$$\begin{aligned}&Plan_1(D, g, k) = \\ &\qquad \alpha(D) \cup AGEN(k) \cup GOALTEST(g, k) \cup SOFTREQ\end{aligned}$$

where $SOFTREQ$ is the CR-Prolog encoding of the soft requirements.

Soft requirements and preferences over them have an immediate application in USA-Advisor (see Section 6.1.1). For example, the left and right subsystems of the RCS are actually connected by the so-called *crossfeed* – a sequence of pipes connecting the plumbing of the two subsystems. The crossfeed is valve-controlled, and is intended to be used when one of the two subsystems is affected by faults preventing the use of the propellant from its own tanks. It is NASA's policy to use the crossfeed as sparingly as possible, to keep the level of propellant in the two subsystems balanced. This policy can of course be seen as a soft requirement, "avoid the use of the crossfeed if at all possible." Another example of the use of soft requirements for USA-Advisor is the encoding of the policy that "computer commands should be avoided if at all possible." (This policy is motivated by the fact that, normally, issuing a computer command requires preparing and uploading a patch of the software of the on-board computer.) The CR-Prolog encoding of the two soft requirements is:

$$r_{xf}(R,T) : allowed(xfeed(R,T)) \overset{+}{\leftarrow} subsystem(R).$$

$$\begin{aligned} \leftarrow\ & subsystem(R), action\_of(A,R), \\ & occurs(A,T), \\ & opens\_xfeed\_valve(A), \\ & \text{not } allowed(xfeed(R,T)). \end{aligned}$$

$$r_{ccs}(R,T) : allowed(ccs(R,T)) \overset{+}{\leftarrow} subsystem(R).$$

$$\begin{aligned} \leftarrow\ & subsystem(R), action\_of(A,R), \\ & occur(A,T), \\ & sends\_computer\_command(A), \\ & \text{not } allowed(ccs(R,T)). \end{aligned}$$

The first cr-rule says that the use of the crossfeed may possibly be allowed at any time step T. The corresponding constraint says that it is impossible for action A of subsystem R to occur at T if A opens a crossfeed valve, and the use of the crossfeed is not allowed in R at time step T. The second cr-rule says that computer commands may possibly be allowed at any time step T. The constraint says that it is impossible for action A of subsystem R to occur at T if A sends a computer command and computer commands are not allowed in R at time step T.

It is of course possible to state preferences between the two soft requirements. For example, if the flight controllers decide that modifying the software of the Shuttle's computer is preferable to losing the balance of the propellant between the left and right subsystems, the following rule can be added to the planner:

$$prefer(r_{ccs}(R2,T2), r_{xf}(R1,T1)).$$

As we showed earlier, it is also possible to express dynamic preferences. For example, the rules:

$$prefer(r_{ccs}(R2,T2), r_{xf}(R1,T1)) \leftarrow computer\_reliable.$$
$$prefer(r_{xf}(R1,T1), r_{ccs}(R2,T2)) \leftarrow \neg computer\_reliable.$$

say that the use of computer commands is preferred to the use of the crossfeed only if the on-board computer is reliable; if the computer is unreliable, instead, the preference is reversed.

Soft requirements can also be used to avoid the generation of irrelevant actions, typical of planning domains in which the goal is divided in independent subgoals, and the execution of parallel actions is allowed. Consider what happens in USA-Advisor if the goal requires that some jets in the forward and left subsystems be set ready to fire, and achieving the subgoal for the forward subsystem takes n_f steps, while achieving the subgoal for the left subsystem takes n_l steps, with $n_f < n_l$. By inspecting the selection rule used in the planning module, one can see that a plan in which the subgoal for the forward subsystem is achieved at step n_f is considered equivalent to one in which the same subgoal is achieved at $n_f + 1$. For this reason, a plan in which an extra, irrelevant action is performed on the forward subsystem at $n' < n_f + 1$ is as likely to be returned as the plan that achieves the subgoal at step n_f. Even using algorithm PC_0 does not help in this case, because the subgoal for the left subsystem forces the shortest plan to contain n_l steps.

Soft requirements can help to avoid the generation of irrelevant actions, so that, if a plan of length $n_f + 1$ is generated for the forward subsystem, at least it is possible to guarantee that no extra action will occur at step n'. The soft requirement to avoid irrelevant actions states that "performing actions should be avoided if at all possible.", and is encoded by the rules:

$$r_{short}(R,T) : allowed(execute\_action(R,T)) \overset{+}{\leftarrow} subsystem(R).$$

$$\begin{aligned}\leftarrow\ & subsystem(R),\\ & action\_of(A,R),\\ & occurs(A,T),\\ & \text{not } allowed(execute\_action(R,T)).\end{aligned}$$

The cr-rule says that, at any step T of the plan for subsystem R, the agent may be possibly allowed to perform actions. The constraint says that it is impossible for action A of subsystem R to occur at step T if the agent is not allowed to execute actions on subsystem R at step T.

Experimental results confirm that the plans generated by the extended version of USA-Advisor (called USA-Smart) are of a significantly higher quality than the plans generated by USA-Advisor even without the introduction of preferences.

We have applied USA-Smart to 800 problem instances from [57], namely the instances with 3, 5, 8, and 10 mechanical faults, respectively, and no electrical faults. (For these experiments, we did not include in the planner the preference statements on crossfeed and computer commands.)

The planning algorithm iteratively invokes the reasoning system with maximum plan length L, checks if a model is returned, and iterates after incrementing L if no model was found. If no plans are found that are 10 or less time steps long, the algorithm terminates and returns no solution. This approach guarantees that plans found by the algorithm are the shortest (in term of number of time steps between the first and the last action in the plan). Notice that the current implementation of CR-Prolog's inference engine returns the models ordered by the number of (ground) cr-rules used to obtain the model, with the model that uses the least cr-rules returned first. Hence, the plan returned by the algorithm is both the shortest and the one that uses the minimum number of cr-rules.

Overall, computer commands were used 27 times, as opposed to 1831 computer commands generated by USA-Advisor. The crossfeed was used 10 times by USA-Smart, and 187 times by USA-Advisor. Moreover, in 327 cases over 800, USA-Smart generated plans that contained less actions than the plans found by USA-Advisor (as expected, in no occasion they were longer). The total number of irrelevant actions avoided by USA-Smart was 577, which is about 12% of the total number of actions used by USA-Advisor (4601).

In spite of the improvement in the quality of plans, the time required by USA-Smart to compute a plan (or prove the absence of a solution) was still largely acceptable. Many plans were found in seconds; most were found in less than 2 minutes, and the program almost always returned an answer in less than 20 minutes (the maximum that the Shuttle experts consider acceptable). The only exception consists of about 10 cases, when planning took a few hours. These outliers were most likely due to the fact that the inference engine for CR-Prolog is still somewhat unoptimized.

8.2 The Selection of Best Diagnoses

As pointed out in Section 6.2.1, the A-Prolog based diagnostic algorithms presented earlier always return reasonable diagnoses, but often find too many of them.

To narrow the search to "best" diagnoses, we have developed a technique based on CR-Prolog that allows the specification of preferences among diagnoses. We couldn't find any general, elegant way to achieve the same result with A-Prolog alone.

The technique consists in viewing exogenous actions as rare events, and using cr-rules to specify that the exogenous actions can occur, although rarely. For example, going back to the electrical circuit from Section 6.2.1, the fact that a power surge may possibly occur at any time is encoded by the cr-rule:

$$r(srg,T):\ o(srg,T) \overset{+}{\leftarrow} .$$

Of course the body of the cr-rule needn't be empty. If we are given information that power surges occur only during storms, we can modify the cr-rule accordingly:

$$r(srg, T): \ o(srg, T) \stackrel{+}{\leftarrow} h(storm, T).$$

The new rule says that a power surge may possibly occur whenever there is a storm, although this is a rare event.

It is worth stressing that, even without introducing preferences, encoding exogenous actions with CR-Prolog yields a substantial improvement in the quality of diagnoses. In fact, the semantics of CR-Prolog is such that only minimal diagnoses (in a set-theoretical sense) are found. We will come back to this topic later, when we consider the advantages of using CR-Prolog over alternative approaches.

The selection of "best" diagnoses is achieved by specifying the relative likelihood of exogenous actions using preferences on the corresponding cr-rules. For example, the information that bulb blow-ups and surges are exogenous actions and that blow-ups are more likely than power surges can be encoded by the rules:

$$\left\{\begin{array}{l} r(srg, T) : o(srg, T) \stackrel{+}{\leftarrow} . \\ r(brk, T) : o(brk, T) \stackrel{+}{\leftarrow} . \\ \\ \text{\% blow-ups are more likely than surges at any time step.} \\ more\_likely(brk, srg, T). \\ \\ prefer(r(A_1, T), r(A_2, T)) \leftarrow \ more\_likely(A_1, A_2, T). \end{array}\right.$$

The first two rules specify the available exogenous actions. The third rule encodes the relative likelihood of the two actions. The last rule states that, if A_1 is more likely than A_2, diagnoses containing the occurrence of A_2 at step T must not be considered if diagnoses exist that contain A_1 at step T."

As shown in the previous example, the availability of dynamic preferences allows the encoding of rather complex likelihood relations. In the next example, we formalize the fact that:

$$\begin{array}{c}\textit{“Power surges are more likely than blow-ups during storms,} \\ \textit{but less likely otherwise.”}\end{array} \tag{8.1}$$

The second part of statement (8.1) is encoded by a rather standard default, together with an axiom stating that likelihood is an anti-symmetric relation; the first part of (8.1) is encoded as a strong exception to the default.

$$\left\{\begin{array}{ll}
\multicolumn{2}{l}{\%\%\ \textit{normally}\text{, blow-ups are more likely than power surges.}} \\
\mathit{more\_likely}(brk, srg, T) \leftarrow & \text{not } \mathit{exception}(d(brk, srg, T)), \\
& \text{not } \neg \mathit{more\_likely}(brk, srg, T). \\
& \\
\multicolumn{2}{l}{\%\%\ \text{relative likelihood is an anti-symmetric relation}} \\
\neg \mathit{more\_likely}(A_2, A_1, T) \leftarrow & \mathit{more\_likely}(A_1, A_2, T). \\
& \\
\multicolumn{2}{l}{\%\%\ \text{surges are more likely than blow-ups during storms.}} \\
\mathit{more\_likely}(srg, brk, T) \leftarrow & h(\mathit{storm}, T).
\end{array}\right.$$

It is worth stressing that such a formalization is made possible by the use of the dynamic preference:

$$\mathit{prefer}(r(A_1, T), r(A_2, T)) \leftarrow \ \mathit{more\_likely}(A_1, A_2, T).$$

The use of cr-rules and preferences can be easily integrated into the algorithms presented in Section 6.2.1. Recall that, there, the computation of candidate diagnoses of symptom $\mathcal{S}$ was reduced to finding the answer sets of diagnostic program (6.5):

$$D_0(\mathcal{S}) = \mathit{Conf}(\mathcal{S}) \cup DM_0.$$

Let us now denote by DM_{cr} the set of rules specifying exogenous actions and their relative likelihood. Then, candidate diagnoses of symptom $\mathcal{S}$ can be computed by means of the diagnostic program:

$$D_{cr}(\mathcal{S}) = \mathit{Conf}(\mathcal{S}) \cup DM_{cr}. \tag{8.2}$$

The main advantage of this approach is the substantial increase in quality of the candidate diagnoses and diagnoses.

Let us re-consider the computation of candidate diagnoses. Intuitively, it seems natural that a rational agent should at first consider only minimal candidate diagnoses.

However, program $D_0(\mathcal{S})$ does not allow to give any preference to such diagnoses. Computing minimal candidate diagnoses in A-Prolog appears to require non-trivial modifications of the *procedural* part of the reasoning algorithm. This is in contrast with our policy that the relevant part of the reasoning process should be performed by *declarative* means, with the procedural code being used as "glue."

On the other hand, because of the semantics of CR-Prolog, $D_{cr}(\mathcal{S})$ automatically finds minimal candidate diagnoses, without the need for additions to the procedural code. This allows the programmer to focus on the really important knowledge representation issues.

The use of preferences allows for a further improvement of the quality of diagnoses. In the example above, we have shown that the availability of dynamic preferences allows for substantially more flexible reasoning. Overall, the introduction of CR-Prolog in diagnosis results in an agent control loop that is entirely declarative and yet computes "best" diagnoses. Notice that, since the preference relation is transitive and anti-symmetric, normally preference specifications are reasonably compact.

Although other recent extensions of A-Prolog can be used to improve the quality of diagnoses, it is not clear whether they yield the same level of improvement. For example, DLV's weak constraints [17, 18] appear to yield unintuitive results when preferences over exogenous actions are combined with uncertainty about the initial situation. Ordered disjunction [14, 15, 16] appears to have problems with certain types of dynamic preferences, as well as to sometimes produce unintuitive results when conflicting preferences are present in the program. We will show later a comparison with representative approaches from the literature.

Our CR-Prolog based diagnostic algorithms have been tested on the RCS model from USA-Advisor with good results. The diagnostic module included the specification of 231 exogenous actions (to simplify the modeling, actions are allowed to occur only at the first time step). Candidate diagnoses for various diagnostic problems were computed in a few seconds. We also tested the consequences of adding information on the relative likelihood of the exogenous actions. We used collections of statements such as:

% It is more likely for a valve to leak than to be stuck.
%
$more\_likely(leak(Valve), stuck(Valve), T).$

% If switch S controls valve V, it is more likely for V
% to be stuck than for S to be stuck.
$more\_likely(stuck(Valve), stuck(Switch), T) \leftarrow controls(Switch, Valve).$

Again, a candidate diagnosis was usually found in less than 15 seconds, in spite of the large number of ground instances of the preference statements.

8.3 The Selection of Best Corrections

The introduction of CR-Prolog in the learning modules proceeds along the same lines of diagnosis. The fact that a precondition or a law is missing is a modeled as a rare event, and encoded in CR-Prolog as follows. These cr-rules are intended to replace the corresponding selection rules.

$$\left\{\begin{array}{l}
\%\text{ Any fresh constant } L \text{ can denote a law that}\\
\%\text{ is missing from the action description.}\\
r(\mathit{missing\_law}(L)) : \mathit{missing\_law}(L) \overset{+}{\leftarrow} \mathit{new\_law\_name}(L).\\
\\
\%\text{ Any AL-literal } \mathit{LNAME} \text{ can be possibly missing from}\\
\%\text{ the body of any law } L.\\
r(\mathit{missing\_prec}(\mathit{LNAME}, N, L)) :\\
\quad \mathit{missing\_prec}(\mathit{LNAME}, N, L) \overset{+}{\leftarrow} \mathit{law}(L),\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{prec\_name\_and\_pos}(\mathit{LNAME}, N, L).
\end{array}\right.$$

Using cr-rules instead of selection rules for the selection process substantially limits the number of corrections found, as by the semantics of CR-Prolog only set-theoretically minimal (w.r.t. the cr-rules used) corrections are returned.

To further improve the quality of the solutions returned, preferences can be specified on the application of cr-rules. For example, consider the rule:

$$\begin{array}{l}
\mathit{prefer}(r(\mathit{missing\_prec}(PN1, N1, L)),\\
\qquad\qquad r(\mathit{missing\_prec}(PN2, N2, L))) \leftarrow \mathit{static}(PN1), \text{not } \mathit{static}(PN2).
\end{array}$$

where not $\mathit{static}(PN2)$ is used as an abbreviation of the closed world assumption on relation *static*. The rule says that fluent literals should not be hypothesized to be missing from L unless no solution can be found by considering only missing statics in L. This corresponds to the intuition that, in specializing L, one may want to ground its variables rather than adding new fluent literals to its body. Notice that the fact that CR-Prolog can be used to express dynamic preferences is quite important here. In fact, criteria like the one above are typically used only under some condition, e.g. if the ratio of fluent literal and static preconditions in L is above some threshold.

As for diagnosis, cr-rules and preferences can be easily integrated into the algorithms presented in Section 6.2.2. Let LM_{cr} be obtained from learning module LM_1 by replacing the selection rules that select $\mathit{missing\_prec}(\mathit{LNAME}, N, L)$ and

$missing\_law(L)$ with the cr-rules above. Let also LM_{cr} contain a set of preference statements (including any auxiliary definition). Then, candidate corrections of symptom $\mathcal{S}$ can be computed by the learning program:

$$L_{cr}(AD, \mathcal{S}, n) = \chi(AD, n) \cup H^n \cup O_n^{cT} \cup R \cup LM_{cr}.$$

CHAPTER IX
RELATED WORK

9.1 Agent Architectures and Execution Monitoring

An interesting approach for the construction of rational agents is the *Golog approach*, described in [70, 71]. There, the domain model is axiomatized using situation calculus [64], a knowledge representation framework formulated in the classical predicate logic. The agent's behavior, with respect to the selection of the actions that achieve the goal, is determined by programs written in Golog, an agent programming language based on situation calculus. Golog programs can be seen as defining complex actions in terms of the set of primitive actions from the axiomatization of the domain. For example, the following Golog program defines the complex action $makeOneTower(z)$, which creates a single tower of blocks on top of block z (we assume that primitive actions $startMove$ and $endMove$ are part of the situation calculus formalization of the domain). Symbol π denotes non-deterministic assignment of values to the variables in its scope.

```
**proc** $makeOneTower(z)$
    $\neg(\exists y).y \neq z \wedge clear(y)? \mid$
    $(\pi\ x, t)[startMove(x, z, t)\ ;$
        $(\pi\ t')endMove(x, z, t')\ ;\ makeOneTower(x)]$
**endProc**
```

Typically, the goal is defined as the successful execution of a complex action. Planning is reduced to finding a constructive proof that the complex action can be executed from the initial situation. For example, plans that achieve the goal of building one tower on top of block z are found by proving:

$$(\exists s)Do(makeOneTower(z), S_0, s)$$

where S_0 is the term associated with the initial situation and $Do(\delta, s, s')$ is a relation whose intuitive meaning is that s' is one of the situations reached by evaluating the program δ beginning in situation s.

The agent loop in the Golog approach can be summarized as follows:

Input: a Golog program, δ, to be executed

Steps:

1. observe the environment;
2. analyze any unexpected observations and modify δ if necessary;
3. execute the next primitive action, as prescribed by δ.

The structure of the loop stresses the central role of the Golog program: the agent is in some sense "pre-programmed", as opposed to the "fully deliberative" agent described in this dissertation.

The choice of knowledge representation formalism adopted to axiomatize the domain substantially influences the class of domains that the agent can deal with. In this respect, there are several important aspects in which A-Prolog appears to differ from situation calculus. First of all, to the best of our knowledge, in situation calculus it is not possible to encode dependencies among fluents that cause non-determinism. As an example, consider the following action description:

$$s_1: \text{ caused } p \text{ if } r, \neg q.$$
$$s_2: \text{ caused } q \text{ if } r, \neg p.$$

$$\{a\} \text{ causes } r.$$

The execution of a in state $\{\neg r, \neg p, \neg q\}$, yields two possible successor states: $\{r, p, \neg q\}$ and $\{r, \neg p, q\}$. In the first case, $\neg q$ stays true by inertia, and law s_1 forces p to become true. The second case is symmetrical to the first, with $\neg p$ staying true by inertia.

The second important difference between A-Prolog and situation calculus is that the solution to the frame problem adopted in the latter formalism makes it difficult to encode *recursive definitions.* For instance, situation calculus is not suitable for the encoding of relation *pressurized_by*, presented in the description of the Plumbing Module of USA-Advisor in Section 6.1.1.

The third important difference is that A-Prolog is non-monotonic, while situation calculus is monotonic. Although monotonic logic has the advantage of being the best studied logic formalism, non-monotonic logics like A-Prolog typically provide for greater elaboration tolerance and easier updates.

The encoding of control knowledge also substantially differs between the two approaches. In the Golog approach, control knowledge is encoded in Golog, which is essentially a *procedural language*. In A-Prolog, control knowledge is expressed *declaratively*, typically in form of constraints.

Moreover, it is not clear whether the agent loop in the Golog approach can work at all if no control knowledge is given, i.e. if the Golog program consists only of a test that checks whether a particular situation has been reached. The A-Prolog based agent loop is designed to accept any amount of control knowledge, from no control knowledge to very detailed, Golog-style control knowledge.

Finally, there is also a difference with respect to the type of observations gathered by the agent, and how they are used in the agent loop. In our approach, observations are statements on whether fluents are true or false at the current moment of time. The agent deals with unexpected observations by explaining them, i.e. by determining which exogenous actions may have caused them, and by re-planning accordingly. In the Golog approach, observations consist either of the occurrence of exogenous actions or of tests on conditions (it seems that the latter are similar to our view of observations). Given the current observations, the agent finds a corrected Golog program that allow the achievement of the goal. No attempt to *explain* the observations is made. We believe that explaining the observations allows the agent to more successfully interact with the environment, both in the current time step and in the future, by exploiting knowledge on the *causes* of the observations. Also notice that, when occurrences of exogenous actions (i.e. explanations, in the sense of the term used in this dissertation) are directly observed by the Golog agent, the algorithm only appears to be designed to accept observations on actions occurred after the latest agent action. This differs from our approach, in which the agent can deal with the occurrence of exogenous actions at any time in the past.

An approach to monitoring the execution of actions in intelligent agents is described in [26, 27, 28], where a tool called KMONITOR is introduced.

KMONITOR is a tool for monitoring plan execution in non-deterministic environments encoded in action language $\mathcal{K}$. Intuitively, KMONITOR monitors the execution of a plan in a domain with non-deterministic actions and ensures that the agent is moving along one of a set of *preferred trajectories*. To keep monitoring overhead low, the agent observes the environment only at certain *checkpoints*. Checkpoints are computed based on a checkpoints policy, specified by a logic program.

When discrepancies between the observed and expected evolution of the environment are detected, the tool determines the step at which the trajectories separated and applies execution recovery techniques, such as plan reversal.

The KMONITOR project has been so far focused on execution monitoring and recovery. It is not clear which parts of our loop the agent architecture includes.

The approach differs from ours in many respects. First of all, it is focused on non-deterministic transition diagrams, while we concentrate on deterministic action descriptions. As a consequence, KMONITOR deals with *possible plans* rather than plans, where by possible plans we mean sequences of actions that *may* achieve the goal, but may as well fail, depending on the actual effects of the actions.

Also, KMONITOR does not allow interferences of the environment by means of exogenous actions. Only "unwanted" observations deriving from the non-deterministic effects of actions are considered. That appears to limit the applicability of the approach in situations in which unexpected observations are caused by physical phenomena or by other agents. This view results in a type of diagnostic reasoning quite different from ours. In KMONITOR, diagnosing a discrepancy means finding a *point of failure* that justifies it, where a point of failure is intuitively a branching point in the transition diagram at which the observed trajectory and the expected one possibly separated.

Finally, the KMONITOR approach does not describe how preferred trajectories are specified, while our approach includes a methodology for the encoding of soft requirements. It is interesting to see if CR-Prolog or prioritized default theories from [60] can be adopted to specify preferred trajectories in KMONITOR.

Other agent architectures are derived from the research on robotics, such as 3T and MDS [12, 24, 55]. All such architectures are characterized by somewhat less formal foundations, and usually by the lack of a precise definition of state. Our agents appear to be substantially more deliberative than theirs. In most of these architectures, intelligent agent behavior emerges from the interaction of relatively simple software components that react to the observations on the current state by performing actions according to a fixed mapping. The components usually maintain either a rather limited history and model of the domain or even none at all. On the other hand, these architectures are quite successful in the interaction with physical environments by means of (even relatively unreliable) sensors and actuators. Although the resulting behavior is often surprisingly intelligent – especially considered the simplicity of the software components – we do not see how truly deliberative behavior can be achieved just by the interaction of non-deliberative components.

9.2 Conditions for Determinism of Action Descriptions

[9] introduces a condition for the determinism of action descriptions that extends the one from [8]. The new condition is based on the notion of *separability* of an action description, defined as follows (in this section, the term action denotes singleton compound actions).

Let R be the collection of the state constraints of action description AD. For any action a and state σ, $E^*(a, \sigma)$ denotes the closure of the direct effects of a in σ with respect to the laws in R.

Action description AD is *separable* if, for any a and σ such that a is executable in σ, if $r \in R$ and $body(r) \cap E^*(a, \sigma) \neq \emptyset$, then $body(r) \subseteq E^*(a, \sigma)$.

Proposition 13.10.1 in [9] states that "any separable action description is deterministic."

This condition for checking the determinism of action descriptions differs from ours in two respects. First of all, algorithms that check the condition are likely to be more computationally complex than those presented in Chapter V. In fact, although no algorithm is described in [9], the condition appears to involve tests on all states of the transition diagram (as well as on the executability of the actions). It is unlikely that this can be done in polynomial time. On the other hand, we have proven that our test for the safety of $dep(AD)$ has polynomial complexity.

Furthermore, as the following examples show, the test from [9] appears to miss action descriptions that our tests correctly identify as deterministic.

Example 9.2.1. *Consider the following action description:*

$$w_1 : q \textit{ if } \neg r, \neg p.$$
$$w_2 : r \textit{ if } \neg q, p.$$
$$a \textit{ causes } p.$$

From Chapter V, we know that this action description is deterministic, and that it can be correctly identified using dependency graphs (but not with simplified dependency graphs).

Now, let us check if the action description is separable. Let us consider state $\sigma = \{p, q, r\}$ *and action* a*. Set* $E^*(a, \sigma)$ *is* $\{p\}$*, and*

$$E^*(a, \sigma) \cap body(w_2) \neq \emptyset \quad \text{BUT} \quad body(w_2) \not\subseteq E^*(a, \sigma).$$

Since the action description is not separable, it is not identified as deterministic.

The next example shows that the condition from [9] is even stronger than our condition on the simplified dependency graph.

Example 9.2.2. *Consider the following action description.*

$$w_1 : p \textit{ if } q, s.$$
$$w_2 : t \textit{ if } s.$$
$$a \textit{ causes } s.$$

Obviously the action description is deterministic, and it can be identified as such by testing either the dependency graph or the simplified dependency graph.

On the other hand, consider state $\sigma = \{\neg s, \neg t, \neg p, \neg q\}$ *and action* a. *We obtain* $E^*(a, \sigma) = \{s, t\}$. *Therefore,*

$$E^*(a, \sigma) \cap body(w_1) \neq \emptyset \quad \text{BUT} \quad body(w_1) \not\subseteq E^*(a, \sigma),$$

and the action description cannot be identified as deterministic by the separability test.

9.3 Planning

Our work on planning extends the study from [57] by embedding those planning algorithms in our agent architecture. We also introduced the use of CR-Prolog in planning, and in particular the use of soft requirements.

Our use of CR-Prolog to encode soft requirements (see Chapter VIII) presents similarities with the use of prioritized default theories in [60]. There, the authors encode preferences by means of prioritized default theories. The language used allows the specification of preferences both on actions and final states.

The main difference between their approach and ours lies in fact that CR-Prolog is a general tool that allows the encoding of preferences of various types, while their preferences are designed only to be used for planning with action languages. The wider applicability of CR-Prolog is demonstrated by the fact that cr-rules and preferences over them have been used in this dissertation to encode both soft requirements, relative likelihood of exogenous actions, and preferences on the different ways to specialize laws in the learning process.

9.4 Diagnosis

There is a numerous collection of papers on diagnosis many of which substantially influenced our views on the subject. The roots of our approach go back to [62] where diagnosis for a static environment were formally defined in logical terms. To the best of our knowledge the first published extensions of this work to dynamic domains appeared in [73], where dynamic domains were described in fluent calculus [75], and in [51] which used situation calculus [37]. Explanation of malfunctioning of system components in terms of unobserved exogenous actions was first clearly articulated in [52]. Generalization and extensions of these ideas [11] which specifies dynamic domains in action language $\mathcal{L}$, can be viewed as a starting point of the work presented in this dissertation. The use of a simpler action language $\mathcal{AL}$ allowed us to substantially simplify the basic definitions of [11] and to reduce the computation of diagnosis to finding stable models of logic programs. As a result we were able to incorporate diagnostic reasoning in a general agent architecture based on the answer set programming paradigm, and to combine diagnostics with planning and other activities of a reasoning agent. On another hand [11] addresses some questions which are not fully addressed by our work. In particular, the underlying action language of [11] allows non-deterministic and knowledge-producing actions absent in our work. While our formulation allows immediate incorporation of the former, incorporation of the latter seems to substantially increase conceptual complexity of the formalism. This is of course the case in [11] too but we believe that the need for such increase in complexity remains an open question. Another interesting related work is [58]. In this paper the authors address the problem of dynamic diagnosis using the notion of pertinence logic from [19]. The formalism allows to define dynamic diagnosis which, among other things, can model intermittent faults of the system. As a result it provides a logical account of the following scenario: Consider a person trying to shoot a turkey. Suppose that the gun is initially loaded, the agent shoots, observes that the turkey is not dead, and shoots one more time. Now the turkey is dead. The pertinence formalism of [58] does not claim inconsistency - it properly determines that the gun has an intermittent fault. Our formalism on another hand is not capable of modeling this scenario - to do that we need to introduce non-deterministic actions. Since, in our opinion, the use of pertinence logic substantially complicates action formalisms it is interesting to see if such use for reasoning with intermittent faults can always be avoided by introducing non-determinism. Additional comparison of the action languages based approach to diagnosis with other related approaches can be found in [11].

A work relevant to our research on diagnosis with CR-Prolog is the introduction of *weak constraints* in DLV [17, 18, 20]. Intuitively, a weak constraint is a constraint that can be violated, if this is needed to obtain an answer set of a program. To each weak constraint, a *weight* is assigned, indicating the cost of violating the constraint[1]. A preferred answer set of a program with weak constraints is one that minimizes the sum of the weights of the constraints that the answer set violates. Consider for example program Π_{dlv} of DLV:

$$\begin{cases} a \text{ OR } b. \\ :\sim a.\ [1:] \\ :\sim b.\ [2:] \end{cases}$$

where the first weak constraint (denoted by symbol :∼) has weight 1 and the second has weight 2. In order to satisfy the first rule, the answer sets of Π_{dlv} must violate one of the constraints. Since violating the first constraint has a lower cost than violating the second, the preferred answer set of Π_{dlv} is $\{a\}$.

Weak constraints are of course similar to our cr-rules and weights can often play the role of preferences. The main disadvantage of using weak constraints instead of cr-rules is that weights induce a total order on the weak constraints of the program, as opposed to the partial order that can be specified on cr-rules. This seems to be a key difference in the formalization of some forms of common-sense knowledge. Consider the following specification of relative likelihoods for exogenous actions *brk* and *srg*

[1]To be precise, two different "costs", weight and level, are assigned to weak constraints, but in our discussion we only consider the weight, since even levels do not seem to solve the problem.

from Section 8.2.

$$more\_likely(brk, srg, T) \leftarrow h(\neg storm, T).$$
$$more\_likely(srg, brk, T) \leftarrow h(storm, T).$$

and the following collection of observations, O_{dlv}:

$$\begin{cases} hpd(close(s_1), 0). \\ \%\text{ no information on whether } storm \text{ is true or false at } 0 \\ \\ obs(\neg on(b), 1). \\ obs(\neg ab(b), 1). \end{cases}$$

To the best of our knowledge, there is no formalization of this domain in DLV with weak constraints, that, given recorded history O_{dlv}, concludes that there are two possible alternatives compatible with O_{dlv}:

$$\{h(storm, 0), o(srg, 0)\}$$
$$\{h(\neg storm, 0), o(srg, 0)\}$$

To see the problem, consider, for example, the following translation of our diagnostic program in DLV[2],

$$D_{wk}(\mathcal{S}) = Conf(\mathcal{S}) \cup DM_{wk},$$

where DM_{wk} is:

$$\begin{cases} :\sim o(brk, T), h(storm, 0).\ [4:] \\ :\sim o(srg, T), h(storm, 0).\ [1:] \\ :\sim o(brk, T), \neg h(storm, 0).\ [1:] \\ :\sim o(srg, T), \neg h(storm, 0).\ [4:] \end{cases}$$

The first two weak constraints say that, if a storm occurred, assuming that action *brk* occurred has a cost of 4, while assuming that action *srg* occurred has a cost of

[2]To be precise, to be actually used with DLV D_{wk} would have to be modified so as to remove all function symbols. Although the process is not difficult, describing it is out of the scope of this dissertation and will not be discussed.

1. The last two weak constraints say that, if a storm did not occur, assuming that action brk occurred has a cost of 1, while assuming that action srg occurred has a cost of 4. The selection of particular weights is fairly arbitrary, but it captures the corresponding dynamic preferences.

The only possible explanation of recorded history O_{dlv} is the occurrence of srg at time 0. Hence, D_{wk} produces two candidate answer sets, containing, as expected, the two set of facts above. Unfortunately, the answer set corresponding to the second set of facts has a total cost of 4, while the answer set corresponding to the first explanation has a cost of 1. This forces the agent to prefer the first answer set, and to assume, without any sufficient reason, the existence of a storm. On the other hand, the corresponding CR-Prolog diagnostic program returns both intended answers.

There are however some classes of programs of CR-Prolog which can be reduced to DLV programs with weak constraints. Study of such classes may be useful not only for improving our understanding of both formalisms, but also for using the efficient computation engine of DLV for CR-Prolog computations.

9.5 Learning

To the best of our knowledge, ours is the first investigation of the use of A-Prolog and its extensions to perform inductive learning of action theories. Previous studies [47, 48], in fact, concentrated on the use of inductive logic programming (ILP) algorithms to learn action theories. Moreover, there, domain models were obtained by importing a situation calculus ontology and using default negation to solve the frame problem.

We believe that, in this context, using A-Prolog instead of ILP techniques has important advantages. In particular, when ILP is used, particular attention needs to be paid to the use of default negation in the action theory, as traditional ILP approaches are not well-suited for these applications. Because of the features of the semantics of A-Prolog, our approach is not affected by this problem.

Overall, our approach results in more declarative and compact learning modules.

This is demonstrated by the fact that, in $L_1(AD, \mathcal{S}, n)$, the search for corrections is essentially performed by means of only two selection rules.

Other researchers [65, 66, 59, 72] dealt with the problem of learning *A-Prolog rules*, rather than causal laws. From the point of view of the A-Prolog rules being added, those approaches result in allowing the addition of rules with virtually no syntactic restrictions. In this dissertation, we consider the simpler task of adding only facts to the program.

Differently from our work, in those approaches the generation of hypotheses occurs *outside* A-Prolog, using procedural, ILP-derived, techniques. It would be interesting to see if learning of arbitrary A-Prolog rules can be accomplished inside an A-Prolog program.

The use of a procedural approach for learning has the limitation that extensions of it are not as straightforward as when A-Prolog is used. As shown in Section 8.3, our approach can be naturally extended to allow the specification of preferences to selection of best corrections. It is not clear how the other approaches can be extended to obtain the same result.

Finally, to the best of our knowledge, all the approaches present in the literature are static, in the sense that the algorithms are not part of an agent architecture and (obviously) do not allow testing of the hypotheses by gathering further observations, like we do in $Find\_Correction$.

9.6 CR-Prolog

Programs of CR-Prolog closely resemble knowledge systems of [38] – pairs $\langle T, H \rangle$ of non-disjunctive programs in which T represents a background knowledge and H is a set of candidate hypotheses. Though syntactically and even semantically similar, the programming methodologies of these two approaches differ considerably. The background theory T of knowledge system seems to be either a collection of integrity constraints or a collection of defaults whose credibility is higher than that of H. This is quite different from the structuring of knowledge advocated in this dissertation. The use of rules from H differ depending on the use of the knowledge system. The emphasis seems to be on default reasoning, where hypothesis are interpreted as defaults and hence rules of H are fired whenever possible. This interferes with the search for explanations, which normally favors some form of minimality and applies the rules sparingly. There are some suggestions of using knowledge systems for this purpose by applying a different strategy for the selection of rules. In our opinion these two types of reasoning are not easily combined together. (In some cases they may even require a different representation of knowledge for each of the reasoning tasks.) The notion of knowledge system is further extended in [67] by introducing priorities over elements of H viewed as defaults. The new work does not seem to change the methodology of knowledge representation of the original paper. Consequently even our priority relations are quite different from each other.

In [14], the author introduces logic programs with ordered disjunction (LPOD). The semantics of LPOD is based on the notion of preferred answer sets. In later papers [15, 16], the authors introduce the notion of Pareto-preference and show that this criterion gives more intuitive results that the other criteria they introduced. For this reason, here we only consider LPOD under Pareto-preference.

In LPOD, rules are statements of the form:

$$h_1 \times h_2 \times \ldots \times h_k \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n$$

where h's and l's are literals. The intuitive meaning of such a rule is "whenever the body is satisfied, if possible believe h_1; if h_1 is not possible, believe h_2; ...; otherwise, believe h_n."

The semantics of LPOD is based on an assignment of numerical weights to the rules of the program, depending on which element of the head is believed (the "best" the element, the smallest the weight). A weight of 1 is associated to rules whose body is not satisfied. A *ceteris paribus* criterion [13] is used to propagate the preference for higher degrees of satisfaction to *preferred answer sets.*

Unfortunately, ordered disjunction appears to have problems with certain types of dynamic preferences, as well as to sometimes produce unintuitive results when conflicting preferences are present in the program.

The first type of problems appears to derive from the way weights are assigned to rules whose body is not satisfied, as pointed out by the following example.[3]

Example 9.6.1. *Consider the story:*
"John wants to go to the movies, if possible; otherwise, he will watch tv. If he goes to the movies then wants to have popcorn if possible; otherwise, he will have candy. Now popcorn is not available."

The story can be formalized in LPOD as follows:

$$\left\{\begin{array}{l} \text{\% John prefers to go to a movie over watching tv.} \\ movie \times tv. \\ \text{\% At the movies, he prefers eating popcorn over candy.} \\ popcorn \times candy \leftarrow movie. \\ \text{\% Popcorn is not available.} \\ \neg popcorn. \end{array}\right.$$

Intuitively, John should prefer going to the movies. Since popcorn is not available, he will eat candy. Watching tv seems a less acceptable option. Under the LPOD semantics, however, the above program has two preferred answer sets: $\{movie, candy\}$ *and* $\{tv\}$*, in contrast to the intuition.*

[3]We thank Richard Watson for noticing the problem with LPOD and suggesting the example.

A possible CR-Prolog formalization is:

$$\left\{\begin{array}{l}
\text{\% John either goes to a movie or watches tv.}\\
r_m : movie \overset{+}{\leftarrow} .\\
r_{tv} : tv \overset{+}{\leftarrow} .\\
\leftarrow not\ movie, not\ tv.\\
\\
\text{\% John prefers to go to a movie over watching tv.}\\
prefer(r_m, r_{tv}).\\
\\
\text{\% At the movie, John can eat either popcorn or candy.}\\
r_p : popcorn \overset{+}{\leftarrow} movie.\\
r_c : candy \overset{+}{\leftarrow} movie.\\
\text{\% ... and must eat one of them.}\\
\leftarrow not\ popcorn, not\ candy, movie.\\
\\
\text{\% At the movie, he prefers eating popcorn over candy.}\\
prefer(r_p, r_c) \leftarrow movie.\\
\\
\text{\% Popcorn is not available.}\\
\neg popcorn.
\end{array}\right.$$

This program has a unique answer set,

$$\{\neg popcorn, movie, candy\},$$

which corresponds to the intuition.

The other difficulty with LPOD is that unintuitive answers can be returned when there are conflicts among preferences. The issue derives from the inability of ceteris paribus preference to make a decision in presence of tradeoffs. Consider the following example from [13]:

- we have four possible choices, $a(1)$,$a(2)$,$b(1)$,$b(2)$;
- we must select one $a(i)$ and one $b(i)$;
- we prefer $a(1)$ to $a(2)$ and $b(1)$ to $b(2)$;
- $a(1)$, $b(1)$ cannot be selected together.

In such a situation, ceteris paribus preference allows to identify $\{a(1), b(2)\}$ and $\{a(2), b(1)\}$ as the best possible solutions, but does not allow to further select among them. Notice that there is a conflicting condition between the preferences: the first solution satisfies the first preference, but does not satisfy the second. The second solution behaves in the opposite way. In LPOD, in such situations, both solutions are considered valid.

The example can be formalized in LPOD as follows:

$$\begin{cases} a(1) \times a(2). \\ b(1) \times b(2). \\ \\ \leftarrow a(1), b(1). \end{cases}$$

As expected, the program has two preferred answer sets: $\{a(1), b(2)\}$ and $\{a(2), b(1)\}$.

On the other hand, in CR-Prolog such conflict situations are resolved by disregarding both solutions. Hence, the program:

$$\begin{cases} n(1).n(2). \\ ra(I) : a(I) \overset{+}{\leftarrow} n(I). \\ \leftarrow \text{not } a(1), \text{not } a(2). \\ \\ prefer(ra(1), ra(2)). \\ \\ rb(I) : b(I) \overset{+}{\leftarrow} n(I). \\ \leftarrow \text{not } b(1), \text{not } b(2). \\ \\ prefer(rb(1), rb(2)). \\ \\ \leftarrow a(1), b(1). \end{cases}$$

has no answer sets.

The difference between the two semantics depends on the fact that Pareto optimality was introduced to satisfy desires and it looks for a set of solutions that satisfy as many desires as possible. On the other hand, our preference criterion corresponds to a more strict reading of the preferences. In this respect, CR-Prolog is more *conservative* than LPOD, in the sense that it does not return an answer unless there is complete certainty that the answer is correct.

It is not difficult to see that the problem with returning all the conflicting solutions is that it does not allow the user to distinguish between conflicting solutions and solutions that satisfy all the preferences, thus forcing him to study carefully all the solutions (somewhat defeating the purpose of automatic computation) when the consequences of taking the wrong decision are extremely negative.

On the other hand, the fact that CR-Prolog disregards conflicting solutions, guarantees that the solutions returned, entirely comply with the preferences. If no solution is returned, that is likely to mean that the preferences should be revised.

CHAPTER X
CONCLUSIONS AND FUTURE WORK

10.1 Conclusions

In this dissertation, we have demonstrated the use of answer set programming and action languages in the design and implementation of intelligent agents.

The result is an agent architecture capable of sophisticated reasoning and interaction with the domain. In particular, our agent is able to:

- generate plans, expected to achieve the agent's goal under reasonable assumptions;
- monitor the execution of actions and detect unexpected observations;
- explain unexpected observations by:
  - hypothesizing that some event occurred, unobserved, in the past;
  - modifying the original description of the domain to match the observations.

We believe that the combination of all these forms of reasoning in a single agent is new in the literature. Also uncommon is the use of diagnosis in explaining unexpected observations. Although our notion of learning is similar to inductive learning and inductive logic programming, to the best of our knowledge this is the first introduction of *answer set based learning*, and the first time learning of the laws of an action language is included in an agent architecture.

We think that this dissertation shows the advantages of the use of these types of reasoning in an agent architecture. Examples have been given showing the importance of each reasoning component *per se*, as well as in combination with the others.

From the point of view of answer set programming, we believe we demonstrated once again the power and flexibility of the approach and of the associated languages by showing how an entire, sophisticated agent can be designed upon it. A-Prolog and

CR-Prolog were successfully employed in this dissertation for substantially different tasks, such as:

- Axiomatizing action descriptions and domain histories.
- Encoding reasoning modules.
- Formalizing control knowledge (using constraints) and preferences (with the new constructs of CR-Prolog).

Thanks to the declarative nature of the languages used, all the tasks resulted in programs that are easy to both understand, modify and reuse.

As we hope we demonstrated, a major difference between answer set based languages and other knowledge representation formalisms is that our languages are very close to the implementation level in spite of being entirely declarative. For this reason, the programs shown in this dissertation are almost directly executable in existing inference engines.[1]

The fact that the same domain model is shared by all the reasoning components is also uncommon in the literature and is another indication of the expressive power and flexibility of the languages used. Having a unique domain model simplifies both the design and maintenance of the model.

10.2 Future Work

Our agent architecture can be extended in several directions, and all of them are intriguing and deserve further investigation. The following is a list of the major steps that we see as natural continuation of this work.

First of all, we believe that the issue of goal selection needs to be studied carefully. In particular, it will be interesting to see how, and to what extent, preferences on goals can be formalized using CR-Prolog.

[1]Essentially, it is sufficient to introduce a few domain predicates to make them executable in SMODELS.

Another interesting topic is that of testing in diagnosis and learning. In this dissertation we have limited testing to abnormality fluents, and have assumed that all such fluents are observable. A natural extension of this work would consist in allowing the agent to deal with non-observable fluents, and to allow testing of other fluents besides abnormality fluents. The first issue has been in part investigated in this context in [35], but more work is needed in this direction. Testing besides abnormality fluents, on the other hand, has not been studied, to the best of our knowledge. It would require a substantially smarter selection of the fluents to be tested, possibly guided by the comparison of alternative candidate diagnoses.

In this version of the agent architecture, testing is carried out in a dedicated loop inside the procedural parts of the diagnostic and learning components. As the tests carried out by the agent get more complex, it will become more important to introduce testing as a sub-goal in the main observe-think-act loop, rather than having a separate testing loop in $Find\_Diag$ and $Find\_Correction$.

As the reader may have noticed, in this dissertation there was an asymmetry between the planning component, and the diagnostic and learning components. The asymmetry consists in the fact that the former is designed for complete information, while the latter can deal with incomplete information. It would be interesting to extend the planning component to deal with incomplete information as well. One way to accomplish this is by finding *conditional plans*, i.e. plans that achieve the goal independently from the missing information. A limitation of this approach is that in some interesting situations no conditional plan exists. Consider a situation in which the agent can use one of two doors to exit a room, knows that one of them is locked, but does not know which one. No conditional plan exists, but still it seems that a truly autonomous, rational agent should be able to solve the problem. An alternative approach to planning that would allow to solve this problem is the use of *possible plans*, i.e. sequences of actions that will achieve the goal under favorable circumstances, but may as well not succeed, depending on the missing information. Possible plans appear to work well when they are used inside a control loop, like ours, that monitors the execution of actions. In this context, they can be seen as a "dynamic variant" of conditional plans. Differently from conditional plans, however, possible plans seem to require fewer computational resources.

In the example above, a possible plan would be "go to door 1 and open it." If the plan succeeds, the agent has achieved its goal of exiting the room. Otherwise, by observing the failure of the plan, the agent will learn that door 1 is locked. Given this information, he will immediately determine that the goal can be achieved by using door 2. An interesting challenge involved in the generation of possible plans is that of guaranteeing the reachability of the goal when plans do not succeed. For example, if trying to open a locked door were known to result in a bomb going off, any rational agent would be expected to select a different course of action than the possible plan above. It will be also interesting to investigate the relationship between the generation of possible plans and the formalization of, and reasoning about, sensing actions.

BIBLIOGRAPHY

[1] K. Apt, A. Blair, and A. Walker. *Towards a theory of declarative knowledge*, pages 89–148. Foundations of deductive databases and logic programming. Morgan Kaufmann, 1988.

[2] Marcello Balduccini. USA-Smart: Improving the Quality of Plans in Answer Set Planning. In *PADL'04*, Lecture Notes in Artificial Intelligence (LNCS), Jun 2004.

[3] Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with A-Prolog. *Journal of Theory and Practice of Logic Programming (TPLP)*, 3(4–5):425–461, Jul 2003.

[4] Marcello Balduccini and Michael Gelfond. Logic Programs with Consistency-Restoring Rules. In Patrick Doherty, John McCarthy, and Mary-Anne Williams, editors, *International Symposium on Logical Formalization of Commonsense Reasoning*, AAAI 2003 Spring Symposium Series, pages 9–18, Mar 2003.

[5] Marcello Balduccini, Michael Gelfond, and Monica Nogueira. A-Prolog as a tool for declarative programming. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE'2000)*, pages 63–72, 2000.

[6] Marcello Balduccini and Veena S. Mellarkod. A-Prolog with CR-Rules and Ordered Disjunction. In *ICISIP'04*, pages 1–6, Jan 2004.

[7] Chitta Baral and Michael Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19(20):73–148, 1994.

[8] Chitta Baral and Michael Gelfond. Reasoning Agents In Dynamic Domains. In *Workshop on Logic-Based Artificial Intelligence*. Kluwer Academic Publishers, Jun 2000.

[9] Chitta Baral and Michael Gelfond. *Logic Programming and Reasoning about Actions*, pages 389–426. Handbook of Temporal Reasoning in Artificial Intelligence. Elsevier, 2005.

[10] Chitta Baral, Michael Gelfond, and Nelson Rushton. Probabilistic Reasoning with Answer Sets. In *Proceedings of LPNMR-7*, Jan 2004.

[11] Chitta Baral, Sheila A. McIlraith, and Tran Cao Son. Formulating diagnostic problem solving using an action language with narratives and sensing. In *Proceedings of the 2000 KR Conference*, pages 311–322, 2000.

[12] R. P. Bonasso, R. J. Firby, E. Gat, David Kortenkamp, D. Miller, and M. Slack. Experiences with an Architecture for Intelligent, Reactive Agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 1997.

[13] Craig Boutilier, Ronen I. Brafman, Carmel Domshlak, Holger H. Hoos, and David Poole. CP-nets: A Tool for Representing and Reasoning with Conditional Ceteris Paribus Preference Statements. *Journal of Artificial Intelligence Research*, 21:135–191, 2004.

[14] Gerhard Brewka. Logic programming with ordered disjunction. In *Proceedings of AAAI-02*, 2002.

[15] Gerhard Brewka, Ilkka Niemela, and Tommi Syrjanen. Implementing Ordered Disjunction Using Answer Set Solvers for Normal Programs. In Sergio Flesca and Giovanbattista Ianni, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002)*, Sep 2002.

[16] Gerhard Brewka, Ilkka Niemela, and Tommi Syrjanen. Logic Programs wirh Ordered Disjunction. 20(2):335–357, 2004.

[17] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Adding Weak Constraints to Disjunctive Datalog. In *Proceedings of the 1997 Joint Conference on Declarative Programming APPIA-GULP-PRODE'97*, 1997.

[18] Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Strong and Weak Constraints in Disjunctive Datalog. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR'97)*, volume 1265 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 2–17, 1997.

[19] Pedro Cabalar and Ramon Otero. Pertinence and Causality. In *Working Notes of 3rd Workshop on Nonmonotonic Reasoning, Action, and Change (NRAC)*, pages 111–119, 1999.

[20] Francesco Calimeri, Tina Dell'Armi, Thomas Eiter, Wolfgang Faber, Georg Gottlob, Giovanbattista Ianni, Giuseppe Ielpa, Christoph Koch, Nicola Leone, Simona Perri, Gerard Pfeifer, and Axel Polleres. The DLV System. In Sergio Flesca and Giovanbattista Ianni, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002)*, Sep 2002.

[21] Sandeep Chintabathina. Modeling Hybrid Domains Using Process Description Language. Master's thesis, Texas Tech University, Dec 2004.

[22] Sandeep Chintabathina, Michael Gelfond, and Richard Watson. Modeling Hybrid Domains Using Process Description Language. In *Proceedings of ASP '05 Answer Set Programming: Advances in Theory and Implementation*, pages 303–317, 2005.

[23] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerard Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 03)*. Morgan Kaufmann, Aug 2003.

[24] Grit Denker and Carolyn Talcott. Maude Specification of the MDS Architecture and Examples. Technical Report 03 2003, SRI International, 2003.

[25] Yannis Dimopoulos, J. Koehler, and B. Nebel. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the 4th European Conference on Planning*, volume 1348 of *Lecture Notes in Artificial Intelligence (LNCS)*, pages 169–181, 1997.

[26] Thomas Eiter, Esra Erdem, and Wolfgang Faber. Diagnosing Plan Execution Discrepancies in a Logic-Based Action Framework. Technical Report INFSYS RR-1843-04-03, Vienna University of Technology, Aug 2004.

[27] Thomas Eiter, Esra Erdem, and Wolfgang Faber. Plan Reversals for Recovery in Execution Monitoring. In *Proceedings 10th Internation Workshop on Nonmonotonic Reasoning (NMR 2004), Action and Causality Track*, pages 147–154, Jun 2004.

[28] Thomas Eiter, Michael Fink, and Jan Senko. KMonitor - A Tool for Monitoring Plan Execution in Action Theories. In *Proceedings of LPNMR-05*, pages 416–421, 2005.

[29] Selim Erdogan and Vladimir Lifschitz. Definitions in answer set programming. In *Proceedings of LPNMR-7*, Jan 2004.

[30] Michael Gelfond. Representing Knowledge in A-Prolog. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, volume 2408, pages 413–451. Springer Verlag, Berlin, 2002.

[31] Michael Gelfond and Nicola Leone. Knowledge Representation and Logic Programming. *Artificial Intelligence*, 138(1–2), 2002.

[32] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of ICLP-88*, pages 1070–1080, 1988.

[33] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, pages 365–385, 1991.

[34] Michael Gelfond and Vladimir Lifschitz. Action Languages. *Electronic Transactions on AI*, 3(16), 1998.

[35] Michael Gelfond and Richard Watson. Diagnostics with answer sets: Dealing with unobservable fluents. In *Proceedings of the 3rd International Workshop on Cognitive Robotics - CogRob'02*, pages 44–51, 2002.

[36] E. Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153:105–140, 2004.

[37] Patrick J. Hayes and John McCarthy. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[38] Katsumi Inoue. Hypothetical reasoning in logic programs. *Journal of Logic Programming*, 18(3):191–227, 1994.

[39] M. Kaminski. A note on the stable model semantics of logic programs. *Artificial Intelligence*, 96(2):467–479, 1997.

[40] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw–Hill CS Series. 1978.

[41] Loveleen Kolvekal. Developing an Inference Engine for CR-Prolog with Preferences. Master's thesis, Texas Tech University, Dec 2004.

[42] Vladimir Lifschitz. On the logic of causal explanation. *Artificial Intelligence*, 96:451–465, 1997.

[43] Vladimir Lifschitz. *Action Languages, Answer Sets, and Planning*, pages 357–373. The Logic Programming Paradigm: a 25-Year Perspective. Springer Verlag, Berlin, 1999.

[44] Vladimir Lifschitz. Answer Set Planning. In *Proceedings of IJCSLP 99*, 1999.

[45] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In *Proceedings of the 11th International Conference on Logic Programming (ICLP94)*, pages 23–38, 1994.

[46] Vladimir Lifschitz and Hudson Turner. Representing transition systems by logic programs. In *Proceedings of the 5th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR-99)*, number 1730 in Lecture Notes in Artificial Intelligence (LNCS), pages 92–106. Springer Verlag, Berlin, 1999.

[47] David Lorenzo. *Learning non-monotonic logic programs to reason about actions and change.* PhD thesis, Corunna University, Nov 2001.

[48] David Lorenzo. Learning non-monotonic causal theories from narratives of actions. In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR'2002)*, Apr 2002.

[49] Norman McCain. *Causality in commonsense reasoning about actions.* PhD thesis, University of Texas, 1997.

[50] Norman McCain and Hudson Turner. A causal theory of ramifications and qualifications. *Artificial Intelligence*, 32:57–95, 1995.

[51] Sheila A. McIlraith. Representing actions and state constraints in model-based diagnosis. In *Proceedings of AAAI-97*, pages 43–49, 1997.

[52] Sheila A. McIlraith. Explanatory diagnosis conjecturing actions to explain observations. In *Proceedings of the 1998 KR Conference*, pages 167–177, 1998.

[53] Veena S. Mellarkod. Optimizing the Computation of Stable Models using Merged Rules. Master's thesis, Texas Tech University, May 2002.

[54] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw–Hill Series in Electrical and Computer Engineering. 1994.

[55] Robin Murphy. *Introduction to AI Robotics.* MIT Press, 2000.

[56] Ilkka Niemela and Patrik Simons. *Extending the Smodels System with Cardinality and Weight Constraints*, pages 491–521. Logic-Based Artificial Intelligence. Kluwer Academic Publishers, 2000.

[57] Monica Nogueira. *Building Knowledge Systems in A-Prolog.* PhD thesis, University of Texas at El Paso, May 2003.

[58] M. Otero and Ramon Otero. Using causality for diagnosis. In *Proceedings of the 11th International Workshop on Principles of Diagnosis*, pages 171–176, 2000.

[59] Ramon Otero. Induction of Stable Models. In *Proceedings of 11th Int. Conference on Inductive Logic Programming, ILP-01*, number 2157 in Lecture Notes in Artificial Intelligence (LNCS), pages 193–205, 2001.

[60] Enrico Pontelli and Tran Cao Son. Reasoning about Actions and Planning with Preferences Using Prioritized Default Theory. *Computational Intelligence*, 20(2):358–404, 2004.

[61] Raymond Reiter. *On Closed World Data Bases*, pages 119–140. Logic and Data Bases. Plenum Press, 1978.

[62] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[63] Raymond Reiter. Natural actions, concurrency and continuous time in the situation calculus. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'96)*, pages 2–13, Nov 1996.

[64] Raymond Reiter. *Knowledge in Action – Logical Foundations for Specifying and Implementing Dynamical Systems.* MIT Press, Sep 2001.

[65] Chiaki Sakama. Inverse Entailment in Nonmonotonic Logic Programs. In *Proceedings of the 10th International Conference on Inductive Logic Programming,*

ILP 00, number 1866 in Lecture Notes in Artificial Intelligence (LNCS), pages 209–224, 2000.

[66] Chiaki Sakama. Induction from answer sets in nonmonotonic logic programs. *ACM Transactions on Computational Logic*, 6(2):203–231, Apr 2005.

[67] Chiaki Sakama and Katsumi Inoue. Prioritized Logic Programming and its Application to Commonsense Reasoning. *Artificial Intelligence*, 123:185–222, 2000.

[68] M. Shanahan. *Solving the frame problem: A mathematical investigation of the commonsense law of inertia*. MIT Press, 1997.

[69] Patrik Simons. Extending the Stable Model Semantics with More Expressive Rules. In *Proceedings of the 5th International Conference on Logic Programming and Non-monotonic Reasoning (LPNMR-99)*, number 1730 in Lecture Notes in Artificial Intelligence (LNCS). Springer Verlag, Berlin, 1999.

[70] Mikhail Soutchanski. High-level Robot Programming and Program Execution. In *Proceedings of the ICAPS-03 Workshop on Plan Execution*, Jun 2003.

[71] Mikhail Soutchanski. *High-Level Robot Programming in Dynamic and Incompletely Known Environments*. PhD thesis, University of Toronto, 2003.

[72] Luis Ng Tari. Learning AnsProlog Rules. Master's thesis, Arizona State University, Jun 2004.

[73] Michael Thielscher. A theory of dynamic diagnosis. *Linkoping Electronic Articles in Computer and Information Science*, 2(11), 1997.

[74] Michael Thielscher. Ramification and causality. *Artificial Intelligence*, 89:317–364, 1997.

[75] Michael Thielscher. Introduction to Fluent Calculus. *Linkoping Electronic Articles in Computer and Information Science*, 3(14), 1998.

[76] Hudson Turner. Splitting a Default Theory. In *Proceedings of AAAI-96*, pages 645–651, 1996.

[77] Hudson Turner. Reprenting Actions in Logic Programs and Default Theories: A Situation Calculus Approach. *Journal of Logic Programming*, 31(1-3):245–298, Jun 1997.

[78] Hudson Turner. Polynomial-Length Planning Spans the Polynomial Hierarchy. In Sergio Flesca and Giovanbattista Ianni, editors, *Proceedings of the 8th European Conference on Artificial Intelligence (JELIA 2002)*, pages 111–124, Sep 2002.