BUILDING KNOWLEDGE SYSTEMS IN A-PROLOG

MONICA DE LIMA NOGUEIRA
Computer Science Department

APPROVED:

Michael Gelfond, Ph.D., Chair

Chitta Baral, Ph.D.

Vladik Kreinovich, Ph.D.

Luc Longpré, Ph.D.

Enrico Pontelli, Ph.D.

Charles H. Ambler, Ph.D.
Dean of the Graduate School

BUILDING KNOWLEDGE SYSTEMS IN A-PROLOG

by

MONICA DE LIMA NOGUEIRA

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

Computer Science Department

THE UNIVERSITY OF TEXAS AT EL PASO

MAY 2003

THE UNIVERSITY OF TEXAS AT EL PASO

Date: **May 2003**

Author: **MONICA DE LIMA NOGUEIRA**

Title: **Building Knowledge Systems in A-Prolog**

Department: **Computer Science**

Degree: **Ph.D.** Convocation: **May** Year: **2003**

Signature of Author

To my children,

Heloisa, Marcus Vinicius, and Daniela,

with love.

Acknowledgements

"As we express our gratitude, we must never forget that the highest appreciation is not to utter words, but to live by them."

John F. Kennedy (1917-1963)

I am blessed to have always had the opportunity to meet extraordinary people. People who have taught me a great deal. I am also blessed to have the support and love from family and friends no matter where in the world I happened to be. Therefore, I am lucky to have a long list of people to thank for the help they gave me to complete my studies.

I am especially indebted to my advisor Michael Gelfond for sharing his knowledge and his vision of science, philosophy, mathematics, politics, religion, literature, beauty, history, and innumerous other subjects. From him I learned about logic and the validity of arguments, both of which have changed the way I think and interact with other people. Without his help and encouragement, this work would not have been successful.

I am grateful for what I learned in the many classes I took with Chitta Baral, Vladik Kreinovich, and Luc Longpré, and for their support and attention to detail when reviewing this dissertation as my committee members. I want to thank my external committee member Enrico Pontelli for his helpful comments and the many suggestions he made to improve this text. I also had the help and support of Son Cao Tran who reviewed an earlier version of the mathematical proofs presented here and pointed out many of the mistakes I needed to correct. I especially need to thank Michael, Chitta, Enrico, and Son for making the effort of coming to El Paso on such short notice at the end of the semester to attend my dissertation defense; and Luc and Vladik for making special arrangements regarding their final exams so they could be present. Luc was also thoughtful enough as to write down his observations about how I can improve my presentation of this work, for which I thank him.

As a demanding client who is very particular about details, I need to thank Marcello Balduccini twice for the many hours he worked in the implementation of the graphical interfaces for the A-Circuit and the USA-Advisor systems that I designed. It was a pleasure working with him on these projects, and I greatly benefited from our discussions.

I am also thankful to other graduate students (past and present) from the Knowledge Representation Laboratory (KRLab), previously at The University of Texas at El Paso (UTEP) and currently at Texas Tech University (TTU). I particularly wish to thank Veena Mellarkod, Joel Galloway, and Mary Heidt who allowed me to participate in their work and to learn from it.

For the last three years I have been a constant visitor at the KRLab at TTU. I wish to thank Daniel Cooke and his staff for making the Computer Science Department at TTU an extension of the one at UTEP; I always felt welcome there and found it a friendly place where I could continue my work. During this time, I was also constantly a guest at the Gelfonds' home and wish to express my gratitude to Lara Gelfond for her gracious hospitality and friendship. I especially wish to thank Lara, Gregory, and Michael for those extra working hours stolen from their family time together.

Since I started working with the SMODELS and the DVL inference engines, I have had the help and support of their development teams, which I would like to thank: Illka Niemelä, Patrik Simons, and Tommi Syrjänen for SMODELS, and Nicola Leone, Wolfgang Faber, and Gerald Pfeifer for DLV.

I am also very fortunate because I participated and learned from many technical discussions and exchanges of ideas in the Texas Action Group (TAG) organized by Vladimir Lifschitz and Michael Gelfond. I wish to thank them both for creating this forum and encouraging researchers and students alike to be part of it. I particularly want to thank Vladimir, Michael, and the TAG members for the ideas they have shared with me. This group includes Yuliya Babovich, Marcello Balduccini, Chitta Baral, Pedro Cabalar, Jonathan Campbell, Esra Erdem, Wolfgang Faber, Paolo Ferraris, Alfredo Gabaldon, Joel Galloway, Mary Heidt, Vladik Kreinovich, Jooyhung Lee, Sheila McIlraith, Veena Mellarkod, Ramon Otero, Gerald Pfeifer, Enrico Pontelli, Tommi Syrjänen, Son Cao Tran, Le-Chi Tuan, Hudson Turner, and many others.

I would like to thank the United Space Alliance (USA) company for providing partial financial support for this research through several research grants and contracts. In addition, I would like to thank Matthew Barry from USA and Richard Watson from TTU for answering questions about the Reaction Control System of the space shuttle which helped the development of this work. My research was also partially supported by a scholarship from the National Science Foundation for which I am very grateful.

There are too many friends who helped me through my studies and for whom I am thankful. Of all these friends, I wish to single out Nelly Delgado and Frank Fernandez for their constant support and unconditional dependability. During the busy preparations for my dissertation, they always made sure the little things got done. They will always have my gratitude and my friendship.

Finally I wish to express my utmost gratitude to my parents for their love and willingness to be my support team at home, supervising my children and my household for weeks and months at a time while my husband and I were many miles away. I would never be able to complete my studies without their help. I want to thank my husband for his support and for believing I could finish this work even when I did not. Lastly, I want to thank my children for understanding and valuing my desire to become a better person through my studies, even if it meant I would be away for long periods of time. I am sure this hardship brought us closer together and was a worthwhile lesson that will always stay with us.

Monica de Lima Nogueira

The University of Texas at El Paso

May 2003

Abstract

This work is written in the context of the logic-based approach to Artificial Intelligence (AI) proposed by John McCarthy in 1959 [134]. According to this approach an agent should have knowledge of its world and its goals, and the ability to use this knowledge to infer its course of action. This logic-based method suggests that a mathematical model of an agent should contain: a formal language capable of expressing commonsense knowledge about the world, a precise characterization of valid conclusions which can be derived from theories stated in this language, and a means which will allow the agent to arrive at these conclusions.

The purpose of this dissertation is to investigate the applicability of one such language, A-Prolog [71, 73], for the development of medium-size knowledge-intensive systems. A-Prolog is a declarative logic programming language based on stable models/answer sets semantics of logic programs [74, 75]. It allows the representation of defaults and several interesting aspects of reasoning about actions and their effects. There is a recently developed methodology of representing knowledge in A-Prolog, and there are also rather efficient inference engines associated with the language. Our goal

was to test this methodology and these inference engines on sizeable engineering applications.

In this dissertation, we developed two such applications. The first is a small system, designed as a classroom tool for teaching digital circuits, which allows the functional and behavioral representation of these circuits at the gate-level of abstraction. The second is a substantially larger application - the implementation of a decision support system for the space shuttle's flight controllers. This work involved the representation of a substantial amount of knowledge about the shuttle as well as the execution of complex planning (and other reasoning) tasks. The project was successful, and the system is now in the hands of United Space Alliance (USA), the company responsible for overseeing the operation of the space shuttle.

This dissertation describes the design and implementation of these systems and discusses some lessons derived from this experience. We believe that the lessons can be of interest to AI researchers working in the areas of knowledge representation, nonmonotonic reasoning, and planning, as well as to software engineers involved in the construction of knowledge-intensive systems.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

"If I hear, I forget.

If I see, I remember.

If I do, I understand."

Proverb

In this chapter we present the background information necessary to understand the subject of this dissertation. The chapter is organized as follows. First, we discuss the avenues of research which originated from the idea of using logic to represent Artificial Intelligence (AI) related knowledge. Secondly, we cover the first difficulties encountered by the logic programming approach and the first nonmonotonic formalisms which sought to solve these problems. Next, we illustrate our programming methology through an example, using the famous blocks world domain. We present the goals and contributions of this work, and finally, we give the organization of the dissertation.

1.1 Logic Approach to AI

In 1959 [134], John McCarthy proposed the use of *logical formulas* to represent AI-related knowledge. He expressed the advantages of his idea as follows:

> *"Expressing information in declarative sentences is far more modular than expressing it in segments of computer programs or in tables. Sentences can be true in a much wider context than specific programs can be used. The supplier of a fact does not have to understand much about how the receiver functions, or how or whether the receiver will use it. The same fact can be used for many purposes because the logical consequences of collections of facts can be available."*

The original approach was to use classical logic to represent and reason about various types of knowledge. Unfortunately, it was soon discovered that this approach may not be adequate for representing commonsense knowledge. In [148], Minsky discussed one such domain which involved describing birds' flying abilities.

This classical problem consisted of representing the statement

$$\text{“Normally, birds fly.”} \tag{1.1}$$

This is a typical example of a *default* - a statement "normally, typically, as a rule, elements of class C have property P." Reasoning with defaults and their exceptions seems to be essential in our everyday life. However, it does not occur in mathematics.

Minsky argued that defaults cannot, in principle, be represented by the means of logic. Indeed, suppose that the common knowledge about flying abilities of birds, including the following four statements:

1. Birds fly.

2. Penguins are birds.

3. Penguins do not fly.

4. Tweety is a bird.

is encoded by a theory $\mathcal{T}$ of classical logic.

Clearly, $\mathcal{T}$ should entail that "Tweety flies" since Tweety is a bird and birds fly. Now, if we happen to know that "Tweety is a penguin," adding this new knowledge to $\mathcal{T}$ will make $\mathcal{T}$ inconsistent. Theory $\mathcal{T}$ will now entail that: (a) "Tweety flies," as before, and (b) "Tweety does not fly" because Tweety is a penguin and penguins do not fly.

Intuitively, we should be able to withdraw the previous conclusion, (a), and conclude (b) in the presence of the new knowledge. This example shows that classical logic is not suitable for formalizing commonsense knowledge.

Reasoning which permits retraction of previous conclusions when confronted with contradicting knowledge is called *nonmonotonic.* In more precise terms an entailment relation $\models$ (over language $\mathcal{L}$) is called *nonmonotonic* if there are formulas A and B and a set of formulas T such that $T \models B$ and $T, A \not\models B$. Otherwise, the entailment is said to be *monotonic.* Classical logic is monotonic. For formalization of commonsense we seem to need nonmonotonicity.

This realization led to the introduction of several formalisms which extend classical logic to allow for nonmonotonic reasoning. Some of the most important ones are Default Logic [169], Autoepistemic Logic [149, 150], and Circumscription [135, 136].

Meanwhile, another line of research led by Kowalski [107, 108] and Colmerauer [44], which originated with the introduction of the *resolution principle* by Robinson [173], and was influenced by Hayes' [94] idea that "computation is controlled deduction," concentrated on developing efficient algorithms that would allow for "programming in logic." This work gave birth to the first interpreter for the PROLOG Programming Language [45].

The language is based on definite Horn clauses. A definite clause has the form:

$$h \leftarrow a_1 \wedge \ldots \wedge a_n$$

where head h is either an atom, and body $a_1 \wedge \ldots \wedge a_n$ is a conjunction of atoms. If the body is empty, the rule is called a *fact.* The symbol "$\leftarrow$" is read as *if.* A program in PROLOG consists of a set of definite clauses, which can be interpreted both declaratively and procedurally. The semantics of a definite clause

$$p \leftarrow q \wedge r$$

can be perceived in one of the two following ways:

(a) p is true if q and r is true;

(b) to prove p, prove q and prove r.

The procedural interpretation is the basis for the implementation of the PROLOG language which answers queries about an input program P. If a query q has no variables, the interpreter returns "yes" if it finds a proof of q from P. Otherwise, it returns "no." If there are variables in the query, e.g. $q(X)$, the system answers "no" if no terms satisfying the query are found, or returns the first "substitution" $X = t$ that is found. The inference is based on the adaptation of Robinson's resolution [173] by [90, 91, 108].

A PROLOG program is often understood as a collection of clauses together with an interpreter. Even though programs

$$\Pi_0 \begin{cases} p \leftarrow p \\ p \end{cases} \qquad \Pi_1 \left\{ \; p \right.$$

are equivalent declaratively, when viewed procedurally they exhibit different behaviors. Program Π_1 stops and returns "yes" to query p, but program Π_0 goes into an infinite loop when trying to find a proof for p.

Later, "Pure Prolog" of definite clauses was expanded by a new logical connective, *not* , called "negation as failure." The first interpretation of this connective was purely procedural and given in terms of the Prolog interpreter. A rule

$$p \;\text{:-}\; q, not\ r$$

reads as "if q is proven and no proof for r is found, then p is proven." The symbol ":-" is read as *if.* A program consisting of this rule and the atom q answers "yes" to p. Addition of r forces the program to withdraw its answer. The Prolog inference process becomes nonmonotonic. The problem of finding a declarative semantics for *not* proved difficult. The first pioneering work to give a semantics for *not* was done by: Clark [42] who introduced the *negation as finite failure* rule and the notion of *completion* of a logic program; Reiter [168] through the encoding of the *Closed World Assumption*; Apt, Blair, and Walker [5] who formalized the notion of *stratification* of logic programs; van Gelder, Ross, and Schlipf's [197] introduction of the *well-founded semantics*; and Gelfond and Lifschitz's [74] *stable models/answer sets semantics*. There are many other approaches. A survey of the use of negation in logic programming is presented in [6].

These two lines of research converged to develop the field of Logic Programming and Nonmonotonic Reasoning, and the A-Prolog language which extends "classical" Prolog by classical negation and disjunction. This language was shown to be closely connected with Default [78] and Autoepistemic Logic [129].

A-Prolog is a declarative logic programming language based on stable models/answer sets semantics [74, 75] of logic programs. It allows the encoding of defaults and various other types of knowledge contained in dynamic domains, e.g. the representation of actions and their effects. In recent years, the development of several different reasoning systems for A-Prolog led to the emergence of answer set programming [131, 152], a new programming paradigm. Currently, the most efficient inference engines for A-Prolog are SMODELS[1] [153, 154, 155, 156], and DLV[2] [41, 55, 53, 54].

Another difficulty in the realization of McCarthy's program was discovered when researchers attempted to represent information about effects of actions.

To illustrate the issues involved in this type of reasoning we will use a classic AI example: the blocks world domain. It consists of a number of blocks which can sit directly on a table or be stacked up by action *move* block X on top of block Y. A similar action can be used to unstack a block and move it to the table or on top of another block.

To model this domain we need to represent blocks, which can easily be done with a collection of facts, e.g.

```
block(a).
block(b).
block(c).
block(d).
```

and the table denoted by t.

There are two types of locations where a block can sit: the top of another block or the table. The following two rules express that a location is either a block or the table.

```
location(X) :- block(X).
```

[1]The SMODELS homepage is located at http://www.tcs.hut.fi/Software/smodels/

[2]The DLV homepage is located at http://www.dbai.tuwien.ac.at/proj/dlv/

```
location(t).
```

A state of the domain is defined by the relation *on(B,L)*, which says that block B is at location L. The truth value of this relation changes with time through the execution of action *move(B,L)*, defined by rule

```
action(move(B,L)) :-
                    block(B),
                    location(L).
```

For instance, execution of action *move(b,a)* changes the state in Figure 1.1(a) into the state shown in Figure 1.1(b).

Figure 1.1: Blocks World Domain.

We would like to specify a transition diagram describing all the possible trajectories of the domain. To do that we need to define the state of the domain and the transition relation $\langle \sigma_0, a, \sigma_1 \rangle$, where σ_1 is the state of the domain after action a is performed in previous state σ_0.

We start with introducing a relation $holds(F,T)$ which defines that fluent F holds, i.e. is true, at time T. By T we mean a discrete time point. (For simplicity, we assume that the execution of each action takes one time unit.)

A state σ_0 will be given by a collection of atoms $holds(on(B, L), 0)$; state σ_1 - by a collection of atoms $holds(on(B, L), 1)$; a state k in the path $\langle \sigma_0, a_0, \ldots, \sigma_k, a_k, \sigma_{k+1} \rangle$ will be given by a collection of atoms $holds(on(B, L), k)$.

Explicit negative information is expressed in A-Prolog through the "classical negation" connective, denoted by $\neg$. Rule

```
¬holds(on(B,L2),T) :-
                    block(B),
                    time(T),
                    location(L1),
                    location(L2),
                    L1 ≠ L2,
                    holds(on(B,L1),T).
```
(1.2)

provides negative information about positions of blocks. It says that a block B is not at a location L_2 at time T if it is at a different location L_1 at this time.

This rule is a logic programming variant of the so called "state constraint" (or static causal law) [132]

$$g \textbf{ if } f \tag{1.3}$$

which says that property g must be true in any state where property f is true. In this case, $\neg on(B, L_2)$ **if** $on(B, L_1), L_1 \neq L_2$.

We also need to define the direct effects of actions. For that we write a "dynamic causal law" of the form

$$a \textbf{ causes } f \textbf{ if } p \tag{1.4}$$

which says that performing action a causes property f to become true if preconditions p are true in this state.

In the blocks world domain this corresponds to rule

```
holds(on(B,L),T+1) :-
                        block(B),
                        location(L),
                        time(T),
                        occurs(move(B,L),T).            (1.5)
```

Relation *occurs(move(B,L),T)* defines an "observation" of the occurrence of action *move(B,L)* in the state T of the domain.

It is not always possible to execute such an action, e.g. if there exists a block B_1 on top of block B then B cannot be moved. Knowledge of this type is referred to as an "impossibility condition," and is represented by rules with empty heads. The empty head of such a rule means that the body must be false in all models of the program. If the head is empty the rule is often called a "constraint."

The impossibility condition identified above is described by rule

```
:- block(B),
   block(B1),
   B ≠ B1,
```

```
    location(L),
    time(T),
    occurs(move(B,L),T),
    holds(on(B1,B),T).
```
(1.6)

It is also not possible to move a block B to a location L if there exists another block B_1 at L, i.e.

```
:- block(B),
    block(B1),
    B ≠ B1,
    location(L),
    time(T),
    occurs(move(B,L),T),
    holds(on(B1,L),T).
```
(1.7)

The following constraint states that a block B cannot be moved on top of itself.

```
:- block(B),
    time(T),
    occurs(move(B,B),T).
```
(1.8)

Rules (1.2), (1.5), (1.6), (1.7), and (1.8) above describe the changes caused in the state of the domain by execution of action *move(B,L)*. It is also necessary to describe what has not changed in the state of the domain after executing action a, i.e. which fluents values have not been altered by a. In logic programming this can be done by the following default

```
holds(on(B,L),T+1) :-
                  block(B),
                  location(L),
                  time(T),
                  holds(on(B,L),T),
                  not ¬holds(on(B,L),T+1).        (1.9)
```

which says that if a block B is at location L at time T, and there is no reason to believe it is not at L in the next moment of time $T+1$, then it is still there at $T+1$.

The above default encodes the *commonsense law of inertia* - "normally, things tend to stay as they are." Formalization of this default was proposed by McCarthy [137] as a possible solution for the *frame problem*. This famous problem, first pointed out in [138], consisted of describing concisely what should not change in the current state of the domain after an action is executed.

Negation as failure, which permits an elegant representation of defaults in A-Prolog, allows for a simple solution of this problem given by rule (1.9). The rule also helps solving the ramification and qualification problems. The *ramification problem* [66] consists of representing the indirect effects of actions. In A-Prolog it is solved by combining the inertia axiom with dynamic causal laws and state constraints. In the case of the blocks world - by rules (1.2), (1.5), and (1.9). If a block B is moved from the table on top of another block B_1, then an indirect effect of the execution of this action will be that B is no longer on the table. The *qualification problem* [135] consists of describing in a concise way the (impossibility) conditions that would prevent the execution of an action. In A-Prolog this is done by writing constraints like (1.6), (1.7), and (1.8).

The resulting state σ_1, after the execution of an action a in a state σ_0, is often called a *successor* state [171]. It was difficult to define the values of fluents for successor states. The solution to the frame, ramification, and qualification problems made this possible. In A-Prolog these solutions are based on the concept of a transition diagram of the domain. There are various definitions of this diagram and its transition relation, [133, 194]. These definitions are independent of the notion of answer sets and based on various theories of causalities. A different definition of transition relation will be given below. (For details see [26]).

We first need to introduce the following notation.

Let Π_0 be a program consisting of dynamic and static causal laws, the inertia axiom, and impossibility constraints, where time $T \in \{0, 1\}$. Let

$$holds(\sigma, k) = \{holds(f, k) : f \in \sigma\} \cup \{\neg holds(f, k) : \neg f \in \sigma\},$$

and let $occurs(a, 0)$ be an observation of the occurrence of action a at $T = 0$.

Definition 1.1. *A* transition $\langle \sigma_0, a, \sigma_1 \rangle$ *belongs to the transition diagram of the domain described by* Π_0 *if there exists an answer set* S *of program*

$$\Pi_0 \;\cup\; holds(\sigma_0, 0) \;\cup\; \{occurs(a, 0)\}$$

such that

(a) $f \in \sigma_1$ *iff* $holds(f, 1) \in S$;

(b) $\neg f \in \sigma_1$ *iff* $\neg holds(f, 1) \in S$.

There is a remarkable relationship between the logic programming based definition of the transition diagram given above and the causality based definitions from [133, 194]. This relationship not only establishes the close connection between causality and beliefs but also allows us to reduce various reasoning tasks of a dynamic agent to computing answer sets of various programs. For instance, program Π_0 can be used to solve classical AI tasks like planning and diagnosis. Let us illustrate the basic idea of this reduction by an example:

Consider the initial situation σ_0 for the blocks world domain, shown in Figure 1.2(a), and goal situation σ_n, shown in Figure 1.2(b).

The initial situation $holds(\sigma_0, 0)$ is described by facts

```
holds(on(a,d),0).
holds(on(b,a),0).
holds(on(c,t),0).
```

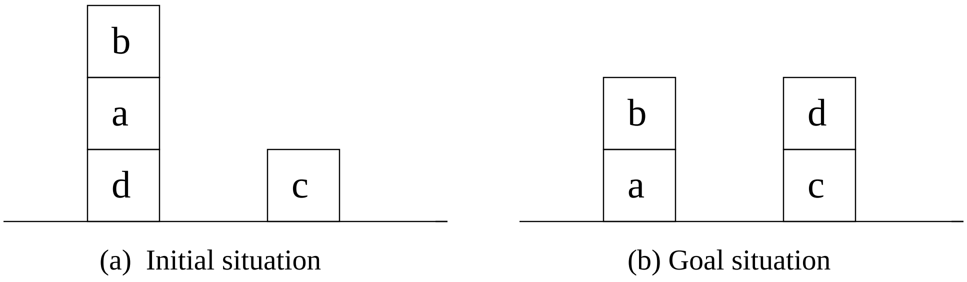


Figure 1.2: Blocks World Domain.

```
holds(on(d,t),0).
```

The goal $\mathcal{G}(\sigma_n, T)$ is represented by rules

```
goal(T) :-
          time(T),
          holds(on(a,t),T),
          holds(on(b,a),T),
          holds(on(c,t),T),
          holds(on(d,c),T).

goal :-
          time(T),
          goal(T).
```

which describe what must be true in the goal situation, and the constraint below that eliminates all models not satisfying the goal.

```
:- not goal.
```

To find a plan of length not exceding n, i.e. $T < n$, let us take program Π_n, which is program Π_0 with time now ranging from 0 to n, and expand this program in the following way. The generation phase of planning will be implemented using a *choice* rule $\mathcal{C}R$, which has the form

```
1{occurs(A,T):action(A)}1 :-
                          time(T),
                          T < n,
                          not goal(T).
```

This rule states that, for each time point $T < n$, if the goal has not been reached, then an action must occur at that time.

Choice rules are part of the language of SMODELS [155]. The head of the choice rule has the form

$$L\{p(\bar{X}) : q(\bar{X})\}U.$$

It defines a subset $p \subseteq q$ of terms such that $L \leq |p| \leq U$. Normally, there are many possible sets satisfying these conditions. Hence, a program containing this type of rules might have multiple answer sets, corresponding to possible choices of p. Choice rules do not extend the expressive power of the logic programming language and can be viewed as a shorthand for a set of standard rules of the language. These rules, however, proved to be very convenient. They substantially shorten the program, and more importantly, they allow for an efficient implementation.

The problem of finding a plan to move from σ_0 to σ_n, of length not exceding n, can be reduced to finding an answer set of program

$$\Pi_n \;\cup\; holds(\sigma_0, 0) \;\cup\; \mathcal{G}(\sigma_n, T) \;\cup\; \mathcal{C}R.$$

It is easy to check, using SMODELS, that we can find an answer set of this program corresponding to a plan which achieves this goal. One such plan is

$$\begin{array}{l}\langle\ occurs(move(b,t),0),\;\; occurs(move(a,t),1),\\ \;\; occurs(move(b,a),2),\;\; occurs(move(d,c),3)\ \rangle\end{array}$$

There are other plans and other answer sets. The complete program given as input to SMODELS is shown in Figure 1.3. Notice that the rules in the program are slightly different from the ones we presented here, in order to accomodate SMODELS syntax and type requirements. We also use SMODELS' display formatting capabilities, e.g. *hide* $p(X)$, in order to display just the atoms that constitute a plan. All plans, of length not exceding 4 steps, which achieve the goal described in this example are given in Figure 1.4, and were computed in 0.06 seconds. There exists no answer set corresponding to a plan of length smaller than 4.

The answer set programming paradigm was shown to be adequate for comparatively small problems/domains. Although most of the attention was given to answer set planning [48, 120], diverse interesting problems have been so far solved using answer

```
%%%%%%%%%%%%%%%%%%%%%     Blocks World Program     %%%%%%%%%%%%%%%%%%%%%

% Objects of the domain
   block(a).
   block(b).
   block(c).
   block(d).

   location(X) :- block(X).
   location(t).

   action(move(B,L)).


% State constraint (Static Causal Law)
  -holds(on(B,L1),T) :- holds(on(B,L),T), neq(L,L1).

% Dynamic Causal Law
  holds(on(B,L),T+1) :- occurs(move(B,L),T).

% Inertia Law
  holds(on(B,L),T+1) :- holds(on(B,L),T), not -holds(on(B,L),T+1).


% Impossibility conditions

% Constraint 1:
% A block topped by another block cannot be moved.
   :- occurs(move(B,L),T), holds(on(B1,B),T), neq(B,B1).

% Constraint 2:
% A block cannot be moved to a location occupied by another block.
   :- occurs(move(B,L),T), holds(on(B1,L),T), neq(B,B1), neq(L,t).

% Constraint 3:
% A block cannot be moved on the top of itself.
   :- occurs(move(B,B),T).


% Plan generation rule
   1{occurs(A,T):action(A)}1 :- T < lasttime, not goal(T).
```

Figure 1.3: Program describing the blocks world domain given as input to SMODELS.

```
%%%%%%%%%%%%%%%%     Blocks World Program (cont.)    %%%%%%%%%%%%%%%%

% Initial situation
   holds(on(a,d),0).
   holds(on(b,a),0).
   holds(on(c,t),0).
   holds(on(d,t),0).

% Goal situation
   goal(T) :- holds(on(a,t),T),
              holds(on(b,a),T),
              holds(on(c,t),T),
              holds(on(d,c),T).

   goal :- goal(T).

   :- not goal.


% Time definition
% Maximum plan length is determined by constant lasttime, provided by
% user at run time.

   time(0..lasttime).


% Types definition
   #domain time(T).
   #domain block(B;B1).
   #domain location(L;L1).


% Display formatting commands
   hide block(X).
   hide location(X).
   hide action(X).
   hide holds(X,Y).
   hide time(X).
   hide goal.
```

Figure 1.3: Program describing the blocks world domain given as input to SMODELS.

```
%%%%%%%%%%%%%%    Results for Blocks World Program    %%%%%%%%%%%%%%

% Command line for SMODELS to compute all answer sets of blocks_world
% input program

lparse --true-negation -c lasttime=4 blocks_world | smodels 0


% Result of computation

smodels version 2.26.  Reading...done

Answer:  1
Stable Model:  occurs(move(b,t),0) occurs(move(a,t),1)
               occurs(move(d,c),2) occurs(move(b,a),3)

Answer:  2
Stable Model:  occurs(move(b,t),0) occurs(move(a,t),1)
               occurs(move(b,a),2) occurs(move(d,c),3)

Answer:  3
Stable Model:  occurs(move(b,c),0) occurs(move(a,t),1)
               occurs(move(b,a),2) occurs(move(d,c),3)


False
Duration:  0.060
Number of choice points:  2
Number of wrong choices:  2
Number of atoms:  384
Number of rules:  1344
Number of picked atoms:  82
Number of forced atoms:  45
Number of truth assignments:  2320
Size of searchspace (removed):  8 (6)
```

Figure 1.4: Results for blocks world program of Figure 1.3.

set programming, e.g. product configuration [184], wire routing [61], etc. In this dissertation, we developed a substantially larger application.

1.2 Goals and Contributions of this work

The goals of this work are to answer the following two questions:

1. Is it possible to represent a real world problem of reasonable size involving complex effects of actions with the A-Prolog language?

2. Are the available inference engines for A-Prolog able to compute the solutions to such a domain in a reasonably efficient manner?

We address these questions in two steps.

The first step was inspired by my work aimed at representing knowledge about digital circuits for the Digital Design II graduate class at The University of Texas at El Paso. In this class, students were required to learn a *Hardware Description Language* (HDL) [30, 84], *VHDL* [101, 176] or *Verilog HDL* [159, 160, 186], in order to complete a class project. This project consisted of representing and simulating a digital circuit using the language and its simulator. The size and complexity of these languages soon led me to wonder if it would not be possible, and simpler, to complete these tasks using a declarative language, more specifically, the A-Prolog language. Because it provides an extensive range of capabilities, the *VHDL* language is considered complex and difficult to understand, even by experienced digital designers [84]. *Verilog*'s syntax is similar to the C programming language, and is regarded by designers as an easy to learn and teach language because of its compact size [160]. This is an understimation caused by the ever present comparison between *Verilog* and *VHDL*, the two most popular Hardware Description Languages today. My limited experience with Hardware Description Languages increased my difficulties and motivated me to program the assignment in A-Prolog. At that point, the original question now focused on whether A-Prolog would allow the representation of digital circuits in a simpler way and if the language would be powerful enough to also permit the simulation of such circuits.

Results were positive and satisfactory in both accounts. We designed a simple tool - the A-Circuit[3] system [14], that can be used by students in Digital Design or other related classes, to represent and simulate (simple) digital circuits. One main advantage of our approach is the use of a single language to describe both structure and behavior of gates, and as a simulation environment, which results in a uniform approach to the simulation of digital circuits. The A-Circuit system also incorporates some other more sophisticated tasks, which cannot currently be achieved by using traditional HDL languages. In some cases, several HDL related tools must be used in order to achieve a task; in other cases, e.g. diagnosis, HDL languages still cannot support such features. We discuss the design of the A-Circuit system in Chapter 3. The correctness of the system is proved by various propositions. Chapter 4 presents these proofs.

[3]The A-Circuit system is available for download from:
http://www.krlab.cs.ttu.edu/Download/A-Circuit/

The second step was more ambitious. We got involved in a real world application, a project supported by NASA's major contractor, the United Space Alliance (USA) company. The objectives of the project were:

1. to represent information about some subsystems of the space shuttle; and

2. to design a decision support system for flight controllers of the shuttle.

The high expressive power and simplicity of the A-Prolog language were fundamental to the success of the project. Both objectives of the project were satisfactorily accomplished and the reception of our results, reported in the *Third International NASA Workshop on Planning and Scheduling for Space*, in September 2002 [18], was very positive.

The representation of the space shuttle's *RCS* system, which corresponds to the first objective above, is presented in Section 5.3 of Chapter 5; the design of the decision support system, *USA-Advisor*[4], the second objective mentioned, is discussed in Section 5.4 of the same chapter. This application involved a substantial amount of knowledge representation, as well as the design and implementation of some tasks, such as plan checking and actual planning. These tasks are discussed in Sections 5.5 and 5.6 of Chapter 5.

[4]The RCS/USA-Advisor system is available for download from:
http://krlab.cs.ttu.edu/~marcy/RCS/

1.3 Organization of the dissertation

This dissertation is organized in the following way. The next chapter presents the syntax and semantics of the A-Prolog language. The description and discussion about the design of the *A-Circuit* system is given in Chapter 3. Theorems and related proofs are presented in Chapter 4. The representation of the space shuttle's *RCS* system and the design of the *USA-Advisor* decision support system are described in Chapter 5. Conclusions, lessons learned, related and future work are discussed in Chapter 6. Appendix A presents tables and graphs summarizing the results of the experiments with the *RCS* system.

Chapter 2

The A-Prolog Language

> *"A representation is called epistemologically adequate for a person or machine if it can be used pratically to express the facts that one actually has about the aspect of the world. A representation is called heuristically adequate if the reasoning processes actually gone through in solving a problem are expressible in the language."*
>
> *John McCarthy and Patrick Hayes [138]*

The A-Prolog language, [71, 73], is a declarative logic programming language based on stable models/answer sets semantics of logic programs [74, 75]. A-Prolog allows the representation of defaults and multiple interesting aspects of reasoning about actions and their effects. We start by defining the syntax and semantics of A-Prolog as given in [71, 73].

2.1 Syntax

The syntax of A-Prolog is determined by a signature $\Sigma = \langle\ \mathbf{T},\ \mathbf{C},\ \mathbf{F},\ \mathbf{P}\ \rangle$ where $\mathbf{T}$, $\mathbf{C}$, $\mathbf{F}$, and $\mathbf{P}$ are sets of symbols. Members of the set $\mathbf{T}$ are called *types*. The set **C** contains *object constants* for each type in **T**. Symbols from sets **F** and **P** are typed functions and predicate constants, respectively. Each function symbol and predicate symbol has an associated integer called its *arity*. It is assumed that the signature contains symbols for integers and for the standard functions and relations of arithmetic. A *term* of Σ is either a typed object constant, or a string of the form $f(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms of the proper types, and f is a typed function symbol of arity n. An *atom* is a string of the form $p(t_1, \ldots, t_n)$, where p is a typed predicate symbol of arity n in Σ, and $t_1, \ldots, t_n$ are terms of the corresponding types. A *literal* is either an atom (also called a positive literal), or an atom preceded by $\neg$ (a negative literal). The symbol $\neg$ is called *classical* or *strong* negation. Literal $\neg a$ is read as "a is believed to be false," under the (epistemic) interpretation of logic programs of [75]. For a literal l, by $\overline{\neg l}$ we mean l, and by $\bar{l}$ we mean $\neg l$. Literals l and $\neg l$ are called contrary. Literals and terms not containing variables are called *ground*. The sets of all ground terms, atoms and literals over Σ are denoted by *terms*(Σ), *atoms*(Σ), and *lit*(Σ), respectively. For a set P of predicate symbols from Σ, *atoms*(P,Σ)(*lit*(P,Σ)) denote the sets of ground atoms (literals) of Σ formed with predicate symbols from P. A set of literals is said to be *consistent* if it does not contain contrary literals. Consistent sets of ground literals over signature Σ containing all arithmetic literals which are true under the standard interpretation of their symbols are called *states* of Σ and denoted by *states*(Σ).

A rule of A-Prolog is a statement of the form:

$$l_0 \leftarrow l_1, \dots, l_m, not\ l_{m+1}, \dots, not\ l_n \tag{2.1}$$

where $n \geq 1$, and l_i's are literals over Σ. Literal l_0 is called the *head* of the rule, and $l_1, \dots, l_m, not\ l_{m+1}, \dots, not\ l_n$ constitutes the *body* of the rule. The symbol *not* is a logical connective called *negation as failure* or *default negation*. An expression *not* l is read as "there is no reason to believe in l." The head l_0 can be either a literal or the symbol $\perp$. If $l_0 = \perp$, rule (2.1) is called a *constraint*. We frequently omit the head, $\perp$, of a constraint rule.

We assume that literals l_i in rules (2.1) are ground. We use rules with variables as a shorthand for the sets of their ground instantiations. Variables are denoted by capital letters.

A *logic program* is a pair $\{\Sigma, \Pi\}$ where Σ is a signature and Π is a collection of rules over Σ.

A literal $l \in lit(\Sigma)$ is *true* in a state S of Σ if $l \in S$; l is *false* in S if $\bar{l} \in S$. Otherwise, l is unknown. The symbol $\perp$ is false for any S.

2.2 Semantics

A program Π in A-Prolog can be viewed as a specification given to a rational agent for constructing beliefs about possible states of the world. Technically, these beliefs are captured by the notion of an *answer set* of program Π.

First, we give the precise definition of answer sets for programs whose rules do not contain negation as failure. Let Π be such a program and let S be a state of $\{\Sigma, \Pi\}$. Set S is said to be *closed* under Π if, for every rule $head \leftarrow body$ of Π, head is true in S whenever *body* is true in S. A constraint rule is closed under Π if its body is not contained in S.

Definition 2.1. ***(Answer set of programs without default negation)***
An answer set *of a program* Π*, consisting of rules not containing default negation, is the smallest set* S *of ground literals of* Σ *which satisfies the following two conditions:*

1. S *is closed under the rules of* $ground(\Pi)$*, i.e., for every rule (2.1) in* Π*, either there is a literal* l *in its body such that* $l \notin S$ *or its non-empty head* $l_0 \in S$*.*

2. *If* S *contains an atom* p *and its negation* $\neg p$*, then* S *contains all ground literals of the language.*

It is not difficult to show that there is at most one set ($Cn(\Pi)$) satisfying these conditions.

Now, let Π be an arbitrary ground program in A-Prolog. For any set S of ground literals of its signature Σ, let the *reduct* of Π relative to S, denoted Π^S, be the program obtained from Π by deleting:

(i) each rule that has an occurrence of *not* l in its body with $l \in S$,

(ii) all occurrences of *not* l in the bodies of the remaining rules.

Definition 2.2. ***(Answer set of arbitrary programs)***

Set S is an answer set of Π if

$$S = Cn(\Pi^S). \tag{2.2}$$

We are interested only in *consistent* programs, i.e., programs with at least one consistent answer set. Let S be an answer set of Π. A ground literal l is *true* in S if $l \in S$; *false* in S if $\neg l \in S$. This is expanded to conjunctions and disjunctions of literals in a standard way.

Definition 2.3. ***(Entailment)***

A program Π entails *a literal* l *(*$\Pi \models l$*) if* l *is true in all answer sets of* Π*. Program* Π *answers* yes *to a query* l *if* $\Pi \models l$*;* no *if* $\Pi \models \bar{l}$*, and* unknown *otherwise.*

Here are some examples. Assume that the signature Σ contains two object constants a and b. The program

$$\Pi_1 \left\{ \begin{array}{l} q(a). \\ \neg p(X) \leftarrow not\ q(X). \end{array} \right.$$

has the unique answer set $S = \{q(a), \neg p(b)\}$. The program

$$\Pi_2 \left\{ \begin{array}{l} p(a) \leftarrow not\ p(b). \\ p(b) \leftarrow not\ p(a). \end{array} \right.$$

has two answer sets, $\{p(a)\}$ and $\{p(b)\}$. The programs

$$\Pi_3 \left\{ \begin{array}{l} p(a) \leftarrow not\ p(a). \end{array} \right.$$

and

$$\Pi_4 \begin{cases} p(a). \\ \leftarrow p(a). \end{cases}$$

have no answer sets.

It is easy to see that programs of A-Prolog are nonmonotonic. For example consider program Π_1. We saw that $\Pi_1 \models \neg p(b)$, however, if some new information, $q(b)$, is added to the program, it forces the withdrawal of the previous conclusion $\neg p(b)$. The new program $\Pi_1 \cup \{q(b)\}$ has the unique answer set $\{q(a), q(b)\}$. Nonmonotonic reasoning is important for the representation of commonsense knowledge, and gives the means for reasoning about time and change. A-Prolog is closely connected with more general nonmonotonic theories. In particular, as was shown in [75, 129], there is a simple and natural mapping of programs in A-Prolog into a subclass of Reiter's default theories [169]. Similar results are also available for Autoepistemic Logic [150].

Next, we present some important theorems and lemmas that exhibit nice properties of A-Prolog programs. They will be frequently used in the proofs of Chapter 4. First, we introduce some necessary notation.

Let r be a rule of the form (2.1). By $head(r)$, $pos(r)$, and $neg(r)$ we denote $\{l_0\}$ and the sets $\{l_1, \ldots, l_m\}$, and $\{l_{m+1}, \ldots, l_n\}$, respectively. $lit(r)$ denotes the set $head(r) \cup pos(r) \cup neg(r)$. For a program Π, $lit(\Pi)$ denotes the set of literals occurring in Π.

For a program Π over the A-Prolog language, a set of literals A, over the language, is a splitting set of Π if for every rule $r \in \Pi$, $head(r) \cap A \neq \emptyset$ implies $lit(r) \subseteq A$.

Let A be a splitting set of Π. The *bottom of* Π *relative to* A, denoted by $b_A(\Pi)$, is the program consisting of all rules $r \in \Pi$ such that $lit(r) \subseteq A$.

Given a splitting set A for Π, and a set X of literals from $lit(b_A(\Pi))$, the *partial evaluation of* Π *by* X *with respect to* A, denoted by $e_A(\Pi, X)$, is the program obtained from Π as follows. For each rule $r \in \Pi \setminus b_A(\Pi)$ such that

1. $pos(r) \cap A \subseteq X$;

2. $neg(r) \cap A$ is disjoint from X;

there is a rule r' in $e_A(\Pi, X)$ such that

1. $head(r') = head(r)$, and

2. $pos(r') = pos(r) \setminus A$,

3. $neg(r') = neg(r) \setminus A$.

Let A be a splitting set of Π. A *solution to* Π *with respect to* A is a pair $\langle X, Y \rangle$ of set of literals satisfying the following two properties:

1. X is an answer set of $b_A(\Pi)$;

2. Y is an answer set of $e_A(\Pi \setminus b_A(\Pi), X)$;

3. $X \cup Y$ is consistent.

Theorem 1. ***(Splitting Set Theorem, [122])***

Let A be a splitting set for a program Π. A set A of literals is a consistent answer set of Π iff $A = X \cup Y$ for some solution $\langle X, Y \rangle$ to Π with respect to A. $\square$

The following example illustrates the notion of a splitting set and the use of the Splitting Set Theorem for the computation of answer sets of logic programs.

Let Π_0 be the program consisting of the following rules

$$\Pi_0 \begin{cases} r(b) \leftarrow q(a). \\ q(a) \leftarrow not\ p(a). \\ p(a) \leftarrow p(b). \\ p(b). \end{cases}$$

Set $A_0 = \{p(a), p(b)\}$ splits Π_0 into bottom program, $b_{A_0}(\Pi_0)$, and top program, $t_{A_0}(\Pi_0)$. The last two rules of Π_0 belong to the bottom, and the first two rules form the top. It is easy to see that the bottom program has the unique answer set $X = \{p(a), p(b)\}$. (Notice that $A_0 = X$ in this example, but this is not always the case.) The partial evaluation of the top with respect to A_0 and the answer set X of the bottom, denoted $e_{A_0}(\Pi, X)$, is obtained by dropping its second rule which is falsified by the negated subgoal $p(a)$. The result of the simplification is the program consisting of a single rule

$$e_{A_0}(\Pi_0, X) \begin{cases} r(b) \leftarrow q(a). \end{cases}$$

It is easy to see that the unique answer set of $e_{A_0}(\Pi_0, X)$ is $Y = \{\}$. Therefore, the only answer set for Π_0, denoted by $\mathcal{A}$, can be obtained by adding the unique answer set of the bottom, X, to Y, i.e.

$$\mathcal{A} = X \cup Y = \{p(a), p(b)\}.$$

Lemma 2.1. ***(Marek and Subrahmanian, [128])***

For any answer set S of a logic program Π consisting of rules of the form (2.1)

(a) *for any instance r of a rule of the type (2.1) from Π, if $pos(r) \subseteq S$ and $neg(r) \cap S = \emptyset$ then $head(r) \in S$;*

(b) *if S is consistent and $l_0 \in S$ then there exists an instance r of a rule of the type (2.1) from Π, such that $pos(r) \subseteq S$, $neg(r) \cap S = \emptyset$, and $head(r) = l_0$.* □

The previous example is used again to illustrate the applicability of Lemma 2.1 for the computation of answer sets of logic programs.

Let us take program Π_0. First, Lemma 2.1 will be used to compute an answer set of Π_0 as follows.

By condition (a) of Lemma 2.1 and the last rule of Π_0,

$$p(b).$$

it trivially follows that

$$p(b) \text{ must belong to all answer sets of } \Pi_0. \tag{2.3}$$

Since $p(b)$ is a consequence of Π_0, given condition (a) of Lemma 2.1, and the third rule of Π_0,

$$p(a) \leftarrow p(b).$$

we have that

$$p(a) \text{ must belong to all answer sets of } \Pi_0. \tag{2.4}$$

Statement (2.4) falsifies the second rule of Π_0

$$q(a) \leftarrow not\ p(a).$$

Because of this fact, and since there exists no other rule in Π_0 with head $q(a)$, it follows that

$$q(a) \text{ does not belong to any answer set of } \Pi_0. \tag{2.5}$$

Hence, no answer set of Π_0 can satisfy the first rule

$$r(b) \leftarrow q(a).$$

Given this fact, and since there exists no other rule in Π_0 with head $r(b)$, we can conclude that

$$r(b) \text{ does not belong to any answer set of } \Pi_0. \tag{2.6}$$

The above argument can be viewed as a construction of a set $\mathcal{A}$ which must be a subset of any answer set of Π_0. We will show that $\mathcal{A}$ is indeed an answer set of Π_0. To do that, let us compute the reduct of Π_0 with respect to $\mathcal{A}$, $\Pi_0^{\mathcal{A}}$. It consists of the following rules

$$\Pi_0^{\mathcal{A}} \left\{ \begin{array}{l} r(b) \leftarrow q(a). \\ p(a) \leftarrow p(b). \\ p(b). \end{array} \right.$$

It is easy to see that $\mathcal{A}$ is an answer set of $\Pi_0^{\mathcal{A}}$. Hence, by the definition of answer sets $\mathcal{A}$ is an answer set of Π_0. $\square$

Chapter 3

Digital Circuits in A-Prolog

> *"There are two ways of constructing a software design; one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."*
>
> *Sir Charles Antony Richard Hoare*

Digital circuits have been extensively studied. However, in most logical approaches, circuits are described by propositional formulas [143, 144]. In our work we use logic programming and build a general theory of digital circuits which contains standard knowledge about circuits from the electrical engineering field.

3.1 Digital Circuits in Electrical Engineering

We start by reviewing the meaning of some terms of the electrical engineering field that are used in this work.

An electronic *gate*, or component, is a device that realizes a logical function. Roughly, *circuit* is a collection of interconnected gates.

In the electrical engineering field, a *signal* is an impulse or a fluctuating electric quantity, such as voltage, or current, whose variations represent coded information. In the area of digital electronics, the precise values of voltage signals, either applied or generated by components and circuits, are not significant toward determination of the logical operation of the gates/circuits; in fact, these values vary from circuit to circuit and from component to component [105]. More importantly, electronic gates are limited by construction to recognize only two ranges of values, "high" and "low," which are, by convention, associated with constants 1 and 0, respectively.

Input (and conversily, output) has been used as a technical term for probably more than a century in the field of physics, then in electrical engineering, and more recently in computer science. In this thesis, input/output (of a component or a circuit) are applied to the domain of electrical engineering. Both terms have been largely misused, but normally each have one of two meanings when used in this domain. Input conveys: (a) energy or power, i.e. a signal, used to activate or drive a component/circuit, or (b) wire or pin at which a (input) signal enters a gate/circuit. By output it is meant: (a) energy or power, i.e. a signal, produced by a component/circuit, or (b) wire or pin at which a (output) signal produced by a gate/circuit is present. Even more confusely, it is possible that the term is used to indicate both concepts (as in items (a) and (b) mentioned before), simultaneously.

For clarity purposes, whenever refering to input[output] as energy or power, we use the expressions "input[output] signal," or "input[output] value"; and for indicating a wire or pin, we use the term “input[output] wire.”

A *circuit* is a collection of interconnected electric components, called *gates*, where the output signal present on the output wire of one component is used to actuate (stimulate) one or more input wires of other components.

A *combinational circuit* is a circuit whose output signals are functions of only the current circuit input signals.

The *propagation delay of a gate g* is the time required to propagate an input signal through g, or to switch the output of g from a value to another.

For simplicity of exposition we restrict this work to circuits that have a single output wire. This implies that the output wire of each and all gates in a circuit, with the exception of a single one, must be connected to at least one input wire of one, or more, gates in the circuit. Moreover, the time required to propagate the input signal values applied to the input wires of a circuit to its output wire will be referred to as the “propagation delay of the circuit.”

3.2 Formalization of Digital Circuits

Normally, computer science students start to study foundations of digital design in their first or second year at the university. First, they concentrate on combinational circuits which are constructed from simple boolean gates and are used to compute boolean functions. Given such a function $Y = f(X_1, \ldots, X_n)$, where Y and

$X_1, \ldots, X_n$ are boolean variables, students learn how to use propositional logic to construct a circuit C which *instantaneously* transforms the values $X_1, \ldots, X_n$ applied on its input wires $W_1, \ldots, W_n$ to the value Y on its output wire W_o. Later, they move to building more complex devices employing more complex, sequential circuits. The model of a circuit remains, however, essentially boolean with the only possible signals corresponding to 0 and 1, and basic gates still performing instantaneous transmission of information. In more advanced classes students normally "discover" that the boolean model they have learned is not always a realistic one. Gates suffer from physical limitations, i.e., do not instantaneously perform the function that they implement because of propagation (and other types) of delays. For a short time, the values of signals may lie somewhere between the levels necessary to classify them as 0 and 1, and will therefore be undefined. There are other situations where the analog (continuous and non-digital) character of gates and signals should be taken into account. To model such phenomena, scientists introduced the notion of a digital circuit with delays ([146, 201]) and three possible input values: 0, 1, and 1/2 (undefined) [201]. These circuits do not instantaneously produce the values of the corresponding functions. Instead, these values are produced after delays, which are determined by the circuit and the vector of input signals.

This approach mimics reality and allows input signal values $s = \{0, 1, u\}$, where u stands for an undefined value. In this case, input signal values $S_1, \ldots, S_n$, where $S_i \in s$, applied to input wires $W_1, \ldots, W_n$ of a circuit C, are converted by a function

$S = g(S_1, \ldots, S_n)$ to the output value S on the output wire W_o of C.

To make it usable for mathematical proofs, this explanation needs to be clarified.

Definition 3.1. *Let* $s = \{0, 1, u\}$ *be the set of possible signal values on wires of a circuit* C. *Let* $g : S^n \to S$ *be the function computed by* C *when values* $S_1, \ldots, S_n$ *are applied to input wires* $W_1, \ldots, W_n$ *of* C *at time* t. *Let* δ *be a non-negative integer. We say that circuit* C *computes* $g(S_1, \ldots, S_n)$ *with a delay* δ *if, in the absence of other inputs, the value on its output wire* W_o *at any time* $t' \geq t + \delta$ *is equal to* S. C computes *function* g *with a delay* δ *if it computes all the values of* g *with this delay.*

Notice that there are cases where even if some input signal value is undefined the circuit's output signal is a defined value. Figure 3.1 shows an example of a circuit with an input signal value undefined but whose output value is 0. The circuit consists only of a NOT and an AND gate with no delays. We show the graphical representation of the three basic gates NOT, AND, and OR on Figure 3.2 and their behavior, in the presence of the "undefined" (u) value, is presented in Table 3.1.

It is important to point out that our use of a 3-valued logic does not affect the *principle of duality* [105] which characterizes operations AND and OR from Boolean algebra.

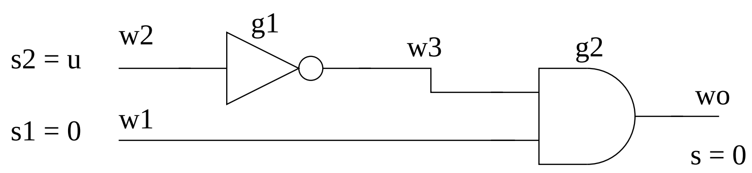


Figure 3.1: Digital circuit with undefined input and defined output.

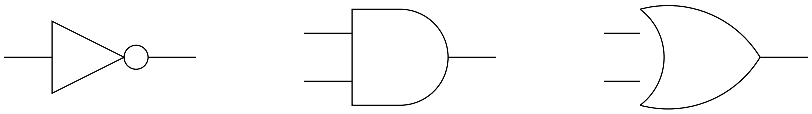


Figure 3.2: Symbolic representation of basic gates.

NOT gate	
Input	*Output*
0	1
1	0
u	u

AND gate		
Inputs		*Output*
I1	*I2*	
0	0	0
0	1	0
0	u	0
u	0	0
u	1	u
u	u	u
1	0	0
1	1	1
1	u	u

OR gate		
Inputs		*Output*
I1	*I2*	
0	0	0
0	1	1
0	u	u
u	0	u
u	1	1
u	u	u
1	0	1
1	1	1
1	u	1

Table 3.1: Definition of behavior of basic gates.

Introduction of delays and undefined signals bring to life a number of questions not present in the case of *ideal* (time independent) boolean circuits. We need to know for instance, how these δ's can be computed, how we can guarantee that a particular circuit computes g with a given δ, how we can check if a component of a circuit can be replaced by a similar component with a smaller/bigger delay without violating some important properties of the circuit, etc. To answer these and similar questions we need to have a precise description of the behavior of a circuit, which, given a vector of values applied to its input wires, will determine the values of signals present on *every* wire of the circuit at any moment of time. In the next section we design and implement a program in A-Prolog which does exactly that. One of the main advantages of using A-Prolog is that the program is very concise, clear, and elaboration tolerant. More importantly, in subsequent sections, we demonstrate that the expressive power of A-Prolog also allows for the description of a variety of tasks, e.g. computing maximum delay of a circuit and detection of glitches.

3.3 Formalizing Digital Circuits in A-Prolog

We start by introducing a simple language $\mathcal{L}_{ckt}$ for describing digital circuits. The language has four types of object constants (names for objects of the domain):

(a) $g_1, g_2 \ldots$ for gates;

(b) $w_1, w_2, \ldots$ for wires;

(c) $0, 1, u$ for signals;

(d) *and_gate*, *or_gate*, *not_gate* for the three basic gate types we chose to represent.

Variables for gates, wires, and signals will be denoted by possibly indexed letters G, W, and S, respectively. We also assume that $\mathcal{L}_{ckt}$ contains standard notation for numbers, needed to denote delays. To describe the geometry of the circuit we use statements of the form $output(W, G)$ and $input(W, G)$ read as "W is an output (input) wire of gate G." The types of gates in the circuit and the gates' delays are expressed by the statements $type\_of(G, gate\_type)$ (G is of type $gate\_type$) and $delay(G, D)$ (G has delay D). In this notation, the circuit from Figure 3.3 corresponds to the following collection of statements of A-Prolog:

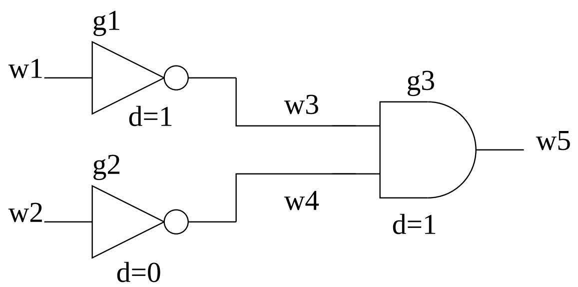


Figure 3.3: Graphical representation of a digital circuit.

$type\_of(g_1, not\_gate).$
$type\_of(g_2, not\_gate).$
$type\_of(g_3, and\_gate).$
$delay(g_1, 1).$
$delay(g_2, 0).$
$delay(g_3, 1).$
$input(w_1, g_1).$
$input(w_2, g_2).$
$input(w_3, g_3).$
$input(w_4, g_3).$
$output(w_3, g_1).$
$output(w_4, g_2).$
$output(w_5, g_3).$

We denote such a representation of a circuit C by $\pi(C)$.

To describe the dynamic behavior of the circuit we need to introduce the notion of time. AI researchers developed a large variety of different models of time. For our purposes, we assume the discrete linear time model in which time is represented by non-negative integers. We view the application of signals to the input wires of a circuit as the execution of an action which changes the previous signals on these wires. This triggers a process of signal propagation through the circuit which goes uninterrupted unless the input signals are changed again. In this way, describing the behavior of the circuit can be reduced to specifying effects of the corresponding actions as it is done in action theories of AI (see for instance [77, 124, 132, 162, 172, 194]). In these theories, dynamic domains consist of actions and fluents (properties whose values depend on time). Action theories are built to specify the values of fluents at an arbitrary moment t, given their values at moment 0 and the domain history (a sequence of actions performed in the domain in the past). In our domain we have only one (parameterized) action $apply(w, s)$ and one (parameterized) propositional fluent, $value(w, s)$. A statement $occurs(apply(w, s), t)$ says that at moment t signal s is applied to wire w, while a statement $holds(value(w, s), t)$ denotes that at moment t the value of the signal on wire w is s. We will also use an auxiliary relation $opposite(s_1, s_2)$ satisfied by the pairs $[0, 1]$, $[1, 0]$ and $[u, u]$, where u corresponds to undefined values between 0 and 1. Direct effects of actions will be represented in A-Prolog by the following rule:

$$holds(value(W, S), T + 1) \quad \leftarrow \quad occurs(apply(W, S), T). \tag{3.1}$$

Here T is a variable for time. To guarantee the computability of our models we assume that T ranges between 0 and some fixed time denoted by the constant *lasttime*. (This constant can be viewed as a parameter of our system and it is entered by the user via the entry program as a part of the problem instance.) The next rule describes the propagation of the applied signal through the NOT gate of the circuit.

$$
\begin{array}{rcl}
holds(value(W_2, S_2), T+D) & \leftarrow & type\_of(G, not\_gate), \\
 & & delay(G, D), \\
 & & input(W_1, G), \\
 & & output(W_2, G), \\
 & & opposite(S_1, S_2), \\
 & & holds(value(W_1, S_1), T).
\end{array}
$$

Auxiliary predicate $opposite(S, S')$ is used only for conciseness of representation. It can be eliminated, in which case there would be three such rules to represent the propagation of a signal through a gate NOT, instead of the single rule above.

To represent the function of gates AND and OR, we need to define some auxiliary relations. The first relation, $not\_all\_inputs(G, S, T)$, holds if at moment T some input wire of the gate G has a signal different from S. This can be expressed by the following rule:

$$
\begin{array}{rcl}
not\_all\_inputs(G, S_1, T) & \leftarrow & input(W, G), \\
 & & S_1 \neq S_2, \\
 & & holds(value(W, S_2), T).
\end{array}
$$

The second relation, $all\_inputs(G, S, T)$, holds if at time T all the input wires of G have value S, and is defined by the rule:

$$all\_inputs(G,S,T) \leftarrow not\ not\_all\_inputs(G,S,T).$$

Finally, the relation $contains\_input(G,S,T)$ holds if at moment T at least one input wire of G has value S, and is defined by the rule:

$$\begin{aligned} contains\_input(G,S,T) \leftarrow\ & input(W,G), \\ & holds(value(W,S),T). \end{aligned}$$

Now we can define the propagation of signals through AND gates:

$$\begin{aligned} holds(value(W,1),T+D) \leftarrow\ & type\_of(G,and\_gate), \\ & delay(G,D), \\ & output(W,G), \\ & all\_inputs(G,1,T). \end{aligned}$$

$$\begin{aligned} holds(value(W,0),T+D) \leftarrow\ & type\_of(G,and\_gate), \\ & delay(G,D), \\ & output(W,G), \\ & contains\_input(G,0,T). \end{aligned}$$

$$\begin{aligned} holds(value(W,u),T+D) \leftarrow\ & type\_of(G,and\_gate), \\ & delay(G,D), \\ & output(W,G), \\ & not\ contains\_input(G,0,T), \\ & contains\_input(G,u,T). \end{aligned}$$

The rules for propagation of signals through OR gates are defined next.

$$\begin{array}{lcl} holds(value(W,0),T+D) & \leftarrow & type\_of(G,or\_gate), \\ & & delay(G,D), \\ & & output(W,G), \\ & & all\_inputs(G,0,T). \end{array}$$

$$\begin{array}{lcl} holds(value(W,1),T+D) & \leftarrow & type\_of(G,or\_gate), \\ & & delay(G,D), \\ & & output(W,G), \\ & & contains\_input(G,1,T). \end{array}$$

$$\begin{array}{lcl} holds(value(W,u),T+D) & \leftarrow & type\_of(G,or\_gate), \\ & & delay(G,D), \\ & & output(W,G), \\ & & not\ contains\_input(G,1,T), \\ & & contains\_input(G,u,T). \end{array}$$

All the above rules define the effects of changes caused in the circuit by applying new signals to its input wires. To complete our program we need to specify when the values of fluents do not change. The task of finding a compact way to specify this in a formal language is called the *frame problem*. J. McCarthy in [138] suggested that this problem is closely related to the problem of representing a particular default called the *law of inertia*. The law says that "normally, things stay as they are," i.e., in dynamic domains fluents do not change their values unless they are forced to. Fortunately, the methodology of representing defaults in A-Prolog is now well understood and can be applied to obtain a simple and natural solution to the frame problem for our domain.

The solution is given by the next two rules.

The first of them is the Law of Inertia:

$$\begin{array}{rcl} holds(value(W,S),T\!+\!1) & \leftarrow & holds(value(W,S),T), \\ & & not\ \neg holds(value(W,S),T\!+\!1). \end{array}$$

This rule allows the reasoner (the program) to assume that the value of a signal on a wire W does not change from one moment to the next, unless it is forced to believe otherwise. The second rule states that there may be at most one signal present on a wire at a given moment of time:

$$\begin{array}{rcl} \neg holds(value(W,S_1),T) & \leftarrow & S_1 \neq S_2, \\ & & holds(value(W,S_2),T). \end{array}$$

Rules of this sort are often called "state constraints". They play an important role in theory of action languages and are mainly responsible for the conciseness of the representation of indirect effects of actions.

We denote the resulting program by CT and call it the *simple circuit theory.* The theory, in conjunction with the specification of a circuit and its history up to the current moment t_c, can be used to specify the values of signals on the circuit wires at an arbitrary moment $0 \leq t \leq lasttime$. We call such a specification a *domain description* at time t_c. It consists of the encoding of a circuit in language $\mathcal{L}_{ckt}$ (see Figure 3.3) together with statements of the form:

$$occurs(apply(w,s),t).$$

where $0 \leq t \leq t_c$. We assume that the initial signals of the circuit are undefined. This assumption can be represented in A-Prolog by facts of the form:

$$holds(value(W, u), 0).$$

which are added to the CT theory. We assume that domain descriptions used in conjunction with CT are consistent, i.e., do not contain physical impossibilities such as: two different signals applied to the same wire at the same time, multiple input wires for the NOT gate, etc.

This can be ensured by expanding the program with the following constraints:

- different signals can not be applied to a single wire simultaneously;

  :- $occurs(apply(w, s), 0),\ occurs(apply(w, s'), 0),\ s \neq s'.$

- the type of a gate is unique;

  :- $type\_of(g, y),\ type\_of(g, y'),\ y \neq y'.$

- there is a unique (propagation) delay associated to each gate;

  :- $delay(g, d),\ delay(g, d'),\ d \neq d'.$

- each gate has a unique output wire;

  :- $output(w, g),\ output(w', g),\ w \neq w'.$

- an output wire can not belong to more than one gate.

  :- $output(w, g),\ output(w, g'),\ g \neq g'.$

Using standard mathematical techniques recently developed by researchers in logic programming and non-monotonic reasoning, it is not difficult to show that for any consistent domain description $\mathcal{D}$, the program $P_0 = CT \ \cup \ \mathcal{D}$ has exactly one consistent answer set. By $CT(\mathcal{D})$ we denote the set of all atoms, formed by predicate symbol *holds*, which belong to this answer set. The set $CT(\mathcal{D})$ can be viewed as a *specification of a dynamic behavior of a combinational circuit with delays.*[1] Let us first show that our specification correctly captures the behavior of "ideal" combinational circuits.

Proposition 3.1. *Let C be a combinational circuit, with input wires $w_1, \ldots, w_n$, output wire w_o, and no delays, which computes a function $f(S_1, \ldots, S_n)$. Then for any input vector $s_1, \ldots, s_n$ of 0's, 1's, and u's, program P_0 has a unique answer set and $holds(value(w_o, s), 1) \in CT(\mathcal{D})$ if and only if $s = f(s_1, \ldots, s_n)$, where $\mathcal{D} = \pi(C) \cup \{occurs(apply(w_i, s_i), 0) \ : \ w_i \in \{w_1, \ldots, w_n\}, s_i \in \{0, 1\}, 1 \leq i \leq k, k \leq n\}$.*

□

The proof of Proposition 3.1, presented in Chapter 4, is by induction on the number $m + 1$ of gates of circuit C. We decompose C into circuits C_1 and C_m containing 1 and m gates respectively, and show that

(a) their corresponding programs P_1 and P_m have unique answer sets, $\mathcal{A}_1$ and $\mathcal{A}_m$;

(b) $holds(value(w_o, s), 1) \in \mathcal{A}_m$ if and only if $s = f(s_1, \ldots, s_n)$.

[1] The above definition works only for circuits computing a "single value" function, i.e., a function returning 0, 1, and u. This restriction is only for simplicity of presentation. All the definitions and programs can be easily extended to functions returning vectors of signal values.

We also show that P_0 is equivalent to $P_1 \cup P_m$, and therefore, by the Splitting Set Theorem [122], we conclude that program P_0 has a unique answer set, $\mathcal{A}_0 = \mathcal{A}_1 \cup \mathcal{A}_m$. From this and (b), it follows that $holds(value(w_o, s), 1) \in \mathcal{A}_0$ if and only if $s = f(s_1, \ldots, s_n)$.

Any combinational circuit C with delays has its *ideal counterpart*, $i(C)$ obtained from C by setting all of the gate delays of C to 0. The following proposition guarantees that for any input vector, $s_1, \ldots, s_n$, the output signal of C will eventually stabilize at the value of $f(s_1, \ldots, s_n)$ where f is the function defined by the ideal counterpart of C. More precisely,

Proposition 3.2. *Let C be a combinational circuit with input wires $w_1, \ldots, w_n$ and output wire w_o, and let $f(S_1, \ldots, S_n)$ be a function computed by its ideal counterpart $i(C)$. Then there is a delay, δ, such that for any $t \geq \delta$ and any input vector $s_1, \ldots, s_n$ of 0's, 1's, and u's, program P_0 has a unique answer set and $holds(value(w_o, s), t) \in CT(\mathcal{D})$ if and only if $s = f(s_1, \ldots, s_n)$, where $\mathcal{D} = \pi(C) \cup \{occurs(apply(w_i, s_i), 0) : w_i \in \{w_1, \ldots, w_n\}, s_i \in \{0, 1\}, 1 \leq i \leq k, k \leq n\}$.* □

Proposition 3.2, follows immediatelly from Propositions 3.1 and 3.3.

3.4 Computing the Maximum Delay of a Circuit

The circuit delay from the above proposition can be found constructively. This can be done by another A-Prolog program, Δ, shown in Figure 3.4.

The program is based on a simple algorithm for computing circuit delays which can

$$
\begin{array}{lcl}
is\_input\_wire(W) & \leftarrow & input(W, G), \\
 & & not\ is\_output(W). \\[1em]
is\_output(W) & \leftarrow & output(W, G). \\[1em]
is\_output\_wire(W) & \leftarrow & output(W, G), \\
 & & not\ is\_input(W). \\[1em]
is\_input(W) & \leftarrow & input(W, G). \\[1em]
in\_gate(G) & \leftarrow & is\_gate(G), \\
 & & not\ inner\_gate(G). \\[1em]
inner\_gate(G) & \leftarrow & input(W, G), \\
 & & \neg is\_input\_wire(W). \\[1em]
\neg is\_input\_wire(W) & \leftarrow & input(W, G1), \\
 & & output(W, G2). \\[1em]
out\_gate(G) & \leftarrow & is\_output\_wire(W), \\
 & & output(W, G). \\[1em]
out\_delay(G, N) & \leftarrow & in\_gate(G), \\
 & & delay(G, N). \\[1em]
in\_delay(G_2, N) & \leftarrow & output(W, G_1), \\
 & & input(W, G_2), \\
 & & out\_delay(G_1, N). \\[1em]
\neg max\_in\_delay(G, N) & \leftarrow & in\_delay(G, N), \\
 & & in\_delay(G, M), \\
 & & M > N. \\[1em]
max\_in\_delay(G, N) & \leftarrow & in\_delay(G, N), \\
 & & not\ \neg max\_in\_delay(G, N). \\[1em]
out\_delay(G, N) & \leftarrow & max\_in\_delay(G, N_1), \\
 & & delay(G, N_2), \\
 & & N = N_1 + N_2. \\[1em]
circuit\_delay(N) & \leftarrow & out\_gate(G), \\
 & & out\_delay(G, N).
\end{array}
$$

Figure 3.4: Program to compute maximum delay of a circuit.

be found in standard introductory texts on digital logic ([105, 175, 201]). The result is not necessarily optimal, but it may serve as a good practical approximation. (It is instructive to notice how rules of A-Prolog are used to encode recursive definitions.) Again, it is not difficult to show that the program $P_0 \cup \Delta \cup \pi(C)$ has exactly one answer set and that the answer set contains exactly one atom formed by the predicate symbol *circuit_delay*. Let us denote this atom by *circuit_delay*(d). We call number d the *computed delay* of C and denote it by $\delta(C)$.

Now we can state the following proposition.

Proposition 3.3. *Let C be a combinational circuit with input wires $w_1, \ldots, w_n$ and output wire w_o, and let $f(S_1, \ldots, S_n)$ be a function computed by its ideal counterpart $i(C)$. Then for any $t \geq \delta(C)$ and any input vector $s_1, \ldots, s_n$ of 0's, 1's, and u's, $P_0 \cup \Delta \cup \pi(C)$ has a unique answer set and $holds(value(w_o, s), t) \in CT(\mathcal{D})$ and $circuit\_delay(\delta(C)) \in CT(\mathcal{D})$ if and only if $s = f(s_1, \ldots, s_n)$, where $\mathcal{D} = \pi(C) \cup \{occurs(apply(w_i, s_i), 0) \mid w_i \in \{w_1, \ldots, w_n\}, s_i \in \{0, 1\},\ 1 \leq i \leq k,\ k \leq n\}$.* $\square$

The proof of Proposition 3.3 is similar to the proof of Proposition 3.1.

In order to compute the maximum delay of a circuit, a user can utilize the graphical interface of A-Circuit to specify the circuit and choose this task to be performed.

3.5 Using the Circuit Theory CT

The discussion in the previous section was limited to the use of declarative semantics of A-Prolog for specifying the behavior of digital circuits. Thanks to the existence of inference engines for A-Prolog, like SMODELS [156], DLV [41], and CMODELS [8], this specification can be combined with simple reasoning programs aimed at solving various design tasks, and it can also be actually executed. We present some examples of such programs next, and in Chapter 5, a "real world" application where our theory of digital circuits is utilized.

3.5.1 Simulating the circuit

In many cases, it may be instructive for a student to see the simulated behavior of the circuit. Ideally, this should be an easy task: the student specifies the circuit and its history using a graphical interface. The corresponding domain description $\mathcal{D}$, combined with CT, is given as an input to one of the A-Prolog inference engines, say SMODELS, which computes the program's unique answer set. The circuit behavior defined by $CT(\mathcal{D})$ is extracted from the answer set and displayed in graphical and numerical form on the screen. The reality is rather close to the ideal situation, but not identical to it. The reason is that different inference engines have different restrictions on the programs needed to guarantee their soundness and completeness with respect to the semantics of A-Prolog. This implies that CT (from the previous section) needs to be slightly modified for the use of SMODELS. Fortunately, the modification is simple and basically amounts to replacing our typed variables by the explicit types (see [16] for details.)

After this modification is done, the resulting system will produce the output shown in

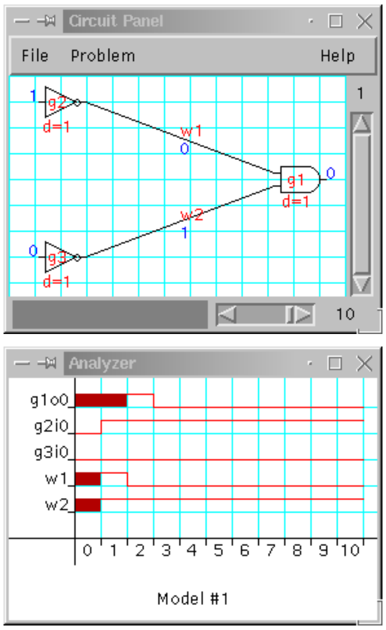


Figure 3.5: (a) Output in numerical form. (b) Timing Analysis.

Figure 3.5, when given the description of the circuit from Figure 3.3 and the following sequence of input values: $[0, 0]$ applied on $[w_1, w_2]$ at time 0, and 1 applied on w_1 at time 1.

The timing analysis output screen in Figure 3.5(b), shows the propagation of symbols through the circuit up to moment 10. This graphical representation helps the student to visualize and better understand the dynamic behavior of the circuit.

3.5.2 Avoiding hazards

One interesting problem when dealing with digital circuits involving delays is the occurrence of transient incorrect signal values, called *glitches*, on some of the circuit wires. A hazard is said to exist when a circuit has a possibility of producing such a

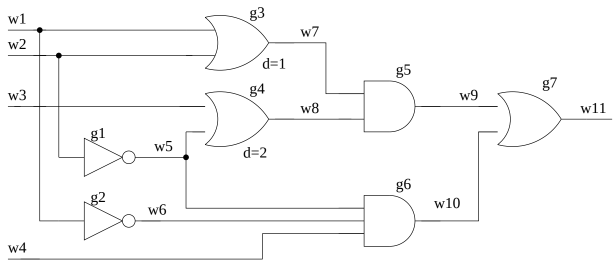


Figure 3.6: Circuit with a hazard.

glitch. A logic designer must be prepared to eliminate hazards even though a glitch may occur only under the worst-case combination of logical and electrical conditions [201]. We briefly describe a declarative program for the detection of a particular form of hazard. Combined with the inference engine of SMODELS this gives us a new algorithm for finding hazards different from the known algorithms (see [139]). Again, we believe that the program is sufficiently clear and the algorithm is reasonably efficient to help a student to understand the phenomenon.

We say that a circuit C, computing boolean function f is *hazardous* if there are two vectors, I_1 and I_2, of input signals which differ on the value of exactly one input wire[2], and during the transition period the value on the output wire of C changes to a signal different from $f(I_2)$.

To better understand this notion let us consider the circuit in Figure 3.6, taken from [201].

[2]We call a consecutive application of such input signals to C a *simple transition*.

In this circuit, there are 3 paths from input wire w_2 to output wire w_{11}. We assume that all gates, except g_3 and g_4, have delay 0. The delay of g_3 is 1 and the delay of g_4 is 2. Two of the paths go through these slower gates and affect the output signal. To understand how, let us consider the following evolution of the circuit signals. (a) Applying input signals $[0, 0, 0, 1]$ to input wires $[w_1, w_2, w_3, w_4]$ causes the output signal to become 1 at time 0. If we change (b) the value on input wire w_2 to 1 at time 1, this change is propagated through the circuit and makes the output value of C become 0 at time 1. However, the output value of gate g_3 is delayed by 1 time unit and (c) will force the output of the circuit to change again to 1 at time 2. Then, (d) the output of the slower gate g_4, with delay 2, also changes, forcing the circuit output signal to finally reach value 0 at time 3. Therefore, a single transition on input wire w_2 caused the values of output wire w_{11} to change three times, as follows: $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$.

Our goal now is to define hazardous circuits in A-Prolog. We construct a program, GD (which stands for "glitch detector"), such that $P_0 \cup GD \cup \pi(C)$ have an answer set if and only if a circuit C is hazardous.

We assume that w is the output wire of circuit C and that there is a relation $required\_output(s)$ such that for any domain description $\mathcal{D}$, $required\_output(s)$ belongs to the answer set of $CT \cup \mathcal{D}$ if and only if s is the output signal of the ideal counterpart $i(C)$ of C. Suppose now we are given a history H, of input signal values applied to C, containing a simple transition from I_1 to I_2. Then, *by definition*, this transition causes a glitch if the following condition holds:

$$
\begin{array}{rcl}
glitch & \leftarrow & required\_output(S_1), \\
& & holds(value(w, S_1), T_1), \\
& & holds(value(w, S_2), T_2), \\
& & S_1 \neq S_2, \\
& & T_2 > T_1.
\end{array}
$$

Adding the above rule together with a constraint

$$\leftarrow \quad not\ glitch.$$

to GD ensures that if C is safe (i.e., has no hazard) then $P_0 \cup GD \cup \pi(C)$ have no answer set.

To complete the construction of GD we need to generate histories containing possible simple transitions and check that they do not contain glitches. This can be done by first generating possible input vectors applied to C at moment 0, which is achieved by the rules:

$$
\begin{array}{rcl}
occurs(apply(W, 1), 0) & \leftarrow & is\_input\_wire(W), \\
& & not\ occurs(apply(W, 0), 0).
\end{array}
$$

$$
\begin{array}{rcl}
occurs(apply(W, 0), 0) & \leftarrow & is\_input\_wire(W), \\
& & not\ occurs(apply(W, 1), 0).
\end{array}
$$

which say that for each input W of C, either a signal value 1 or a signal value 0, is applied to W at time 0.

Then, we proceed by introducing a new relation $change(W)$ which holds when at moment 1 the signal applied to wire W at 0 is changed to its opposite.

$$\begin{array}{rcl} occurs(apply(W, S_1), 1) & \leftarrow & change(W), \\ & & occurs(apply(W, S_2), 0), \\ & & opposite(S_1, S_2). \end{array}$$

To ensure that histories generated by our program contain only simple transitions we need to add the following rule:

$$change(w_1) \text{ or } \ldots \text{ or } change(w_k),$$

where $w_1, \ldots, w_k$ is the list of the input wires of C. The DLV [41] inference engine would understand this rule and would properly compute the corresponding answer sets. However, to make it work for SMODELS[3] we need to eliminate the disjunction, which can be done by the following rules:

$$\begin{array}{rcl} change(W) & \leftarrow & is\_input\_wire(W), \\ & & not\ other\_changed(W). \end{array}$$

$$\begin{array}{rcl} other\_changed(W) & \leftarrow & change(W_1), \\ & & W \neq W_1. \end{array}$$

As mentioned in Chapter 1, for efficiency reasons, this rule is written in the form of a "choice rule" of the language of SMODELS, as follows:

$$1\{change(W) : is\_input\_wire(W)\}1.$$

Let GD be the program consisting of the rules of CT, which were introduced in this subsection, and the definition of the relation *required_output*.

[3]Notice that disjuntive rules were recently added to the language of SMODELS.

Proposition 3.4. *A combinational circuit C is hazardous if and only if*

$$P_0 \cup GD \cup \pi(C)$$

is consistent, i.e., has an answer set. $\square$

The proof of Proposition 3.4 is similar to the proof of Proposition 3.1.

Notice that each answer set describes a simple transition causing a glitch and the signals propagation through the circuit. The graphical interface allows the user to specify a circuit and request it to be checked for glitches.

The simple theory for circuits, CT, can be used in a similar way to solve other problems associated with digital design. CT, along with various reasoning modules, can be used to decide what signals should be applied to the input wires of a circuit to produce the desired output, to find malfunctioning components responsible for the incorrect behavior of a circuit, to simulate certain forms of sequential circuits, etc. In the following chapter, we demonstrate how it can be integrated in a pratical system and applied to obtain such results.

3.6 Graphical Interface for A-Circuit

To simplify the user/program interface we implemented[4] a schematic entry program, written in Java, which allows the user to:

[4]The graphical interface for the A-Circuit system was implemented primarily by Marcello Balduccini.

- draw a circuit diagram by choosing from the options available on the ToolBox Window (shown in Figure 3.7(a)). For example, Figure 3.8(b) shows how the circuit diagram presented in Figure 3.3 appears on the graphical interface.

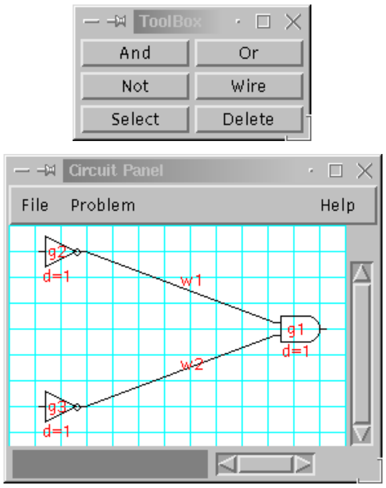


Figure 3.7: (a) ToolBox Window. (b) The complete circuit.

- automatically translate a circuit drawing to the corresponding A-Prolog representation.

- specify the circuit's input values graphically.

- eliminate the possibility of inconsistent data to be entered into the corresponding domain description. In particular, the graphical interface does not allow:

  - assigning more than a single type to a gate;
  - associating more than a single propagation delay with a gate;
  - creating gates with more than a single output wire;

  - assigning the same output wire to more than one gate;
  - applying different signals to a single wire simultaneously.

- compute the maximum delay of a circuit.
- check a circuit for glitches. For example, in the circuit C from Figure 3.6, the program returns a message box informing the user that the circuit is hazardous, and it also graphically shows, via the Analyzer Window, the situations in which the glitch occurs, (see Figure 3.8.)

3.7 Related Work

The relationship between logic and combinational circuits is not new. The connection was established by Shannon [181, 182] who developed the algebra of switching circuits, and showed its relation to the calculus of propositions and Boolean algebra.

The relationship allowed standartization of circuits and the use of various logic-based algorithms in the circuit design. For instance, boolean minimization algorithms [105] are commonly used to construct the desired circuits with the minimal number of gates, or other nice properties. Boolean logic, however, is only used for specification of circuits without delays. More detailed analysis requires the introduction of time and three valued logic. As we have shown, A-Prolog seems to be a natural tool for analysis of circuits on this level of abstraction.

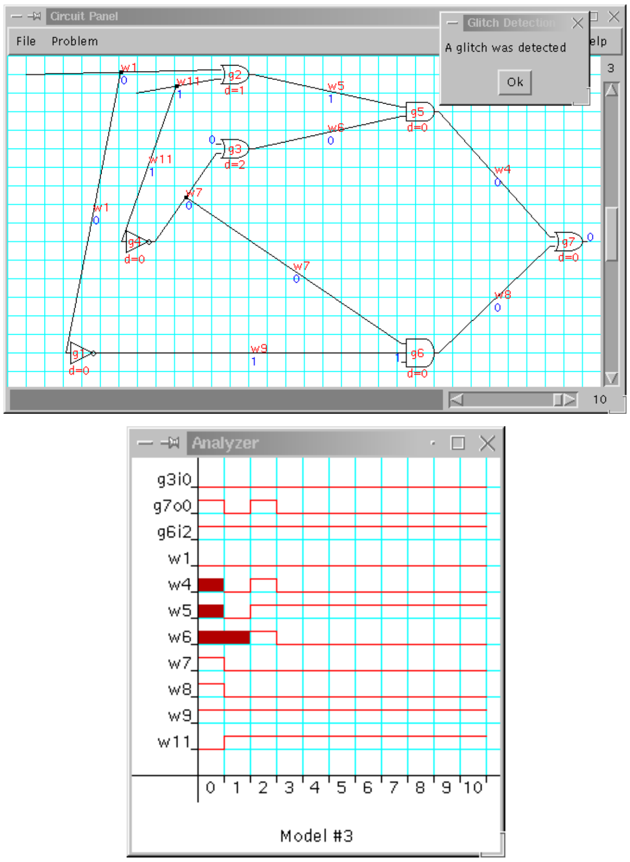


Figure 3.8: Interface output for glitch detection problem.

Another relationship between logic and combinational circuits, a much more recent one, can be found in [143], where Intuitionistic Logic is applied to the timing analysis of digital (combinational) circuits. The author uses a fairly complex intuitionstic modal logic to model circuits with delays depending on both, the properties of the gate and on its input signal values. In contrast, our model only considers delays independent from the input values. It seems, however, that a simple modification of our theory of circuits will cover these, more complex, delays. Moreover, unlike our formalization, the modeling mechanism of [143] does not suggest any logical algorithms for tasks different from the simple timing analysis of the circuit, e.g. discovery of glitches and other types of diagnosis.

Our work also has rather close connections with Hardware Description Languages (HDLs) [84]. In industry, digital designers use HDLs to represent large digital circuits in several different levels of abstraction. There are systems that support these languages to perform various tasks, in particular, simulation. The most popular HDLs today are *VHDL* [101] and *Verilog HDL* [159] which are used in serious applications for design, simulation and a limited type of synthesis of digital circuits. These languages, especially *Verilog*, which has became of public domain after the introduction of *VHDL*, are also used in classrooms for teaching several disciplines, e.g. digital design.

As mentioned in the introduction, the relative complexity of these languages makes it difficult for students to rapidly represent and simulate even simple circuits. Normally, the tools available to students do not have a graphical interface to speed up the circuit's description or the specification of the input stimuli. These steps must be realized prior to performing any simulation task.

Classroom projects involve circuits' descriptions which are comparable in size to the ones that we can realize with the A-Circuit system. Combining circuits together can also be done in the A-Prolog language in a fairly easy way. The graphical interface of the A-Circuit tool permits a speedy representation of a circuit and the specification of the input stimuli to be utilized in a simulation. In addition, A-Circuit permits rapid prototyping and has a variety of tasks available to users interested in different properties of a digital circuit. These characteristics makes the A-Circuit system a very attractive tool for teaching digital design and related classes.

At the moment, A-Circuit is only apropriate for the gate level of abstraction. In principle, it can be expanded to other levels of abstraction, althouth this was not an objective of this project. HDL languages like *VHDL* and *Verilog* are really more expressive and allow the specification of many more properties of digital circuits then the simple portion we can cover with our system. On the other hand, the A-Circuit tool allows checking a circuit for glitches and other types of analysis that are not readily available for HDLs.

In the next chapter, we present an extension to the Circuit Theory described in this chapter which incorporates additional types of gates. Moreover the modified theory allows us to do more complex diagnosis of digital circuits. Even though these and many other extensions are rather natural it is not clear if our representation can suggest any algorithms for the synthesis of digital circuits. Finding such methods is an interesting open problem.

Chapter 4

Proofs for A-Circuit

"No matter how correct a mathematical theorem may appear to be, one ought never to be satisfied that there was not something imperfect about it until it also gives the impression of being beautiful."

George Boole (1815–1869)

4.1 Problem Formulation

Consider a circuit C with input wires, $w_i, \ldots, w_n$, and input signals vector $\mathcal{I} = \{s_1, \ldots, s_n\}$, such that $s_i \in \{0, 1, u\}$ for $1 \leq i \leq n$. Signals 0 and 1 are called *definite*, while u is an *undefined* signal.

A *circuit description* $\pi(C)$ over a circuit signature Σ (defined in Chapter 2) is a collection of atoms of the form:

- *type_of(g,y)* denotes that the type of gate g in C is y;
- $delay(g, d)$ denotes that d, a natural number, is the delay associated with gate g;

- $input(w, g)$ denotes that w is an input wire of gate g;
- $output(w, g)$ denotes that w is the output wire of gate g;

By an *observation*, $\mathcal{O}(\mathcal{I})$ we mean a set

$$\{occurs(apply(w_i, s_i), 0) : s_i \in \mathcal{I}, s_i \neq u\}.$$

An observation is used to denote a "definite" input of the circuit at time 0.

By *domain description* $\mathcal{D}(C, \mathcal{I})$, we mean

$$\mathcal{D}(C, \mathcal{I}) = \pi(C) \ \cup \ \mathcal{O}(\mathcal{I}),$$

where $\pi(C)$ is a circuit description and $\mathcal{O}(\mathcal{I})$ is an observation.

A domain description is called *consistent* if it satisfies the following constraints:

- different signals can not be applied to a single wire simultaneously; represented as a logic programming constraint as:

  :- $occurs(apply(w, s), 0),\ occurs(apply(w, s'), 0),\ s \neq s'$.

- the type of a gate is unique;

  :- $type\_of(g, y),\ type\_of(g, y'),\ y \neq y'$.

- there is a unique (propagation) delay associated to each gate;

  :- $delay(g, d),\ delay(g, d'),\ d \neq d'$.

- each gate has a unique output wire;

  :- $output(w, g),\ output(w', g),\ w \neq w'$.

- an output wire can not belong to more than one gate.

  :- $output(w, g),\ output(w, g'),\ g \neq g'$.

The description of the dynamic behaviour of a circuit C over time is reduced to specifying the effects of actions which apply signal values to the input wires of C. These signals are propagated through the circuit without interruption until the input signals are changed by the application of new actions.

The logic program formed by the ground instances of rules (4.1)−(4.14) below is called the *simple circuit theory* CT_0. The rules of CT_0 can be divided into the following groups: dynamic and static causal laws, law of inertia, initial situation and auxiliary relations. In all the rules, variables W, G stand for wires, and gates, respectively; S, S' are variables for signals, while variable $\overline{S}$ stands for the signal opposite to signal S; and variable T denotes time and belongs to interval $[0, 1]$, since we are interested only in moments of time 0 and 1.

1. Each ground instance of rule (4.1) is called a *dynamic causal law*, and expresses that the effect of action *apply* signal S to input wire W at time T is that *value* S *holds* on input wire W at time $T+1$.

$$holds(value(W, S), T+1) \ \ :\!-\ \ occurs(apply(W, S), T). \tag{4.1}$$

2. Rules (4.2 – 4.8) are known as *static causal laws*. They express the indirect effects of applying signals to the input wires of the following gates:

   (a) NOT gate

   Rule (4.2) says that if S is the signal value present at input wire W_1 of a NOT gate G at time T, and G has a (propagation) delay D, then signal value $\overline{S}$ (the opposite signal to S) will be present at the output wire W of G at time $T+D$.

$$\begin{aligned} holds(value(W,\overline{S}),T+D) \ \ \text{:-} \ \ & type\_of(G,notg), \\ & delay(G,D), \\ & input(W_1,G), \\ & output(W,G), \\ & opposite(S,\overline{S}), \\ & holds(value(W_1,S),T). \end{aligned} \tag{4.2}$$

   (b) AND gate

   Given an AND gate G with (propagation) delay D and output wire W, rule (4.3) expresses that if signal 1 is present (or holds) on all input wires of G at time T, then signal 1 will hold at W at time $T+D$. Rule (4.4) says that if signal 0 holds on at least one of the input wires of G, then signal 0 will hold at W at time $T+D$. Rule (4.5) expresses that if signal 0 is not present at any of the input wires of G, but signal u holds on at least one of G's input wires, then signal u will also hold at W at time $T+D$.

$$holds(value(W,1),T{+}D) \;\text{:-}\; \begin{array}{l} type\_of(G, andg), \\ delay(G,D), \\ output(W,G), \\ all\_inputs(G,1,T). \end{array} \tag{4.3}$$

$$holds(value(W,0),T{+}D) \;\text{:-}\; \begin{array}{l} type\_of(G, andg), \\ delay(G,D), \\ output(W,G), \\ contains\_input(G,0,T). \end{array} \tag{4.4}$$

$$holds(value(W,u),T{+}D) \;\text{:-}\; \begin{array}{l} type\_of(G, andg), \\ delay(G,D), \\ output(W,G), \\ not\ contains\_input(G,0,T), \\ contains\_input(G,u,T). \end{array} \tag{4.5}$$

(c) OR gate

Analogously, given an OR gate G with (propagation) delay D and output wire W, rule (4.6) expresses that if signal 0 is present (or holds) on all input wires of G at time T, then signal 0 will hold at W at time T+D. Rule (4.7) says that if signal 1 holds on at least one of the input wires of G, then signal 1 will hold at W at time $T+D$. Rule (4.8) expresses that if signal 1 is not present at any of the input wires of G, but signal u holds on at least one of G's input wires, then signal u will also hold at W at time $T{+}D$.

$$holds(value(W,0),T{+}D) \quad \text{:-} \quad \begin{array}{l} type\_of(G,org), \\ delay(G,D), \\ output(W,G), \\ all\_inputs(G,0,T). \end{array} \tag{4.6}$$

$$holds(value(W,1),T{+}D) \quad \text{:-} \quad \begin{array}{l} type\_of(G,org), \\ delay(G,D), \\ output(W,G), \\ contains\_input(G,1,T). \end{array} \tag{4.7}$$

$$holds(value(W,u),T{+}D) \quad \text{:-} \quad \begin{array}{l} type\_of(G,org), \\ delay(G,D), \\ output(W,G), \\ not\ contains\_input(G,1,T), \\ contains\_input(G,u,T). \end{array} \tag{4.8}$$

3. Rules $(4.9 - 4.11)$ are auxiliary relations used in the definition of the static laws for gates of type AND, or OR. Rule (4.9) expresses that if a signal S is present at one of the input wires of a gate G, of a type other than NOT, at time T, then we can infer that a different signal S' is not present at all input wires of G at T. When rule (4.10) fails to prove that a signal S is not present at all input wires of G, of a type other than NOT, at time T, then it deduces that signal S holds on all input wires of G at T. Rule (4.11) says that if signal S holds on input

wire W of a gate G, of a type other than NOT, at time T, then G contains at least one input which holds signal S at T.

$$\begin{aligned} not\_all\_inputs(G, S', T) \;\; \text{:-} \;\; & type\_of(G, Y), \\ & Y \neq notg, \\ & input(W, G), \\ & S \neq S', \\ & holds(value(W, S), T). \end{aligned} \tag{4.9}$$

$$\begin{aligned} all\_inputs(G, S, T) \;\; \text{:-} \;\; & type\_of(G, Y), \\ & Y \neq notg, \\ & not\ not\_all\_inputs(G, S, T). \end{aligned} \tag{4.10}$$

$$\begin{aligned} contains\_input(G, S, T) \;\; \text{:-} \;\; & type\_of(G, Y), \\ & Y \neq notg, \\ & input(W, G), \\ & holds(value(W, S), T). \end{aligned} \tag{4.11}$$

4. Rule (4.12) represents the *law of inertia* which states that "*normally, things tend to stay as they are.*" Rule (4.13) is a static causal law used in conjunction with the law of inertia, which determines that each single wire can only hold a distint signal value at each point in time.

$$\begin{aligned} holds(value(W, S), T+1) \;\; \text{:-} \;\; & holds(value(W, S), T), \\ & not\ \neg holds(value(W, S), T+1). \end{aligned} \tag{4.12}$$

$$\neg holds(value(W,S'),T) \quad :- \quad S \neq S', \qquad (4.13)$$
$$holds(value(W,S),T).$$

5. Rule (4.14) expresses the assumption that the signals present on a circuit in the *initial situation*, or initial moment of time 0, are unknown.

$$holds(value(W,u),0). \qquad (4.14)$$

Let $\mathcal{D}(C,\mathcal{I})$ be a consistent domain description and CT_0 be the simple circuit theory. The A-Prolog program representing circuit C is:

$$P_0 = CT_0 \ \cup \ \mathcal{D}(C,\mathcal{I}) = CT_0 \ \cup \ \pi(C) \ \cup \ \mathcal{O}(\mathcal{I}).$$ [1]

If C is a circuit consisting of a single NOT gate g with input wire w_1, output wire w_o, and no delays, then its description in A-Prolog, denoted by $\pi(C_{\text{NOT}})$, consists of the following statements:

$$\pi(C_{\text{NOT}}) = \begin{cases} type\_of(g, notg). \\ delay(g,0). \\ input(w_1,g). \\ output(w_o,g). \\ opposite(0,1). \\ opposite(1,0). \\ opposite(u,u). \end{cases}$$

[1]For simplicity, we will drop the parameters when writting $\mathcal{D}$ and $\mathcal{O}$.

If C is a circuit consisting of a single AND gate g with input wires $w_1, \ldots, w_n$, output wire w_o, and no delays, then its description in A-Prolog, denoted by $\pi(C_{\text{AND}})$, consists of the following statements:

$$\pi(C_{\text{AND}}) = \begin{cases} \mathit{type\_of}(g, \mathit{andg}). \\ \mathit{delay}(g, 0). \\ \mathit{input}(w_1, g). \\ \vdots \\ \mathit{input}(w_n, g). \\ \mathit{output}(w_o, g). \end{cases}$$

If C is a circuit consisting of a single OR gate g with input wires $w_1, \ldots, w_n$, output wire w_o, and no delays, then its description in A-Prolog, denoted by $\pi(C_{\text{OR}})$, consists of the following statements:

$$\pi(C_{\text{OR}}) = \begin{cases} \mathit{type\_of}(g, \mathit{org}). \\ \mathit{delay}(g, 0). \\ \mathit{input}(w_1, g). \\ \vdots \\ \mathit{input}(w_n, g). \\ \mathit{output}(w_o, g). \end{cases}$$

4.2 Proof of Lemma 4.1 - NOT gate

Lemma 4.1. *Let C be a combinational circuit consisting of a single* NOT *gate g with input wire w_1, output wire w_o, and no delays. Let s be an input signal vector of C and let $\mathcal{O} = \{\mathit{occurs}(\mathit{apply}(w_1, s), 0) \ : \ s \in \{0, 1\}\}$. Then*

1. *Program P_0 has a unique answer set; and*

2. *If $\mathcal{A}_0$ is the unique answer set of P_0 then*

$$\overline{s} = \text{NOT}(s) \textit{ if and only if } holds(value(w_o, \overline{s}), 1) \in \mathcal{A}_0.$$

□

Sketch of the proof - We construct a collection of programs P_0, P_1, P_2, such that

(i) P_{i-1} has a unique answer set if and only if P_i has a unique answer set,

(ii) If $\mathcal{A}_i$ is an answer set of P_i then s is an input signal of C if and only if $holds(value(w_o, \overline{s}), 1) \in \mathcal{A}_i$.

At each step the previous program will be substantially simplified. At the end we show that P_2 indeed has a unique answer set containing $holds(value(w_o, \overline{s}), 1)$.

Proof.

Step 1. Let U_0 be the set of literals formed by predicates *type_of*, *delay*, *input*, *output*, *opposite*, and *occurs*, over signature Σ. Let P_1 be the following program:

$$holds(value(w_1, v), 1). \qquad \text{from rule (4.1)} \tag{4.15}$$

$$\text{if } occurs(apply(w_1, v), 0) \in \mathcal{O} \text{ and } v \in \{0, 1\}$$

$$holds(value(w_o, \overline{s}), 0) \quad :\!- \quad holds(value(w_1, s), 0). \tag{4.16}$$

$$holds(value(w_o, \overline{s}), 1) \quad :\!- \quad holds(value(w_1, s), 1). \tag{4.17}$$

from rule (4.2)

if $s \in \{0, 1, u\}$ and

$\overline{s} = 0$ if $s = 1$,

$\overline{s} = 1$ if $s = 0$,

$\overline{s} = u$ if $s = u$

$$holds(value(w, s), 1) \quad :- \quad holds(value(w, s), 0), \quad not\ \neg holds(value(w, s), 1). \tag{4.18}$$

from rule (4.12)

if $w \in \{w_1, w_0\}$ and $s \in \{0, 1, u\}$

$$\neg holds(value(w, s'), 0) \quad :- \quad holds(value(w, s), 0). \tag{4.19}$$

$$\neg holds(value(w, s'), 1) \quad :- \quad holds(value(w, s), 1). \tag{4.20}$$

from rule (4.13)

if $w \in \{w_1, w_0\}$, $s \neq s'$ and $s, s' \in \{0, 1, u\}$

$$holds(value(w, u), 0). \tag{4.21}$$

from rule (4.14)

if $w \in \{w_1, w_0\}$

To show that P_0 and P_1 satisfy conditions (i)-(ii) notice that set U_0 splits program P_0. The bottom program, $b_{U_0}(P_0) = \pi(C) \cup \mathcal{O}$, consists only of facts. Hence, it has the unique answer set

$$\mathcal{A}_{b_0} = \pi(C) \ \cup \ \mathcal{O}.$$

It is easy to see that P_1 is the partial evaluation of the top $t_{U_0}(P_0)$ with respect to U_0 and $\mathcal{A}_{b_0}$, i.e.,

$$P_1 = e_{U_0}(t_{U_0}(P_0), \mathcal{A}_{b_0}).$$

By the Splitting Set Theorem, $\mathcal{A}_0$ is an answer set of P_0 if and only if $\mathcal{A}_0 = \mathcal{A}_{b_0} \cup \mathcal{A}_1$, where $\mathcal{A}_1$ is an answer set of P_1. Then,

(a) P_0 has a unique answer set if and only if P_1 does;

(b) $\mathcal{A}_{b_0}$ is a set of atoms (not containing predicate *holds*), hence $holds(value(w_o, s), 1) \in \mathcal{A}_0$ if and only if $holds(value(w_o, s), 1) \in \mathcal{A}_1$.

Step 2. Program P_2 will be constructed in two steps. First, let U_1 be the set of literals of P_1 whose time parameter is 0, i.e.,

$$U_1 = \{holds(value(w, s), 0), \neg holds(value(w, s), 0)\},$$

where $w \in \{w_1, w_o\}$, and $s \in \{0, 1, u\}$. Set U_1 splits P_1 into bottom $b_{U_1}(P_1)$ and top $t_{U_1}(P_1)$. Let $Q_1 = b_{U_1}(P_1)$ be program

$$holds(value(w_o, \overline{s}), 0) \quad \text{:-} \quad holds(value(w_1, s), 0). \tag{4.22}$$

from rule (4.16)

if $s \in \{0, 1, u\}$ and

$\overline{s} = 0$ if $s = 1$,

$\overline{s} = 1$ if $s = 0$,

$\overline{s} = u$ if $s = u$

$$\neg holds(value(w, s'), 0) \quad :- \quad holds(value(w, s), 0). \tag{4.23}$$

from rule (4.19)

if $w \in \{w_1, w_0\}$, $s \neq s'$ and $s, s' \in \{0, 1, u\}$

$$holds(value(w, u), 0). \tag{4.24}$$

from rule (4.21)

if $w \in \{w_1, w_0\}$

It is easy to see that program Q_1 has the unique answer set,

$$\begin{aligned} \mathcal{A}_{b_1} \quad = \quad & \{holds(value(w_1, u), 0), \neg holds(value(w_1, 1), 0), \neg holds(value(w_1, 0), 0) \\ & holds(value(w_o, u), 0), \neg holds(value(w_o, 1), 0), \neg holds(value(w_o, 0), 0)\}. \end{aligned}$$

Now let P_2 be program

$$holds(value(w_1, v), 1). \tag{4.25}$$

from rule (4.15)

if $occurs(apply(w_1, v), 0) \in \mathcal{O}$ and $v \in \{0, 1\}$

$$holds(value(w_o, \overline{s}), 1) \quad :- \quad holds(value(w_1, s), 1). \tag{4.26}$$

from rule (4.17)

if $s \in \{0, 1, u\}$ and

$$\overline{s} = 0 \text{ if } s = 1,$$

$$\overline{s} = 1 \text{ if } s = 0,$$

$$\overline{s} = u \text{ if } s = u$$

$$holds(value(w,u),1) \quad \text{:-} \quad not\ \neg holds(value(w,u),1). \tag{4.27}$$

from rule (4.18)

if $w \in \{w_1, w_0\}$ and $s \in \{0,1,u\}$

$$\neg holds(value(w,s'),1) \quad \text{:-} \quad holds(value(w,s),1). \tag{4.28}$$

from rule (4.20)

if $w \in \{w_1, w_0\}$, $s \neq s'$ and $s, s' \in \{0,1,u\}$

It is easy to see that P_2 is the partial evaluation of the top $t_{U_1}(P_1)$ with respect to U_1 and $\mathcal{A}_2$, i.e.,

$$P_2 = e_{U_1}(t_{U_1}(P_1), \mathcal{A}_{b_1}).$$

By the Splitting Set Theorem, $\mathcal{A}_1$ is an answer set of P_1 if and only if $\mathcal{A}_1 = \mathcal{A}_{b_1} \cup \mathcal{A}_2$, where $\mathcal{A}_2$ is an answer set of P_2. Then,

(a) P_1 has a unique answer set if and only if P_2 does;

(b) $\mathcal{A}_{b_1}$ is a set of literals formed by predicate *holds* for $t = 0$ only, hence $holds(value(w_o, s), 1) \in \mathcal{A}_1$ if and only if $holds(value(w_o, s), 1) \in \mathcal{A}_2$.

Now we will show that

(iii) Program P_2 has a unique answer set, A_2, and v is the input of C if and only if $holds(value(w_o, \overline{v}), 1) \in \mathcal{A}_2$.

There are two cases to consider.

Case 1. v is a *definite* input signal of C. By definition of $\mathcal{O}$, it follows that input v is definite if and only if $occurs(apply(w_1, v), 0) \in \mathcal{O}$.

Let U_2 be the set of positive literals of the form $holds(value(w, 0), 1)$ and $holds(value(w, 1), 1)$, where $w \in \{w_1, w_o\}$. Set U_2 splits P_2 into bottom $b_{U_2}(P_2)$ and top $t_{U_2}(P_2)$.

The bottom consists of rules:

$holds(value(w_1, v), 1).$ from rule (4.25)

if $occurs(apply(w_1, v), 0) \in \mathcal{O}$ and $v \in \{0, 1\}$

$$holds(value(w_o, 0), 1) \quad \text{:-} \quad holds(value(w_1, 1), 1).$$
$$holds(value(w_o, 1), 1) \quad \text{:-} \quad holds(value(w_1, 0), 1).$$

from rule (4.26)

It is easy to see that

$$\mathcal{A}_{b_2} = \{holds(value(w_1, v), 1), holds(value(w_o, \overline{v}), 1)\}$$

is an answer set of $b_{U_2}(P_2)$. Since $b_{U_2}(P_2)$ is a definite program, this answer set is unique.

The top consists of rules:

$$holds(value(w_o, u), 1) \quad :\!- \quad holds(value(w_1, u), 1). \tag{4.29}$$

from rule (4.26)

$$holds(value(w, u), 1) \quad :\!- \quad not\ \neg holds(value(w, u), 1). \tag{4.30}$$

from rule (4.27)

if $w \in \{w_1, w_0\}$

$$\neg holds(value(w, s'), 1) \quad :\!- \quad holds(value(w, s), 1). \tag{4.31}$$

from rule (4.28)

if $w \in \{w_1, w_0\}$, $s \neq s'$ and $s, s' \in \{0, 1, u\}$

The partial evaluation of the top $t_{U_2}(P_2)$ with respect to U_2 and $\mathcal{A}_{b_2}$, i.e., $P_3^I = e_{U_2}(t_{U_2}(P_2), \mathcal{A}_{b_2})$, consists of rules:

$$holds(value(w_o, u), 1) \quad :\!- \quad holds(value(w_1, u), 1).$$

from rule (4.29)

$$holds(value(w,u),1) \quad :- \quad not\ \neg holds(value(w,u),1).$$

from rule (4.30)

$$\neg holds(value(w_1,\overline{v}),1).$$ from rule (4.31)

$$\neg holds(value(w_1,u),1).$$ if $v \in \{0,1\}$ and

$$\neg holds(value(w_o,v),1).$$ $\overline{v}$ is the dual signal of v

$$\neg holds(value(w_o,u),1).$$

In order to prove that P_2 has a unique answer set, we need to show that P_3^I also does.

Let U_3 be the set of all negative literals formed by predicate *holds*. Set U_3 splits P_3^I into bottom $b_{U_3}(P_3^I)$ and top $t_{U_3}(P_3^I)$. The bottom consists of facts of P_3^I, and has unique answer set

$$\begin{aligned}\mathcal{A}_{b_3} &= \{\neg holds(value(w_1,\overline{v}),1), \neg holds(value(w_1,u),1),\\ &\quad \neg holds(value(w_o,v),1), \neg holds(value(w_o,u),1)\}.\end{aligned}$$

The partial evaluation of the top $t_{U_3}(P_3^I)$ with respect to U_3 and $\mathcal{A}_{b_3}$, i.e., $P_4 = e_{U_3}(t_{U_3}(P_3^I), \mathcal{A}_{b_3})$ consists of a single rule:

$$holds(value(w_o,u),1) \quad :- \quad holds(value(w_1,u),1).$$

and has the unique answer set:

$$\mathcal{A}_4 = \{\}.$$

Hence, by the Splitting Set Theorem, we can conclude that if v is a definite input signal of C, then P_2 has unique answer set $\mathcal{A}_2^I = \mathcal{A}_{b_2} \cup \mathcal{A}_{b_3} \cup \mathcal{A}_4$, i.e.,

$$\begin{aligned}\mathcal{A}_2^I \;=\; & \{holds(value(w_1, v), 1), \neg holds(value(w_1, \overline{v}), 1), \neg holds(value(w_1, u), 1), \\ & holds(value(w_o, \overline{v}), 1), \neg holds(value(w_o, v), 1), \neg holds(value(w_o, u), 1)\}\end{aligned}$$

which implies that program P_2 has a unique answer set, namely $\mathcal{A}_2^I$, and that for every input signal $v \in \{0, 1\}$ of C, $holds(value(w_o, \overline{v}), 1) \in \mathcal{A}_2^I$.

Case 2. $v = u$. By definition of $\mathcal{O}$, $v = u$ if and only if $\mathcal{O} = \emptyset$.

Let U_4 be the set of literals of the form

1. $\neg holds(value(w, u), 1)$,
2. $holds(value(w, 0), 1)$,
3. $holds(value(w, 1), 1)$,

where $w \in \{w_1, w_o\}$. Set U_4 splits P_2 into bottom $b_{U_4}(P_2)$ and top $t_{U_4}(P_2)$.

The bottom consists of rules:

from (4.26): $holds(value(w_o, 0), 1)$:- $holds(value(w_1, 1), 1)$.

$holds(value(w_o, 1), 1)$:- $holds(value(w_1, 0), 1)$.

from (4.28): if $w \in \{w_1, w_0\}$

$$\neg holds(value(w,u),1) \text{ :- } \quad holds(value(w,0),1).$$

$$\neg holds(value(w,u),1) \text{ :- } \quad holds(value(w,1),1).$$

It is easy to see that $b_{U_4}(P_2)$ has the unique answer set

$$\mathcal{A}_{b_4} = \{\}.$$

The top $t_{U_4}(P_2)$ consists of rules

from (4.26): $$holds(value(w_o,u),1) \text{ :- } \quad holds(value(w_1,u),1). \tag{4.32}$$

from (4.27): if $w \in \{w_1, w_0\}$

$$holds(value(w,u),1) \text{ :- } not\ \neg holds(value(w,u),1). \tag{4.33}$$

from (4.28): if $w \in \{w_1, w_0\}$

$$\neg holds(value(w,0),1) \text{ :- } \quad holds(value(w,1),1). \tag{4.34}$$

$$\neg holds(value(w,0),1) \text{ :- } \; holds(value(w,u),1). \tag{4.35}$$

$$\neg holds(value(w,1),1) \text{ :- } \; holds(value(w,0),1). \tag{4.36}$$

$$\neg holds(value(w,1),1) \text{ :- } \; holds(value(w,u),1). \tag{4.37}$$

The partial evaluation of the top $t_{U_4}(P_2)$ with respect to U_4 and $\mathcal{A}_{b_4}$, i.e., $P_3^{II} = e_{U_4}(t_{U_4}(P_2), \mathcal{A}_{b_4})$ consists of rules

$$\text{from (4.32):} \quad holds(value(w_o,u),1) \text{ :- } \; holds(value(w_1,u),1). \tag{4.38}$$

from (4.33): if $w \in \{w_1, w_0\}$

$$holds(value(w,u),1). \tag{4.39}$$

from (4.35): if $w \in \{w_1, w_0\}$

$$\neg holds(value(w,0),1) \text{ :- } \; holds(value(w,u),1). \tag{4.40}$$

from (4.37): if $w \in \{w_1, w_0\}$

$$\neg holds(value(w,1),1) \text{ :- } \; holds(value(w,u),1). \tag{4.41}$$

Now, to prove that P_2 has a unique answer set, it is enough to show that P_3^{II} also does.

Let U_5 be the set of positive literals of the form $holds(value(w,u),1)$, where $w \in \{w_1, w_o\}$. Set U_5 splits P_3^{II} into bottom $b_{U_5}(P_3^{II})$ and top $t_{U_5}(P_3^{II})$.

The bottom consists of the following rules:

from (4.38): $holds(value(w_o,u),1)$:- $holds(value(w_1,u),1)$.

from (4.39): $holds(value(w_1,u),1)$.

$holds(value(w_o,u),1)$.

It is easy to see that

$$\mathcal{A}_{b_5} = \{holds(value(w_1,u),1), holds(value(w_o,u),1)\}$$

is an answer set of $b_{U_5}(P_3^{II})$. Since $b_{U_5}(P_3^{II})$ is a definite program, this answer set is unique.

The top $t_{U_5}(P_3^{II})$ consists of the following rules:

from (4.40): if $w \in \{w_1, w_0\}$

$\neg holds(value(w,0),1)$:- $holds(value(w,u),1)$.

from (4.41): if $w \in \{w_1, w_0\}$

$\neg holds(value(w,1),1)$:- $holds(value(w,u),1)$.

The partial evaluation of the top $t_{U_5}(P_3^{II})$ with respect to U_5 and $\mathcal{A}_{b_5}$, i.e., $P_5 = e_{U_5}(t_{U_5}(P_3^{II}), \mathcal{A}_{b_5})$ consists of atoms, and therefore, has the unique answer set:

$$\begin{aligned}\mathcal{A}_5 &= \{\neg holds(value(w_1, 0), 1), \neg holds(value(w_1, 1), 1), \\ &\quad \neg holds(value(w_o, 0), 1), \neg holds(value(w_o, 1), 1)\}.\end{aligned}$$

Hence, by the Splitting Set Theorem, we can conclude that if $v = u$, then P_2 has unique answer set $\mathcal{A}_2^{II} = \mathcal{A}_{b_4} \cup \mathcal{A}_{b_5} \cup \mathcal{A}_5$, i.e.,

$$\begin{aligned}\mathcal{A}_2^{II} &= \{holds(value(w_1, u), 1), \neg holds(value(w_1, 0), 1), \neg holds(value(w_1, 1), 1), \\ &\quad holds(value(w_o, u), 1), \neg holds(value(w_o, 0), 1), \neg holds(value(w_o, 1), 1)\}.\end{aligned}$$

which implies that program P_2 has a unique answer set, namely $\mathcal{A}_2^{II}$, and that for every input signal u of C, $holds(value(w_o, u), 1) \in \mathcal{A}_2^{II}$.

which concludes the proof of (iii).

The Lemma follows immediately from (i),(ii), and (iii). □

4.3 Proof of Lemma 4.2 - AND gate

Lemma 4.2. *Let C be a combinational circuit consisting of a single* AND *gate g with input wires $w_1, \ldots, w_n$, output wire w_o, and no delays. Let $v_1, \ldots, v_n$ be an input signal vector of C and let $\mathcal{O} = \{occurs(apply(w_i, v_i), 0) \ : \ w_i \in \{w_1, \ldots, w_n\}, v_i \in \{0, 1\}$, for $1 \leq i \leq k, k \leq n\}$. Then*

1. *Program P_0 has unique answer set; and*

2. If $\mathcal{A}_0$ is the unique answer set of P_0 then

$$v = \text{AND}(v_1, \ldots, v_n) \;\; \textit{if and only if} \;\; holds(value(w_o, v), 1) \in \mathcal{A}_0.$$

$\square$

Sketch of the proof - This proof follows the same scheme as the proof for Lemma 4.1.

Proof.

Step 1. Let U_0 be the set of literals formed by predicates *type_of*, *delay*, *input*, *output*, and *occurs* over signature Σ. Let P_1 be program

$$holds(value(w_i, v_i), 1). \quad \text{from rule (4.1) if } occurs(apply(w_i, v_i), 0) \in \mathcal{O}, \; w_i \in \{w_1, \ldots, w_n\}, \text{ and } v_i \in \{0, 1\} \tag{4.42}$$

$$holds(value(w_o, 1), t) \;\; \text{:-} \;\; all\_inputs(g, 1, t). \tag{4.43}$$

from rule (4.3)

if $t \in \{0, 1\}$

$$holds(value(w_o, 0), t) \;\; \text{:-} \;\; contains\_input(g, 0, t). \tag{4.44}$$

from rule (4.4)

if $t \in \{0, 1\}$

$$holds(value(w_o, u), t) \;\; \text{:-} \;\; not \; contains\_input(g, 0, t), \tag{4.45}$$

$$contains\_input(g, u, t).$$

from rule (4.5)

if $t \in \{0, 1\}$

$$not\_all\_inputs(g, s_j, t) \quad \text{:-} \quad holds(value(w_i, s_i), t). \tag{4.46}$$

from rule (4.9)

if $w \in \{w_1, \ldots, w_n\}$,

$s_i, s_j \in \{0, 1, u\}, s_i \neq s_j$

$1 \leq i, j \leq n,$ and $t \in \{0, 1\}$

$$all\_inputs(g, s, t) \quad \text{:-} \quad not\ not\_all\_inputs(g, s, t). \tag{4.47}$$

from rule (4.10)

if $s \in \{0, 1, u\}$ and $t \in \{0, 1\}$

$$contains\_input(g, s_i, t) \quad \text{:-} \quad holds(value(w_i, s_i), t). \tag{4.48}$$

from rule (4.11)

if $w_i \in \{w_1, \ldots, w_n\}$,

$s_i \in \{0, 1, u\}, 1 \leq i, j \leq n,$ and $t \in \{0, 1\}$

$$holds(value(w, s), 1) \quad \text{:-} \quad holds(value(w, s), 0), \tag{4.49}$$

$$not\ \neg holds(value(w,s),1).$$

from rule (4.12)

if $w \in \{w_1,\dots,w_n,w_o\}$, and $s \in \{0,1,u\}$

$$\neg holds(value(w,s'),t) \quad \text{:-} \quad holds(value(w,s),t). \tag{4.50}$$

from rule (4.13)

if $w \in \{w_1,\dots,w_n,w_o\}$,

$s \neq s'$ and $s,s' \in \{0,1,u\}$

$$holds(value(w,u),0). \qquad \text{from rule (4.14)} \tag{4.51}$$

if $w \in \{w_1,\dots,w_n,w_o\}$

Set U_0 splits program P_0 into two parts: bottom, $b_{U_0}(P_0) = \pi(C)\ \cup\ \mathcal{O}$, and top, $t_{U_0}(P_0) = P_0 \setminus (\pi(C)\ \cup\ \mathcal{O})$. The bottom has the unique answer set

$$\mathcal{A}_{b_0} = \pi(C)\ \cup\ \mathcal{O}.$$

It is easy to see that P_1 is the partial evaluation of the top $t_{U_0}(P_0)$ with respect to U_0 and $\mathcal{A}_{b_0}$, i.e.,

$$P_1 = e_{U_0}(t_{U_0}(P_0), \mathcal{A}_{b_0}).$$

By the Splitting Set Theorem, $\mathcal{A}_0$ is an answer set of P_0 if and only if $\mathcal{A}_0 = \mathcal{A}_{b_0} \cup \mathcal{A}_1$, where $\mathcal{A}_1$ is an answer set of P_1. Then

(a) P_0 has unique answer set if and only if P_1 does; and

(b) $holds(value(w_o, s), 1) \in \mathcal{A}_0$ if and only if $holds(value(w_o, s), 1) \in \mathcal{A}_1$.

Step 2. Program P_2 will be constructed in two steps. First, let U_1 be the set of literals of P_1 whose time parameter is 0, i.e., all literals of the form

1. $holds(value(w, s), 0)$,
2. $\neg holds(value(w, s), 0)$,
3. $all\_inputs(g, s, 0)$,
4. $contains\_input(g, s, 0)$,
5. $not\_all\_inputs(g, s, 0)$,

where $w \in \{w_1, \ldots, w_n\}$, and $s \in \{0, 1, u\}$.

Set U_1 splits P_1 into bottom $b_{U_1}(P_1)$ and top $t_{U_1}(P_1)$. Program $b_{U_1}(P_1)$ consists of rules

$$holds(value(w_o, 1), 0) \quad :- \quad all\_inputs(g, 1, 0).$$

$$holds(value(w_o, 0), 0) \quad :- \quad contains\_input(g, 0, 0).$$

$$\begin{aligned} holds(value(w_o, u), 0) \quad :- \quad & not\ contains\_input(g, 0, 0), \\ & contains\_input(g, u, 0). \end{aligned}$$

$$not\_all\_inputs(g, s_j, 0) \quad \text{:-} \quad holds(value(w_i, s_i), 0). \qquad : 1 \leq i \leq n,\ s_i \neq s_j$$

$$all\_inputs(g, s, 0) \quad \text{:-} \quad not\ not\_all\_inputs(g, s, 0).$$

$$contains\_input(g, s_i, 0) \quad \text{:-} \quad holds(value(w_i, s_i), 0). \qquad : 1 \leq i \leq n$$

$$\neg holds(value(w, s'), 0) \quad \text{:-} \quad holds(value(w, s), 0). \qquad : s \neq s'$$

$$holds(value(w, u), 0).$$

We need to show that program $Q_1 = b_{U_1}(P_1)$ has a unique answer set. For that, let N_0 be the set of literals of the form

1. $holds(value(w_o, 0), 0)$,
2. $contains\_input(g, s, 0)$,
3. $not\_all\_inputs(g, s, 0)$,
4. $holds(value(w_i, s), 0)$,
5. $\neg holds(value(w_i, s), 0)$,

where $w_i \in \{w_1, \ldots, w_n\}$ and $s \in \{0, 1, u\}$. Set N_0 splits Q_1. The bottom $b_{N_0}(Q_1)$ consists of rules

$$holds(value(w_o, 0), 0) \quad \text{:-} \quad contains\_input(g, 0, 0).$$

$$
\begin{array}{rcl}
not\_all\_inputs(g,0,0) & \text{:-} & holds(value(w_i,1),0). \\
not\_all\_inputs(g,0,0) & \text{:-} & holds(value(w_i,u),0). \\
not\_all\_inputs(g,1,0) & \text{:-} & holds(value(w_i,0),0). \\
not\_all\_inputs(g,1,0) & \text{:-} & holds(value(w_i,u),0). \\
not\_all\_inputs(g,u,0) & \text{:-} & holds(value(w_i,0),0). \\
not\_all\_inputs(g,u,0) & \text{:-} & holds(value(w_i,1),0).
\end{array}
$$

$$
\begin{array}{rcl}
contains\_input(g,0,0) & \text{:-} & holds(value(w_i,0),0). \\
contains\_input(g,1,0) & \text{:-} & holds(value(w_i,1),0). \\
contains\_input(g,u,0) & \text{:-} & holds(value(w_i,u),0).
\end{array}
$$

$$
\begin{array}{rcl}
\neg holds(value(w_i,0),0) & \text{:-} & holds(value(w_i,1),0). \\
\neg holds(value(w_i,0),0) & \text{:-} & holds(value(w_i,u),0). \\
\neg holds(value(w_i,1),0) & \text{:-} & holds(value(w_i,0),0). \\
\neg holds(value(w_i,1),0) & \text{:-} & holds(value(w_i,u),0). \\
\neg holds(value(w_i,u),0) & \text{:-} & holds(value(w_i,0),0). \\
\neg holds(value(w_i,u),0) & \text{:-} & holds(value(w_i,1),0).
\end{array}
$$

$$holds(value(w_i,u),0).$$

It is easy to see that bottom $b_{N_0}(Q_1)$ has the unique answer set

$$
\begin{array}{rcl}
\mathcal{A}_{b_{N_0}} & = & \{holds(value(w_i,u),0), \neg holds(value(w_i,0),0), \neg holds(value(w_i,1),0), \\
& & not\_all\_inputs(g,0,0), not\_all\_inputs(g,1,0), contains\_input(g,u,0)\}.
\end{array}
$$

The top $t_{N_0}(Q_1)$ consists of rules

$$holds(value(w_o, 1), 0) \quad :- \quad all\_inputs(g, 1, 0).$$

$$\begin{aligned} holds(value(w_o, u), 0) \quad :- \quad & not\ contains\_input(g, 0, 0), \\ & contains\_input(g, u, 0). \end{aligned}$$

$$\begin{aligned} all\_inputs(g, 0, 0) \quad &:- \quad not\ not\_all\_inputs(g, 0, 0). \\ all\_inputs(g, 1, 0) \quad &:- \quad not\ not\_all\_inputs(g, 1, 0). \\ all\_inputs(g, u, 0) \quad &:- \quad not\ not\_all\_inputs(g, u, 0). \end{aligned}$$

$$\begin{aligned} \neg holds(value(w_o, 0), 0) \quad &:- \quad holds(value(w_o, 1), 0). \\ \neg holds(value(w_o, 0), 0) \quad &:- \quad holds(value(w_o, u), 0). \\ \neg holds(value(w_o, 1), 0) \quad &:- \quad holds(value(w_o, 0), 0). \\ \neg holds(value(w_o, 1), 0) \quad &:- \quad holds(value(w_o, u), 0). \\ \neg holds(value(w_o, u), 0) \quad &:- \quad holds(value(w_o, 1), 0). \\ \neg holds(value(w_o, u), 0) \quad &:- \quad holds(value(w_o, 0), 0). \end{aligned}$$

$$holds(value(w_o, u), 0).$$

The partial evaluation of the top $t_{N_0}(Q_1)$ with respect to N_0 and $\mathcal{A}_{b_{N_0}}$, i.e.,

$$Q_2 = e_{N_0}(t_{N_0}(Q_1), \mathcal{A}_{b_{N_0}}),$$

consists of rules

$$holds(value(w_o,1),0) \quad :- \quad all\_inputs(g,1,0).$$

$$holds(value(w_o,u),0).$$

$$all\_inputs(g,u,0).$$

$$\begin{aligned}
\neg holds(value(w_o,0),0) &\quad :- \quad holds(value(w_o,u),0).\\
\neg holds(value(w_o,1),0) &\quad :- \quad holds(value(w_o,u),0).\\
\neg holds(value(w_o,0),0) &\quad :- \quad holds(value(w_o,1),0).\\
\neg holds(value(w_o,u),0) &\quad :- \quad holds(value(w_o,1),0).
\end{aligned}$$

$$holds(value(w_o,u),0).$$

It is easy to see that Q_2 has the unique answer set

$$\begin{aligned}
\mathcal{A}_{Q_2} &= \{all_inputs(g,u,0), holds(value(w_o,u),0),\\
&\quad \neg holds(value(w_o,0),0), \neg holds(value(w_o,1),0)\}
\end{aligned}$$

By the Splitting Set Theorem, we have that $Q_1 = b_{U_1}(P_1)$ has unique answer set $\mathcal{A}_{b_1} = \mathcal{A}_{b_{N_0}} \cup \mathcal{A}_{Q_2}$, i.e.,

$$\begin{aligned}
\mathcal{A}_{b_1} &= \{holds(value(w_i,u),0), \neg holds(value(w_i,0),0), \neg holds(value(w_i,1),0),\\
&\quad not_all_inputs(g,0,0), not_all_inputs(g,1,0),
\end{aligned}$$

$all\_inputs(g,u,0), contains\_input(g,u,0),$

$holds(value(w_o,u),0), \neg holds(value(w_o,0),0), \neg holds(value(w_o,1),0)\}.$

Second, let P_2 be program

$$holds(value(w_i,v_i),1). \qquad \text{from rule (4.1) if } occurs(apply(w_i,v_i),0) \in \mathcal{O}, \; w_i \in \{w_1,\ldots,w_n\}, \text{ and } v_i \in \{0,1\} \tag{4.52}$$

$$holds(value(w_o,1),1) \quad :- \quad all\_inputs(g,1,1). \tag{4.53}$$

$$holds(value(w_o,0),1) \quad :- \quad contains\_input(g,0,1). \tag{4.54}$$

$$holds(value(w_o,u),1) \quad :- \quad not\ contains\_input(g,0,1), \; contains\_input(g,u,1). \tag{4.55}$$

$$not\_all\_inputs(g,s_j,1) \quad :- \quad holds(value(w_i,s_i),1). \qquad 1 \le i \le n, \; s_i \ne s_j \tag{4.56}$$

$$all\_inputs(g,s,1) \quad :- \quad not\ not\_all\_inputs(g,s,1). \tag{4.57}$$

$$contains\_input(g,s_i,1) \quad :- \quad holds(value(w_i,s_i),1). \qquad 1 \le i \le n \tag{4.58}$$

$$holds(value(w,u),1) \quad :- \quad not\ \neg holds(value(w,u),1). \tag{4.59}$$

$$\neg holds(value(w, s'), 1) \quad :- \quad holds(value(w, s), 1). \tag{4.60}$$
$$s \neq s'$$

It is easy to see that P_2 is the partial evaluation of the top $t_{U_1}(P_1)$ with respect to U_1 and $\mathcal{A}_{b_1}$, i.e.,

$$P_2 = e_{U_1}(t_{U_1}(P_1), \mathcal{A}_{b_1}).$$

By the Splitting Set Theorem, $\mathcal{A}_1$ is an answer set of P_1 if and only if $\mathcal{A}_1 = \mathcal{A}_{b_1} \cup \mathcal{A}_2$, where $\mathcal{A}_2$ is an answer set of P_2. Then

(a) P_1 has unique answer set if and only if P_2 does; and

(b) $holds(value(w_o, s), 1) \in \mathcal{A}_1$ if and only if $holds(value(w_o, s), 1) \in \mathcal{A}_2$.

Step 3. Now we need to show that

1. Program P_2 has a unique answer set, $\mathcal{A}_2$;
2. For every input signal vector $\mathcal{I} = \{v_1, \dots, v_n\}$ of C,

$$v = \text{AND}(v_1, \dots, v_n) \text{ if and only if } holds(value(w_o, v), 1) \in \mathcal{A}_2.$$

There are three cases to consider:

1. $\forall v_i \in \mathcal{I}$, $v_i = 1$;
2. $\exists v_k \in \mathcal{I}$ such that $v_k = 0$; and

3. $\forall v_i \in \mathcal{I}$, $v_i \neq 0$, and $\exists v_k \in \mathcal{I}$ such that $v_k = u$.

Case 1. For every $v_i \in \mathcal{I}$, $v_i = 1$, which implies that $occurs(apply(w_i, 1), 0) \in \mathcal{O}$.

Let H_0 be the set of literals of the form

1. $holds(value(w_i, 1), 1)$,
2. $holds(value(w_i, 0), 1)$,
3. $not\_all\_inputs(g, u, 1)$,
4. $contains\_input(g, 0, 1)$,
5. $contains\_input(g, 1, 1)$,
6. $holds(value(w_o, 0), 1)$,
7. $\neg holds(value(w_i, u), 1)$,

where $w_i \in \{w_1, \ldots, w_n\}$. Set H_0 splits P_2 into bottom $b_{H_0}(P_2)$ and top $t_{H_0}(P_2)$. The bottom consists of rules

$$holds(value(w_i, 1), 1).$$

$$holds(value(w_o, 0), 1) \quad :\!- \quad contains\_input(g, 0, 1).$$

$$\begin{aligned} not\_all\_inputs(g, u, 1) \quad &:\!- \quad holds(value(w_i, 0), 1). \\ not\_all\_inputs(g, u, 1) \quad &:\!- \quad holds(value(w_i, 1), 1). \end{aligned}$$

$$contains\_input(g, 0, 1) \quad :\!- \quad holds(value(w_i, 0), 1).$$

$$contains\_input(g,1,1) \quad :- \quad holds(value(w_i,1),1).$$

$$\neg holds(value(w_i,u),1) \quad :- \quad holds(value(w_i,0),1).$$
$$\neg holds(value(w_i,u),1) \quad :- \quad holds(value(w_i,1),1).$$

It is easy to see that $b_{H_0}(P_2)$ has the unique answer set

$$\begin{aligned} \mathcal{A}_{b_{H_0}} &= \{holds(value(w_i,1),1), not\_all\_inputs(g,u,1), \\ &\quad contains\_input(g,1,1), \neg holds(value(w_i,u),1)\}. \end{aligned}$$

Top $t_{H_0}(P_2)$ consists of rules

$$\begin{aligned}
holds(value(w_o,1),1) &\quad :- \quad all_inputs(g,1,1). \\
holds(value(w_o,u),1) &\quad :- \quad not\ contains_input(g,0,1), \\
&\qquad\quad contains_input(g,u,1). \\
not_all_inputs(g,0,1) &\quad :- \quad holds(value(w_i,1),1). \\
not_all_inputs(g,0,1) &\quad :- \quad holds(value(w_i,u),1). \\
not_all_inputs(g,1,1) &\quad :- \quad holds(value(w_i,0),1). \\
not_all_inputs(g,1,1) &\quad :- \quad holds(value(w_i,u),1). \\
all_inputs(g,1,1) &\quad :- \quad not\ not_all_inputs(g,1,1). \\
all_inputs(g,0,1) &\quad :- \quad not\ not_all_inputs(g,0,1). \\
all_inputs(g,u,1) &\quad :- \quad not\ not_all_inputs(g,u,1). \\
contains_input(g,u,1) &\quad :- \quad holds(value(w_i,u),1). \\
holds(value(w_i,u),1) &\quad :- \quad not\ \neg holds(value(w_i,u),1). \\
holds(value(w_o,u),1) &\quad :- \quad not\ \neg holds(value(w_o,u),1).
\end{aligned}$$

$$\begin{array}{rcll}
\neg holds(value(w_i,0),1) & :- & holds(value(w_i,1),1). & \\
\neg holds(value(w_i,0),1) & :- & holds(value(w_i,u),1). & \\
\neg holds(value(w_i,1),1) & :- & holds(value(w_i,0),1). & \\
\neg holds(value(w_i,1),1) & :- & holds(value(w_i,u),1). & \\
\neg holds(value(w_o,s'),1) & :- & holds(value(w_o,s),1). & : s \neq s'
\end{array}$$

The partial evaluation of the top $t_{H_0}(P_2)$ with respect to H_0 and $\mathcal{A}_{b_{H_0}}$, i.e. $P_3^I = e_{H_0}(t_{H_0}(P_2), \mathcal{A}_{b_{H_0}})$, consists of rules

$$\begin{array}{rcl}
holds(value(w_o,1),1) & :- & all_inputs(g,1,1). \\
holds(value(w_o,u),1) & :- & contains_input(g,u,1). \\
not_all_inputs(g,0,1). & & \\
not_all_inputs(g,0,1) & :- & holds(value(w_i,u),1). \\
not_all_inputs(g,1,1) & :- & holds(value(w_i,u),1). \\
all_inputs(g,1,1) & :- & not\ not_all_inputs(g,1,1). \\
all_inputs(g,0,1) & :- & not\ not_all_inputs(g,0,1). \\
contains_input(g,u,1) & :- & holds(value(w_i,u),1). \\
holds(value(w_o,u),1) & :- & not\ \neg holds(value(w_o,u),1). \\
\neg holds(value(w_i,0),1). & & \\
\neg holds(value(w_i,0),1) & :- & holds(value(w_i,u),1). \\
\neg holds(value(w_i,1),1) & :- & holds(value(w_i,u),1). \\
\neg holds(value(w_o,0),1) & :- & holds(value(w_o,1),1). \\
\neg holds(value(w_o,u),1) & :- & holds(value(w_o,1),1). \\
\neg holds(value(w_o,0),1) & :- & holds(value(w_o,u),1). \\
\neg holds(value(w_o,1),1) & :- & holds(value(w_o,u),1).
\end{array}$$

In order to prove that P_2 has a unique answer set, we need to show that P_3^I also does. Let H_1 be the set of atoms of the form

1. $holds(value(w_i, u), 1)$,
2. $not\_all\_inputs(g, 0, 1)$,
3. $not\_all\_inputs(g, 1, 1)$,
4. $contains\_input(g, u, 1)$,
5. $\neg holds(value(w_i, 1), 1)$,
6. $\neg holds(value(w_i, 0), 1)$,

where $w_i \in \{w_1, \ldots, w_n\}$. Set H_1 splits P_3^I. The bottom $b_{H_1}(P_3^I)$ consists of rules

$$\begin{array}{rcl} not\_all\_inputs(g,0,1). & & \\ not\_all\_inputs(g,0,1) & \text{:-} & holds(value(w_i,u),1). \\ not\_all\_inputs(g,1,1) & \text{:-} & holds(value(w_i,u),1). \\ contains\_input(g,u,1) & \text{:-} & holds(value(w_i,u),1). \\ \neg holds(value(w_i,0),1). & & \\ \neg holds(value(w_i,0),1) & \text{:-} & holds(value(w_i,u),1). \\ \neg holds(value(w_i,1),1) & \text{:-} & holds(value(w_i,u),1). \end{array}$$

It is easy to see that bottom $b_{H_1}(P_3^I)$ has the unique answer set

$$\mathcal{A}_{b_{H_1}} = \{not\_all\_inputs(g,0,1), \neg holds(value(w_i,0),1)\}.$$

Top $t_{H_1}(P_3^I)$ consists of rules

$$
\begin{array}{rcl}
holds(value(w_o, 1), 1) & \text{:-} & all\_inputs(g, 1, 1). \\
holds(value(w_o, u), 1) & \text{:-} & contains\_input(g, u, 1). \\
all\_inputs(g, 1, 1) & \text{:-} & not\ not\_all\_inputs(g, 1, 1). \\
all\_inputs(g, 0, 1) & \text{:-} & not\ not\_all\_inputs(g, 0, 1). \\
holds(value(w_o, u), 1) & \text{:-} & not\ \neg holds(value(w_o, u), 1). \\
\neg holds(value(w_o, 0), 1) & \text{:-} & holds(value(w_o, 1), 1). \\
\neg holds(value(w_o, u), 1) & \text{:-} & holds(value(w_o, 1), 1). \\
\neg holds(value(w_o, 0), 1) & \text{:-} & holds(value(w_o, u), 1). \\
\neg holds(value(w_o, 1), 1) & \text{:-} & holds(value(w_o, u), 1).
\end{array}
$$

The partial evaluation of the top $t_{H_1}(P_3^I)$ with respect to H_1 and $\mathcal{A}_{b_{H_1}}$, i.e. $P_4^I = e_{H_1}(t_{H_1}(P_3^I), \mathcal{A}_{b_{H_1}})$ consists of rules

$$
\begin{array}{rcl}
holds(value(w_o, 1), 1) & \text{:-} & all\_inputs(g, 1, 1). \\
all\_inputs(g, 1, 1). & & \\
holds(value(w_o, u), 1) & \text{:-} & not\ \neg holds(value(w_o, u), 1). \\
\neg holds(value(w_o, 0), 1) & \text{:-} & holds(value(w_o, 1), 1). \\
\neg holds(value(w_o, u), 1) & \text{:-} & holds(value(w_o, 1), 1). \\
\neg holds(value(w_o, 0), 1) & \text{:-} & holds(value(w_o, u), 1). \\
\neg holds(value(w_o, 1), 1) & \text{:-} & holds(value(w_o, u), 1).
\end{array}
$$

We will show that P_4^I also has a unique answer set. Let H_2 be the set of literals of the form

1. $all\_inputs(g, 1, 1)$,
2. $holds(value(w_o, 1), 1)$,
3. $\neg holds(value(w_o, u), 1)$.

Set H_2 splits P_4^I. The bottom $b_{H_2}(P_4^I)$ consists of rules

$$\begin{array}{rcl} holds(value(w_o, 1), 1) & :- & all\_inputs(g, 1, 1). \\ all\_inputs(g, 1, 1). & & \\ \neg holds(value(w_o, u), 1) & :- & holds(value(w_o, 1), 1). \end{array}$$

It is easy to see that $b_{H_2}(P_4^I)$ has unique answer set

$$\mathcal{A}_{b_{H_2}} = \{all\_inputs(g, 1, 1), holds(value((w_o, 1), 1), \neg holds(value(w_o, u), 1)\}.$$

Top $t_{H_2}(P_4^I)$ consists of rules

$$\begin{array}{rcl} holds(value(w_o, u), 1) & :- & not\ \neg holds(value(w_o, u), 1). \\ \neg holds(value(w_o, 0), 1) & :- & holds(value(w_o, 1), 1). \\ \neg holds(value(w_o, 0), 1) & :- & holds(value(w_o, u), 1). \\ \neg holds(value(w_o, 1), 1) & :- & holds(value(w_o, u), 1). \end{array}$$

The partial evaluation of the top $t_{H_2}(P_4^I)$ with respect to H_2 and $\mathcal{A}_{b_{H_2}}$, i.e. $P_5^I = e_{H_2}(t_{H_2}(P_4^I), \mathcal{A}_{b_{H_2}})$, consists of rules

$$\begin{array}{lll} \neg holds(value(w_o, 0), 1). & & \\ \neg holds(value(w_o, 0), 1) & :- & holds(value(w_o, u), 1). \\ \neg holds(value(w_o, 1), 1) & :- & holds(value(w_o, u), 1). \end{array}$$

Clearly, P_5^I has the unique answer set

$$\mathcal{A}_{P_5^I} \quad = \quad \{\neg holds(value(w_o, 0), 1)\}.$$

By the Splitting Set Theorem, we conclude that if for every $v_i \in \mathcal{I}$, $v_i = 1$, then P_2 has unique answer set $\mathcal{A}_2 = \mathcal{A}_{b_{H_0}} \cup \mathcal{A}_{b_{H_1}} \cup \mathcal{A}_{b_{H_2}} \cup \mathcal{A}_{P_5^I}$, i.e.

$$\begin{aligned} \mathcal{A}_2 \quad = \quad & \{holds(value(w_i, 1), 1), \neg holds(value(w_i, u), 1), \neg holds(value(w_i, 0), 1), \\ & not\_all\_inputs(g, u, 1), not\_all\_inputs(g, 0, 1), \\ & all\_inputs(g, 1, 1), contains\_input(g, 1, 1), \\ & holds(value(w_o, 1), 1), \neg holds(value(w_o, u), 1), \neg holds(value(w_o, 0), 1)\} \end{aligned}$$

and since $holds(value(w_i, 1), 1) \in \mathcal{A}_2$, we conclude the proof for Case 1.

Case 2. There exists $v_k \in \mathcal{I}$ such that $v_k = 0$, which implies that

$$occurs(apply(w_k, 0), 0) \in \mathcal{O}. \tag{4.61}$$

We need to show that program P_2 has a unique answer set, $\mathcal{A}_2$, and that $holds(value(w_o, 0), 1) \in \mathcal{A}_2$.

Let $\mathcal{A}_2$ be an answer set of P_2.

Given statement (4.61) and rule (4.52) of P_2, we have that

$$holds(value(w_k, 0), 1) \in \mathcal{A}_2. \tag{4.62}$$

By rule (4.58) of P_2 and statement (4.62), it holds that

$$contains\_input(g, 0, 1) \in \mathcal{A}_2. \tag{4.63}$$

Since $contains\_input(g, 0, 1)$ is a consequence of program P_2, it falsifies rule (4.55) of P_2.

By rule (4.60) of P_2 and statement (4.62), we can conclude that

$$\neg holds(value(w_k, u), 1) \in \mathcal{A}_2 \tag{4.64}$$

and

$$\neg holds(value(w_k, 1), 1) \in \mathcal{A}_2. \tag{4.65}$$

Again, because of statement (4.62) and rule (4.56), it follows that

$$not\_all\_inputs(g, u, 1) \in \mathcal{A}_2. \tag{4.66}$$

From statement (4.66) and rule (4.57) of P_2, we conclude that

$$all\_inputs(g, u, 1) \notin \mathcal{A}_2. \tag{4.67}$$

By rule (4.56) of P_2 and statement (4.62), it follows that

$$not\_all\_inputs(g, 1, 1) \in \mathcal{A}_2. \tag{4.68}$$

From statement (4.68) and rule (4.57) of P_2, we conclude that

$$all\_inputs(g, 1, 1) \notin \mathcal{A}_2. \tag{4.69}$$

Statement (4.69) falsifies rule (4.53) of P_2, and since atom $holds(value(w_o, 1), 1)$ does not appear as head of any other rule of P_2, we can conclude that

$$holds(value(w_o, 1), 1) \notin \mathcal{A}_2. \tag{4.70}$$

By statement (4.63) and rule (4.54) of P_2, it follows that

$$holds(value(w_o, 0), 1) \in \mathcal{A}_2. \tag{4.71}$$

Statement (4.71) and rule (4.60) of P_2 imply that

$$\neg holds(value(w_o, 1), 1) \in \mathcal{A}_2 \tag{4.72}$$

and

$$\neg holds(value(w_o, u), 1) \in \mathcal{A}_2. \tag{4.73}$$

Statement (4.73) falsifies rule (4.59) of P_2 for $w = w_o$. The only rule left in P_2 with an atom of the form $holds(value(w_o, s), 1)$ in the head, where $s \in \{0, 1\}$, is rule (4.54), hence no contrary literals for w_o can be derived from P_2. Thus, no literals contrary to $holds(value(w_o, s), 1)$ belong to $\mathcal{A}_2$.

Now we need to consider all other input values on input wires $w_i \in \{w_1, \ldots, w_n\}$ such that $w_i \neq w_k$. Let us consider such w_j. There are two possibilities.

(a) If $w_j \in \{w_1, \ldots, w_n\}$, where $w_j \neq w_k$, such that $occurs(apply(w_j, v_j), 0) \in \mathcal{O}$, then from rule (4.52) of P_2, it follows that

$$holds(value(w_j, v_j), 1) \in \mathcal{A}_2. \tag{4.74}$$

Statement (4.74) and rule (4.60) of P_2 imply that

$$\neg holds(value(w_j, v_j'), 1) \in \mathcal{A}_2 : v_j \neq v_j' \tag{4.75}$$

Statement (4.75) implies that

$$\neg holds(value(w_j, u), 1) \in \mathcal{A}_2. \tag{4.76}$$

which falsifies rule (4.59) of P_2 for $w_j \neq w_k$. Therefore

$$holds(value(w_j, u), 1) \notin \mathcal{A}_2.$$

(b) If $w_j \in \{w_1, \ldots, w_n\}$, where $w_j \neq w_k$, such that $v_j = u$, P_2 does not contain rule (4.52) for w_j. This implies that

$$holds(value(w_j, 0), 1) \notin \mathcal{A}_2 \tag{4.77}$$

and

$$holds(value(w_j, 1), 1) \notin \mathcal{A}_2. \tag{4.78}$$

By statements (4.77) and (4.78), and since rule (4.60) is the only rule with a head of the form $\neg holds(value(w,s),1)$ in P_2, we can conclude that

$$\neg holds(value(w_j,u),1) \notin \mathcal{A}_2. \tag{4.79}$$

Statement (4.79) and rule (4.59) of P_2 imply that

$$holds(value(w_j,u),1) \in \mathcal{A}_2. \tag{4.80}$$

Therefore, no contrary literals for can be derived from P_2 for this case too.

Finally, consider the case when all the input signals v_j on the input wires are equal to 0. It is easy to see that in this case $\mathcal{A}_2$ contain neither

$contains\_input(g,1,1)$, $contains\_input(g,u,1)$, nor $not\_all\_inputs(g,0,1)$. Hence, by rule (4.57), $\mathcal{A}_2$ must contain $all\_inputs(g,0,1)$.

If at least one input value is different of 0, $\mathcal{A}_2$ cannot contain $all\_inputs(g,0,1)$, and must contain

$not\_all\_inputs(g,0,1)$. In this case, if $v_j = 1$ and/or $v_j = u$ then $\mathcal{A}_2$ must contain $contains\_input(g,1,1)$ and/or $contains\_input(g,u,1)$, respectively.

The above argument can be viewed as a construction of a set $\mathcal{B}$ which must be a subset of any answer set $\mathcal{A}_2$ of P_2. We will show that $\mathcal{B}$ is an answer set of P_2. To do that, let us take $\mathcal{B}$ and construct the reduct of P_2 with respect to $\mathcal{B}$. From the construction it is easy to see that $\mathcal{B}$ is an answer set of the reduct of P_2, and by the definition of answer sets $\mathcal{B}$ is an answer set of P_2.

To prove uniqueness of this answer set, assume that $\mathcal{A}$ is an answer set of P_2. By construction, $\mathcal{B} \subseteq \mathcal{A}$. By the anti-chain property of answer sets, $\mathcal{B} = \mathcal{A}$.

Since $\mathcal{B}$ is the unique answer set of P_2 and $holds(value(w_o, 0), 1) \in \mathcal{B}$, we conclude the proof for Case 2 of Lemma 4.2.

Case 3. Assume $\forall v_i \in \mathcal{I},\ v_i \neq 0$, and $\exists v_k \in \mathcal{I}$ such that $v_k = u$.

We need to show that program P_2 has a unique answer set, $\mathcal{A}_2$, and that $holds(value(w_o, u), 1) \in \mathcal{A}_2$.

Let $\mathcal{A}_2$ be an answer set of P_2.

By the assumption that $\forall v_i \in \mathcal{I},\ v_i \neq 0$, we have that for every $1 \leq i \leq n$

$$occurs(apply(w_i, 0), 0) \notin \mathcal{O}. \tag{4.81}$$

By the assumption that $\exists v_k \in \mathcal{I}$ such that $v_k = u$, it follows that

$$occurs(apply(w_k, 1), 0) \notin \mathcal{O}. \tag{4.82}$$

By rule (4.52) of P_2 and statement (4.81), it follows that for every $1 \leq i \leq n$

$$holds(value(w_i, 0), 1) \notin \mathcal{A}_2. \tag{4.83}$$

By statement (4.83) and since $contains\_input(g, 0, 1)$ can only be deduced from rule (4.58) of P_2, it holds that

$$contains\_input(g, 0, 1) \notin \mathcal{A}_2, \tag{4.84}$$

which falsifies rule (4.54) of P_2.

By statement (4.81) and rule (4.52) of P_2, it follows that

$$holds(value(w_k, 0), 1) \notin \mathcal{A}_2, \tag{4.85}$$

while statement (4.82) and rule (4.52) of P_2 imply that

$$holds(value(w_k, 1), 1) \notin \mathcal{A}_2. \tag{4.86}$$

By rule (4.60) of P_2 and statements (4.85) and (4.86), we can conclude that

$$\neg holds(value(w_k, u), 1) \notin \mathcal{A}_2. \tag{4.87}$$

By rule (4.59) of P_2 and statement (4.87), it holds that

$$holds(value(w_k, u), 1) \in \mathcal{A}_2. \tag{4.88}$$

By rule (4.60) of P_2 and statement (4.88), we have that,

$$\neg holds(value(w_k, 0), 1) \in \mathcal{A}_2 \tag{4.89}$$

and

$$\neg holds(value(w_k, 1), 1) \in \mathcal{A}_2. \tag{4.90}$$

By rule (4.58) of P_2 and statement (4.88), it follows that

$$contains\_input(g, u, 1) \in \mathcal{A}_2. \tag{4.91}$$

By rule (4.55) of P_2 and statements (4.84) and (4.91), we conclude that

$$holds(value(w_o, u), 1) \in \mathcal{A}_2. \tag{4.92}$$

By rule (4.56) of P_2 and statement (4.88), we have that

$$not\_all\_inputs(g, 0, 1) \in \mathcal{A}_2 \tag{4.93}$$

and

$$not\_all\_inputs(g, 1, 1) \in \mathcal{A}_2. \tag{4.94}$$

By rule (4.57) of P_2 and statement (4.93), it follows that

$$all\_inputs(g, 0, 1) \notin \mathcal{A}_2, \tag{4.95}$$

and by rule (4.57) of P_2 and statement (4.94), we have that

$$all\_inputs(g, 1, 1) \notin \mathcal{A}_2, \tag{4.96}$$

which falsifies rule (4.53) of P_2.

By rule (4.60) of P_2 and statement (4.92), it follows that

$$\neg holds(value(w_o, 0), 1) \in \mathcal{A}_2, \tag{4.97}$$

and

$$\neg holds(value(w_o, 1), 1) \in \mathcal{A}_2. \tag{4.98}$$

Since both rules (4.53) and (4.54) were falsified, and there are no other rules of P_2 whose head is of form $holds(value(w_o, v), 1)$, where $v \in \{0, 1\}$, then

$$holds(value(w_o, 0), 1) \notin \mathcal{A}_2, \tag{4.99}$$

and

$$holds(value(w_o, 1), 1) \notin \mathcal{A}_2, \tag{4.100}$$

which together with rule (4.60), implies that

$$\neg holds(value(w_o, u), 1) \notin \mathcal{A}_2. \tag{4.101}$$

Finally, consider the case when $w_j \in \{w_1, \ldots, w_n\}$, $w_j \neq w_k$, such that $v_j = u$. It is easy to see that in this case $\mathcal{A}_2$ contains neither $contains\_input(g, 1, 1)$, nor $not\_all\_inputs(g, u, 1)$, and hence, it must contain $all\_inputs(g, u, 1)$.

If all input values v_j are equal to 1, it is easy to see that $\mathcal{A}_2$ must contain $not\_all\_inputs(g, u, 1)$, and $contains\_input(g, 1, 1)$. Hence, it cannot contain $all\_inputs(g, u, 1)$.

The above argument can be viewed as a construction of a set $\mathcal{B}$ which must be a subset of any answer set $\mathcal{A}_2$ of P_2. We will show that $\mathcal{B}$ is an answer set of P_2. To do that, let us take $\mathcal{B}$ and construct the reduct of P_2 with respect to $\mathcal{B}$. From the construction it is easy to see that $\mathcal{B}$ is an answer set of the reduct of P_2, and by the definition of answer sets $\mathcal{B}$ is an answer set of P_2.

To prove uniqueness of this answer set, assume that $\mathcal{A}$ is an answer set of P_2. By construction, $\mathcal{B} \subseteq \mathcal{A}$. By the anti-chain property of answer sets, $\mathcal{B} = \mathcal{A}$.

Since $\mathcal{B}$ is the unique answer set of P_2 and $holds(value(w_o, u), 1) \in \mathcal{B}$, we conclude the proof for Case 3 of Lemma 4.2.

Lemma 4.2 follows immediately from Cases 1, 2, and 3. □

4.4 Proof of Lemma 4.3 - OR gate

Lemma 4.3. *Let C be a combinational circuit consisting of a single* OR *gate g with input wires $w_1, \ldots, w_n$, output wire w_o, and no delays. Let $v_1, \ldots, v_n$ be an input signal vector of C and let $\mathcal{O} = \{occurs(apply(w_i, v_i), 0) \; : \; w_i \in \{w_1, \ldots, w_n\}, v_i \in \{0, 1\}$, for $1 \leq i \leq k, k \leq n\}$. Then*

1. *Program P_0 has unique answer set; and*
2. *If $\mathcal{A}_0$ is the unique answer set of P_0 then*

$$v = \text{OR}(v_1, \ldots, v_n) \;\; \textit{if and only if} \;\; holds(value(w_o, v), 1) \in \mathcal{A}_0.$$

□

Proof. - Follows immediately from Lemma 4.2, and the application of the *Principle of Duality* which characterizes the AND and OR operations of Boolean algebra.

4.5 Proof of Proposition 3.1

We now prove

Proposition 3.1. *Let C be a combinational circuit, with input wires $w_1, \ldots, w_n$, output wire w_o, and no delays, which computes a function $f(S_1, \ldots, S_n)$. Then for any input vector $s_1, \ldots, s_n$ of 0's, 1's, and u's, program P_0 has a unique answer set and $holds(value(w_o, s), 1) \in CT(\mathcal{D})$ if and only if $s = f(s_1, \ldots, s_n)$, where $\mathcal{D} = \pi(C) \cup \{occurs(apply(w_i, s_i), 0) \;:\; w_i \in \{w_1, \ldots, w_n\}, s_i \in \{0, 1\}, 1 \leq i \leq k, k \leq n\}$.*

$\square$

Proof. The proof is by induction on the number m of gates in C.

Base case: $m = 1$. Follows immediately from Lemmas 4.1, 4.2, and 4.3.

Induction step: Suppose that we have proved Proposition 3.1 for $m \geq 1$. Now, we need to show that the proposition also holds for $m + 1$.

Let C_{m+1} be a combinational circuit with $m+1$ gates. C_{m+1} can always be decomposed [181] into circuits C_m and C_1 shown in Figure 4.1.

Note, that the sets W_m and W_1 of input wires of C_m and C_1 are not necessarily disjoint and that the set W_{m+1} of input wires of C_{m+1} is equal to $(W_0 \cup W_1)$ where $W_0 = W_m \setminus \{w_o^1\}$. The decompostion has the following property:

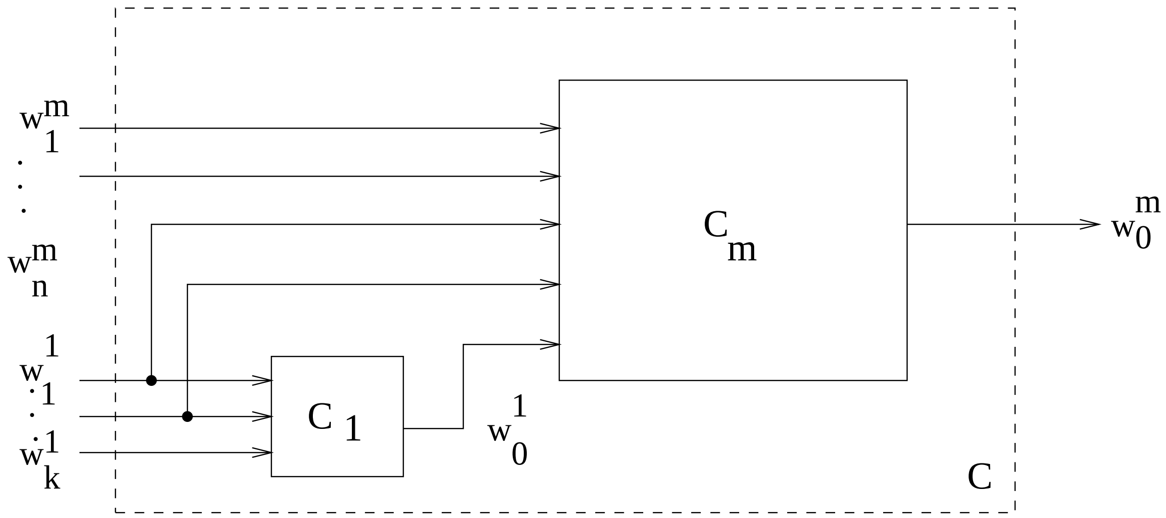


Figure 4.1: Blocks diagram for digital circuit C decomposed into circuits C_m and C_1.

For every set of input signals I_{m+1} assigned to the input wires of C_{m+1}

$$f_{m+1}(I_{m+1}) = f_m(I_0, f_1(I_1)). \tag{4.102}$$

Here I_0 and I_1 are input signals from I_{m+1} applied to the wires of W_0 and W_1 and f_{m+1}, f_m and f_1 are functions defined by the circuits C_{m+1}, C_m, and C_1. (Without loss of generality we assume that w_o^1 is the last argument of f_m.)

Let P_{m+1} and P_1 be lp-descriptions of C_{m+1} with input I_{m+1} and C_1 with input I_1 respectively. By the inductive hypothesis P_1 has a unique answer set, A_1, such that

$$holds(value(w_o^1, s), 1) \in A_1 \text{ if and only if } s = f_1(I_1). \tag{4.103}$$

It is easy to see that $U_0 = lit(P_1)$ is a splitting set of P_{m+1} and hence, by the Splitting

Set Theorem and the uniqueness of A_1

$$A_{m+1} \text{ is an answer set of } P_{m+1} \text{ iff it is an answer set of program } T_1 = A_1 \cup P_{m+1}. \tag{4.104}$$

To apply the inductive hypothesis we need to establish the relationship between this program and the lp-description P_m of the circuit C_m with input wires W_m. Let $W_1' = W_1 \setminus W_m$ be the set of input wires of C_1 different from that of C_m and let U_1 be the set of literals of P_1 which contain names of the wires from W_1' or w_0^1.

U_1 is a splitting set of T_1. The program $b_{U_1}(T_1)$ can be viewed as an lp-representation of a new circuit, C_1' obtained from C_1 by removing the input wires common to C_m. Hence, by the Lemmas 4.1, 4.2, and 4.3, it has the unique answer set, A_0. Let $R_1 = e_{U_1}(t_{U_1}(P_{m+1}), A_0)$ be the result of the partial evaluation of P_{m+1} with respect to A_0 and $R = R_1 \cup holds(w_0^1, s, 1)$ where $s = f_1(I_1)$. From equation (4.104) and the Splitting Set Theorem, we have that

$$A_{m+1} \text{ is an answer set of } P_{m+1} \text{ iff it is an answer set of program } T_2 = A_1 \cup R. \tag{4.105}$$

Now let us notice that P_m contains information about wire w_o^1 which does not belong to R - a rule

(a) $holds(value(w_o^1, S), 1) \leftarrow occurs(apply(w_o^1, S), 0).$,

and the assignment $holds(value(w_o^1, u), 0)$, or $occurs(apply(w_o^1, s), 0)$, to this wire.

(b) R contains $holds(value(w_0^1, s), 1)$, where $s = f_1(I_1)$, while P_m does not.

These two conditions imply that R is the partial evaluation of P_m with respect to the set $\{occurs(apply(w_o^1, S), 0), holds(value(w_o^1, u), 1)\}$.

By the inductive hypothesis for C_m and the construction of its input, we have that P_m has the unique answer set $\mathcal{B}$ such that

$$holds(value(w_o, v), 1) \in \mathcal{B} \text{ if and only if } v = f_m(I_0, s)$$

where I_0 is the assignment given by I_{m+1} to the wires from W_0, and $s = f_1(I_1)$.

This, together with equations (4.102), (4.104), and the construction of A_0 guarantees that P_{m+1} has a unique answer set A_{m+1}, and that $holds(value(w_o, v), 1) \in A_{m+1}$ if and only if $v = f_{m+1}(I_{m+1})$.

This concludes the proof of Proposition 3.1. $\square$

Chapter 5

The Reaction Control System - Action Theory and Answer Set Programming for Controling the Space Shuttle

> *"To advance and communicate scientific knowledge and understanding of the earth, the solar system, and the universe. To advance human exploration, use, and develoment of space. To research, develop, verify, and transfer advanced aeronautics and space technologies."*
>
> *NASA Mission Statement [151]*

In this chapter we present and discuss in detail the application of the theory of action and change and the emergent programming paradigm - answer set programming - to a complex "real world" domain, the Reaction Control System (known as the RCS) of the space shuttle. The design and implementation of a system to control a complex medium-size domain in the answer set programming paradigm is one of the achievements of this research. The successful results thus far obtained with the system can be considered as a promising step in the use of answer set programming as a powerful and efficient tool for programming real world applications. This work is also the first application of such techniques to a real world domain of this size.

5.1 On NASA, the Space Exploration Program, USA, and the Space Shuttle

Created in 1958 to study and propel human space flight, the National Aeronautics and Space Administration (NASA) agency has collected innumerous unique scientific and technological achievements in areas far beyond space science and aeronautics. The agency's long list of important scientific discoveries has reached diverse fields of human knowledge and has impacted our lives in ways that were not foreseen in its inception. Both essential and ordinary every day items such as clothing, food, medicine, even pens, have been modified by such discoveries. Today a large portion of NASA's efforts are concentrated in help building the International Space Station.

NASA's space exploration program has answered fundamental questions about human space flight, aeronautics, the space and the planet earth through several always evolving projects. In particular, the dream of human space flight and space exploration was addressed through Project Mercury, the first manned space flight program which verified the possibility of human survival in space; Project Gemini that used double manned spacecrafts for two weeks long flights; and the program for scientific exploration of the moon, the Project Apollo that allowed for the landing of humans on the moon in 1969.

Economical requirements and political pressure for the exploration of space in a continous basis led to the development of a Space Transportation System (STS) consisting of reusable spacecrafts, commonly known as space shuttles. In 1981, NASA crossed yet another frontier with the successful first flight of the space shuttle *Columbia*[1] into space. From the original six space vehicles, the present shuttle (or orbiter) fleet is reduced to three operable spacecrafts: Discovery, Atlantis, Endevour, and the first orbiter Enterprise which has been used only as a test-bed for the shuttle program, but has never flown into space.

In 1996, NASA prompted the consolidation of the multiple Space Shuttle Program contracts under a single prime contractor. In particular, the flight support operations conducted by Rockwell and ground operations managed by Lockheed Martin were merged to form the United Space Alliance (USA), a Limited Liability Company which overviews the training of personnel and operation of the shuttle fleet, as well as the International Space Station. Shortly after the contract was effectuated, Boeing Corporation bought Rockwell's share of USA and became part of the space flight program. Today USA maintains the safety and reliable management of the space shuttle fleet as its primary goals, and is constantly searching for new tools to help in achieving these goals. Eighty percent of the seven billion Phase I contract, covering a period of six years, between USA and NASA is attached to maintaining safety and

[1]The space shuttle Columbia and its seven crewmembers were recently lost on their landing descent to Kennedy Space Center on February 1, 2003. The cause(s) of the accident are presently still under investigation. Almost twenty years earlier, all seven crewmembers and the Challenge space shuttle were destroyed shortly after launching on January 28, 1986 when a booster failure caused the breakup of the vehicle.

related standards. A crucial task to ensure the safe launch, orbiting, and return of the space shuttle is flight control.

The space shuttle vehicle contains approximately 2,506,450 parts, from which nearly 2,000,000 comprise the orbiter; the remaining parts belonging to the external tank and solid rocket boosters. The space shuttle orbiter has more than a dozen sophisticated systems, including among others the main propulsion system, the thermal protection system, the orbital maneuvering system, the reaction control system, the electrical power system, and the environmental control and life support system. Each of the space shuttle orbiter's systems is subdivided into multiple subsystems which are supervised by a team of specially trained flight controllers assigned to it. Flight controller personnel are responsible for monitoring and resolving any problems affecting a system during a mission. When not on a mission assigment, flight controllers study possible future problems that can happen to the system they work with and generate solutions to these problems. Since the space shuttle systems are relatively complex involving a high number of components, multiple failures are possible and it becomes unfeasible to consider and find plans in advance to solve all such situations. When confronted with a multiple failure situation during a mission, flight controllers must rapidly come up with a correct solution. Pressured by strict requirements on both time and precision, flight controllers must perform near perfection. The cost of a single error can vary from abortion of a mission, in the best scenario, to loss of the space vehicle and the crew's lives, in the worst case. An interruption on the communication capability between a flight crew and the control center, overseing the mission from earth, can also require that the crew formulate a plan(s) to solve eventual problem(s).

A large collection of historical documents, reports, real time data, photos, interactive images, and tutorials, including all information presented in this section about NASA, its space exploration program, the space shuttle and its missions is available online at NASA's web page [151]. Details about USA and the operation of the space shuttle program are also available online at USA's web page [195]. In the next sections we discuss the Reaction Control System of the space shuttle.

5.2 The RCS and the USA-Advisor Systems

The RCS is the shuttle's system that has primary responsibility for maneuvering the aircraft while it is in space. It consists of fuel and oxidizer tanks, valves and other plumbing needed to provide propellant to the maneuvering jets of the shuttle. It also includes electronic circuitry: both to control the valves in the fuel lines and to prepare the jets to receive firing commands.

The RCS is computer controlled during takeoff and landing. While in orbit, however, astronauts have the primary control. When an orbital maneuver is required, the astronauts must perform whatever actions are necessary to prepare the RCS. These actions generally require flipping switches, which are used to open or close valves or to energize the proper circuitry. In more extreme circumstances, such as a faulty switch, the astronauts communicate the problem to the ground flight controllers, who will come up with a sequence of computer commands to perform the desired task and will instruct the shuttle's computer to execute them.

During normal shuttle operations, there are pre-scripted plans that tell the astronauts what should be done to achieve certain goals. The situation changes when there are failures in the system. The number of possible sets of failures is too large to pre-plan for all of them. Continued correct operation of the RCS in such circumstances is necessary to allow for the completion of the mission and to help ensure the safety of the crew.

The RCS/USA-Advisor System[2] presented here can be viewed as a part of a decision support system for shuttle flight controllers. It is an intelligent system capable of verifying and generating plans that prepare the RCS for the required maneuver.

Part of this dissertation builds on previous work [31, 81, 202, 203] in which the authors developed a prototype of a system, denoted by M_0, capable of checking correctness of plans. The system was based on the programming language Prolog and, to a certain extent, was tailored toward its inference engine. One of the main contributions of this work is the development of the new, substantially more powerful, model of the RCS not suffering from these limitations. In particular, we

1. substantially simplify the model of the part of the RCS represented by M_0 without loss of detail,

[2]Referred to simply as USA-Advisor.

2. implement the new model in a different programming paradigm - answer set programming,

3. include information about electrical circuits of the RCS, which was missing in M_0,

4. include a new type of action – computer commands, controlling the position of valves,

5. include a planning module(s) containing a large amount of heuristic information (this substantially improves the quality of the plans and efficiency of the search),

6. include a Java interface to simplify the use of the system by a flight controller and by the system designers.

The resulting system, USA-Advisor[3], is now suitable for practical applications. This project has been funded by United Space Alliance, as mentioned before, the company contracted by NASA to overview the shuttle's operation and missions. Programmers from USA have recently started the work of modifying the interface of the system in order to customize the system for its deployment.

To understand the functionality of the USA-Advisor let us imagine a shuttle controller who is considering how to prepare the shuttle for a maneuver when faced with a collection of faults present in the RCS (for example, switches and valves can be stuck

[3]The RCS/USA-Advisor system is available for download from: http://krlab.cs.ttu.edu/~marcy/RCS/

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 (a plan) to ready the shuttle for the maneuver. Finding manually such a plan is, in general, not a trivial task. The most difficult part, however, is proving that the plan will achieve the expected results, given the current conditions of the shuttle, without causing any possibly dangerous side effect. The RCS/USA-Advisor can serve as a tool facilitating this task. First of all, 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, and has no side effects. Moreover, the system can be used to automatically find such a plan, which is therefore guaranteed to be correct. It is expected that, in the near future, the USA-Advisor will be mainly employed in this second way. In emergency situations, it will be used "on-line" to generate plans that achieve the desired goal. The rest of the time, the system will be used "off-line," generating pre-packaged plans for situations that might occur in future missions.

The main issues involved in building the USA-Advisor are:

- Modeling the RCS as a dynamic domain: this includes representing information at multiple levels of detail. At the lowest level we need to describe the effects of the valves positions on the plumbing system. At the highest level we specify the electrical circuits used to control the valves.

- Representing knowledge in several separate modules and combining the appropriate modules depending on the task given to the system – notice that one of the modules had been independently developed before the start of the USA-Advisor project.

- Developing a planning module containing a large amount of heuristic information (which substantially improves quality of the plans and efficiency of the search).

The solutions devised for the correct modeling and implementation of the RCS in the answer set programming paradigm and part of the methodologies developed to attack these issues are original work developed by this research and constitute some of its contributions.

5.3 The RCS System

The RCS is the system used to maneuver the space shuttle while it is in orbit, e.g. during the "separation burn" phase to distance itself from the space station. During a mission, this system is used to roll or move the spacecraft in the direction required for a photography or by an experiment to be accomplished by the crewmembers. Three subsystems form the RCS system: the Forward RCS, located on the forward fuselage nose area of the orbiter; the Left RCS, and the Right RCS, both located with the Orbital Maneuvering System (OMS) in the aft fuselage of the orbiter vehicle. The RCS subsystems provide the thrust for attitude (rotational) maneuvers (pitch, yaw and roll) and also allow for translation maneuvers through small changes in velocity along the orbiter axis.

The propellants for the RCS jets, or thrusters, are stored on fuel and oxidizer tanks, pressurized with helium, and are distributed through several different types of pressure regulation and relief valves (namely tank isolation, manifold isolation, and crossfeed valves), distribution (here termed plumbing) lines and filling and draining connections, termed junction. There exists a physical interconnection between the Left and Right RCS in the OMS pod, and also between the OMS and the aft RCS systems allowing the RCS to utilize the OMS's propellant for firing its jets. This provision is part of the redundancy capabilities added to the space shuttle to ensure the safety of its operation. In case of failure of an OMS engine, the aft RCS can be utilized to complete any OMS deorbit thrusting period.

The RCS jets are of two different types: (a) primary thrusters, which are robust jets; and (b) smaller engines called vernier thrusters. In total there are 38 primary and 6 vernier thrusters in the RCS divided in the following way: the forward RCS has 14 primary and two vernier engines, and the Left and Right RCS have 12 primary and two vernier jets each. The flight crew can select which jets to use for attitude control in orbit; vernier thrusters are used normally for on-orbit attitude hold. It must be noted that no redundancy is provided for vernier jets.

In order for the space shuttle to perform a given maneuver, a set of jets, belonging to the correct subsystems and pointing in the correct directions, must be prepared to fire. Preparing a jet to fire involves providing an open, non-leaking path for the fuel to flow from pressurized fuel tanks to the jet. The flow of fuel is controlled by opening and closing pressure regulation and relief valves. Valves are opened and closed by either having an astronaut flip a switch or by instructing the onboard computer to issue special commands. In a very simplified form, the RCS can be viewed as the directed graph in Figure 5.1 whose nodes are tanks, jets and pipe junctions, and whose arcs are labeled by valves. Switches are connected to valves through fairly complex electrical circuits.

5.4 USA-Advisor System's Design

The USA-Advisor system consists of a collection of largely independent modules, represented by lp-functions[4] [72], and a graphical Java interface[5], J. The interface gives simple means for the user to enter information about the history of the RCS, its faults, and the task to be performed.

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 goal, G, and finding a plan for G of a length not exceeding some number of steps, N. Based on this information, J verifies that the input is complete, selects an appropriate combination of modules, assembles them into an A-Prolog program, Π, and passes Π as an input to a reasoning system for computing stable models (In the USA-Advisor this role is currently played

[4] In more precise terms, an lp-function is a 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.

[5] The graphical interface for the USA-Advisor system was implemented primarily by Marcello Balduccini.

by SMODELS, however we also plan to investigate performance of other systems.) In this approach the task of checking a plan P is reduced to checking if there exists a model of the program $\Pi \cup P$. A planning module is used to describe a set of possible plans the user is interested in. The general correctness theorem [123] from the theory of action guarantees that there is a one-to-one correspondence between the plans and the set of stable models of the program. Planning is reduced to finding such models. Finally, the Java interface extracts the appropriate answer from the SMODELS output and displays it in a user-friendly format.

In our design, the RCS is described at two levels of detail, the appropriate level being selected depending on the task to be performed. At the highest level of abstraction, electrical circuits are assumed to be working correctly. Thus, their internal functioning can be ignored, and the function they compute is described explicitly in terms of the effects that switches and computer commands have on the corresponding valves. At the lowest level of abstraction, used when electrical circuits contain faulty components, circuits are represented explicitly.

The RCS is decomposed in four main modules: the Plumbing Module, the Valve Control Module, the Circuit Theory Module, and the Planning Module. The Plumbing Module models the plumbing system of the RCS. The Valve Control Module describes how switches and computer commands affect the position of valves. The Circuit Theory Module describes the behavior of standard combinatorial digital circuits, augmented with other components, like delay units, power units, switches, and valves. The Planning Module is responsible for generating plans achieving the desired goal, and contains a large number of heuristics aimed at improving both the quality of plans and the efficiency of the planner. Additional modules provide the description of the schematics of each electrical circuit.

In the rest of this section we give a detailed description of particular modules.

5.4.1 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, G, shown in Figure 5.1, 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. Pressurization of fuel and oxidizer tanks is obtained by releasing

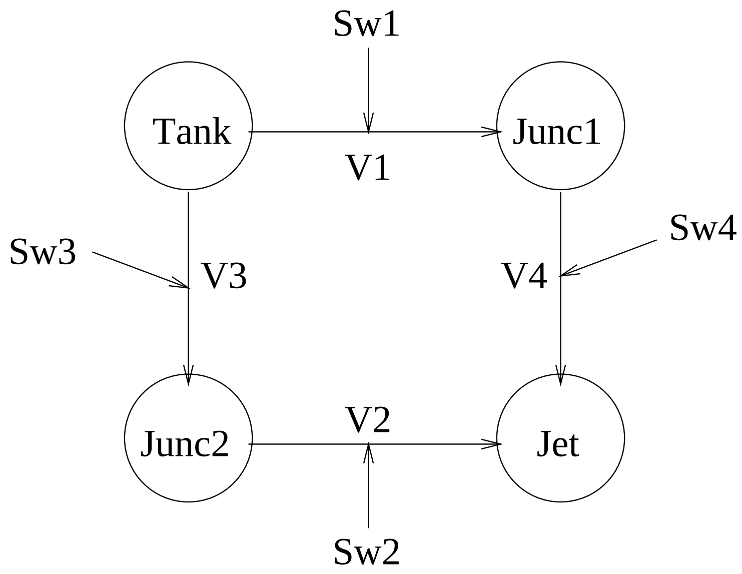


Figure 5.1: A simplified view of the RCS.

helium from the helium tanks connected to the fuel and oxidizer tanks. The necessary condition for a fluid to flow from a tank to a jet, and in general to any node of G, 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, G, of the plumbing system, its state, including position of valves and the list of faulty components, and determines: (a) the distribution of pressure through the nodes of G, (b) the jets ready to fire, and (c) the maneuvers ready to be performed.

The elements of the plumbing system are represented in PM as follows. The arcs of graph G are described by relation *link(N1,N2,V)* which holds iff G contains a directed arc from node $N1$ to $N2$ and this arc is labeled by the valve V. For instance, a statement *link(ffh,ff,ffha)* says that fuel helium tank *ffh* is connected to fuel propellant tank *ff* by valve *ffha*. Relations *jets_of(J,R)* and *vernier_of(J,R)* identify jets and verniers and the subsystem they belong to. As explained in Section 5.3, the RCS is partitioned into three subsystems: 1. Forward RCS, located on the forward fuselage nose area of the orbiter; 2. Left RCS, and 3. Right RCS, both located with the Orbital Maneuvering System (OMS) in the aft fuselage of the orbiter vehicle. The subsystems of the RCS are identified by statements: *system(fwd_rcs)*, *system(left_rcs)*, and *system(right_rcs)*. So, for instance, a statement *jet_of(f1u,fwd_rcs)* says that jet *f1u* belongs to the forward subsystem of the RCS. Relation *direction(J,D)* specifies the direction of jets and verniers. For instance, statement *direction(f1u,up)* says that jet *f1u* is directed upwards. There are six different possible directions different jets point to: up, down, left, right, forward, and aft. The downward firing jets on the Forward RCS must always operate in pairs, one on each side (left and right). To facilitate this operation, a list of such pairs is kept in the form of statements *pair_of_jets(J1,J2)*. For example, a statement *pair_of_jets(f1d,f2d)* indicates that jets *f1d, f2d* constitute one of such pairs. If one of the jets is not functional the other one cannot be fired. Relation *tank_of(T,R)* links each tank to the subsystem it belongs to. For instance, a statement *tank_of(ffh,fwd_rcs)* says that the forward fuel helium tank belongs to the forward subsystem. There are twelve possible maneuvers to be performed by firing jets of shuttle. They are: $+X$, $-X$, $+Y$, $-Y$, $+Z$, $-Z$, *+roll*, *-roll*, *+pitch*, *-pitch*, *+yaw*, *-yaw*. Statements of the form *maneuver(M)* list the types of maneuvers possible. For instance, statement *maneuver(plus_x)* indicates that *plus_x* is one such maneuver.

The initial state of the plumbing module is characterized by fluent *in_state(V,S)*, specifying that valve V is in state S (open or closed), and a collection of faulty components described by atoms *has_leak(V)* and *damaged(J)*, and relation *stuck(V,S)* (valve V is stuck in position S). It is assumed that all helium tanks are pressurized in the initial state and that normally functioning valves are initially closed. The last statement is expressed by the default

```
holds(in_state(V,closed),0) :-
                    of_type(V,valve),
                    not holds(in_state(V,open),0).
```

It is also assumed that normally functioning switches are initially in state GPC, i.e. are controlled by the on-board General Purpose Computers, which is described by default

```
holds(in_state(Sw,gpc),0) :-
                    of_type(Sw,v_switch),
                    not ¬holds(in_state(Sw,gpc),0).
```

The current state is described by a set of fluents that includes the fluents used for the initial state, together with fluent *ready_to_fire(J)*, where J is a jet, and *pressurized_by(N,TK)*, stating that fluid under pressure is flowing from tank TK to node N (we say that "N is pressurized by TK").

Each maneuver is described by a rule whose body can be satisfied by a collection of jets located in the corresponding RCS and pointed in the specified directions. Performing a maneuver corresponds to preparing such jets for firing. Fluent *ready_to_fire(J)* is true when jet J is simultaneously pressurized by both fuel and oxidizer tanks, and J is not damaged. A fluent *pressurized_by(N,TK)*, which reads "node N is pressurized by tank TK," is true if there is an open and non-leaking path from TK to N. To define such path we use an auxiliary fluent *leaking(N)* where N is a node of graph G.

The shuttle is ready for a maneuver M if and only if a set of jets satisfying the requirements for maneuver M is ready to fire. In order to increase the efficiency in planning the actions required for a maneuver, fluent *maneuver_of(M,R)* is used to indicate that the portion of maneuver M executed by RCS subsystem R is ready. If M does not require any action of RCS subsystem R, we add relation *done(M,R)* to the description. The following rule ensures that maneuver M of subsystem R is ready at time T.

```
holds(maneuver_of(M,R),T) :-
                                done(M,R).
```

In the case one or more RCS subsystems require actions to prepare jets to fire in order to perform a certain maneuver, the above rule is not applicable to these subsystems. Instead, a maneuver M of a subsystem R is ready at time T if the required jet J of R is ready to fire in direction D at time T, defined by rule

```
holds(maneuver_of(M,R),T) :-
```

```
                        jet_of(J,R),
                        direction(J,D),
                        holds(ready_to_fire(J),T).
```

For instance, the shuttle is ready for maneuver $+X$ (*plus_x*) if an aft jet is ready to fire on both the Left and Right RCS. This is defined by the following rules

```
holds(maneuver_of(plus_x,left_rcs),T) :-
                        jet_of(J,left_rcs),
                        direction(J,aft),
                        holds(ready_to_fire(J),T).

holds(maneuver_of(plus_x,right_rcs),T) :-
                        jet_of(J,right_rcs),
                        direction(J,aft),
                        holds(ready_to_fire(J),T).

done(plus_x,fwd_rcs).
```

(Note that since no jet from the forward rcs is required for this maneuver, statement $done(plus\_x, fwd\_rcs)$ was added to the description.)

To further illustrate the issues involved in the construction of the Plumbing Module, let us consider the definition of fluent $pressurized\_by(N, Tk)$, describing the pressure on a node N by a tank Tk. Helium tanks are treated as special nodes and presently assumed to be always pressurized. Hence, the definition of this relation for tank nodes is trivial. In the initial situation it is given by facts of the form

```
holds(pressurized_by(Tk,Tk),0).
```

where Tk corresponds to a constant identifying a tank.

The inertia rule below states that tanks maintain correct pressure in all subsequent situations unless new information is added through relation

$\neg holds(pressurized\_by(Tk, Tk), T')$, where $T' > 0$.

```
holds(pressurized_by(Tk,Tk),T+1) :-
                tank_of(Tk,R),
                holds(pressurized_by(Tk,Tk),T),
                not ¬holds(pressurized_by(Tk,Tk),T+1).
```

For other nodes, the definition is recursive. It says that any non-tank 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 standard Prolog is problematic, since the corresponding graph can contain cycles. (This fact is partially responsible for the relative complexity of this module in M_0.) 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.

```
holds(pressurized_by(N1,Tk),T) :-
                        not tank_of(N1,R),
                        tank_of(Tk,R),
                        not holds(leaking(N1),T),
                        link(N2,N1,V),
                        holds(in_state(V,open),T),
                        holds(pressurized_by(N2,Tk),T).
```

This rule states that non-tank node $N1$ is pressurized by tank Tk if $N1$ is not leaking and is connected by an open valve to a node which is pressurized by tank Tk. The relation that describes pressurization of tank nodes is

```
holds(pressurized_by(N1,Tk),T) :-
                        tank_of(N1,R),
                        tank_of(Tk,R),
                        link(N2,N1,V),
                        holds(in_state(V,open),T),
                        holds(pressurized_by(N2,Tk),T).
```

It says that tank node $N1$ is pressurized by tank Tk if $N1$ is connected by an open valve to a node which is pressurized by tank Tk.

Faults in the RCS system are indicated by the system's user, or possibly in the future by signals sent by sensors directly connected to different parts of the system. For instance, if there is a leak on a valve V then it is necessary to determine which nodes belonging to the same fuel path of V may be affected, i.e. leaking. This is easily achieved in A-Prolog by the following rules:

```
holds(leaking(N1),T) :-
                        link(N1,N2,V),
                        has_leak(V),
                        holds(in_state(V,open),T).
```

which says that a node is leaking at any time it is connected to a leaking valve which is open; and

```
holds(leaking(N1),T) :-
                        link(N1,N2,V),
                        holds(in_state(V,open),T),
                        holds(leaking(N2),T).
```

stating that a node is leaking if it is connected by an open valve to another node which is leaking.

There are two propellant lines (one for fuel and one for oxidizer) interconnecting the Left and the Right RCS subsystems called RCS-to-RCS crossfeed. Crossfeed valves control the flow of propellant through fuel junctions, denoted by *fxfeed*, and oxidizer junctions, *oxfeed*, in these lines. One of the RCS requirements is to avoid situations where crossfeed (junctions and valves) are simultaneously pressurized by two tanks from different subsystems. This can be nicely described in A-Prolog by constraints:

```
:- tank_of(X,R),
   tank_of(Y,R1),
   neq(X,Y),
   holds(pressurized_by(fxfeed,X),T),
   holds(pressurized_by(fxfeed,Y),T).


:- tank_of(X,R),
   tank_of(Y,R1),
   neq(X,Y),
   holds(pressurized_by(oxfeed,X),T),
   holds(pressurized_by(oxfeed,Y),T).
```

These constraints eliminate any models (solutions) where two different tanks simultaneously pressurize any of the crossfeed junctions, and consequently, crossfeed valves.

The Plumbing Module consists of approximately 40 rules.

5.4.2 Valve control module

The flow of fuel and oxidizer propellants from tanks to jets is controlled by opening/closing valves along the path connecting these nodes. 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.

In some specific phases of operation of the shuttle, such as launch and landing, the on-board general purpose computers, GPCs, will be in charge of opening/closing valves and will achieve this objective by sending computer commands. If the shuttle is in orbit, or the computer system is malfunctioning, an astronaut can normally override these commands by manually flipping the switches that control the valves to be opened/closed.

The Valve Control Module, VCM, describes how computer commands and changes in the position of switches affect the state of valves. 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.

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 relevant 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).

Effects of actions in the basic VCM are described in a variant of action language $\mathcal{B}$ [77], which contains both static and dynamic causal laws, as well as impossibility conditions. Recall that the dynamic causal law, a **causes** f **if** p, says that f will be true in a state the system moves to, after the execution of a in any state satisfying condition p. A static causal law, often referred to as a state constraint, is of the form f **if** p. It says that every state of the system satisfying condition p must also satisfy f. Note that the rules of the plumbing module can be viewed as state constraints - that module contains no dynamic causal laws. Our version of $\mathcal{B}$ uses a slightly different syntax to avoid lists and nesting of function symbols, because of limitations of the inference engines currently available.

The use of the semantics of $\mathcal{B}$ which is defined independently from the logic programming notions, allows one to prove correctness of the logic programming implementation of causal laws [72]. (Of course, it does not guarantee correctness of the causal laws *per se*. This can only be done by domain experts.)

Connections between switches and valves, termed *devices*, are described by relation *controls(Sw,V)* meaning that switch Sw controls the state of valve V. In the extended level, it is necessary to define that this connection is achieved through an electrical circuit. Rule

```
controls(Sw,V) :- controls(Sw,V,C).
```

allows us to have a single set of statements (of the form *controls(Sw,V,C)*) establishing the connection and to generate all the facts for both the basic and the extended valve control modules. For instance, statement *controls(fm1,ffm1,fmc1)* states that switch *fm1* controls valve *ffm1* through electrical circuit *fmc1* used in the extended level, and together with the rule above define the simplified connection *controls(fm1,ffm1)* used in the basic level. In the RCS some valves of critical importance can be moved in one position only by issuing two computer commands simultaneously. If a valve V can be moved to a state S by a single computer command, this is denoted by statements of the form *basic_command(CC,V,S)*. For instance, statement *basic_command(opena_ffha, ffha,open)* says that command *opena_ffha* opens valve *ffha*. There are more than 130 such commands.

Otherwise, statements of the form *commands(cc(CC1,CC2),V,S)* are employed to express that valve V requires computer commands *CC1* and *CC2* to be simultaneously issued in order to achieve the desired effect.

For example, statement *commands(cc(closea_fi12,closeb_ffi12),ffi12,closed)* says that to close valve *ffi12* it is necessary to issue simultaneously the commands: *closea_fi12* and *closeb_ffi12*.

An electrical malfunction of the circuitry controlling valve V is represented by statement *bad_circuitry(V)*. The mechanical malfunction is represented by relation *stuck(D,S)*, stating that device D is stuck in state S. For example, statement *stuck(fhb,closed)* states that switch *fhb* is stuck *closed*.

The dynamic behaviour of the basic VCM is described by a set of fluents and actions. Actions are represented as follows:

- *action_of(flip(Sw,S),R)* - flipping switch Sw to state S is an action of the subsystem R of the RCS.
- *action_of(cc(CC1,CC2),V,S),R)* - issuing a pair of computer commands $CC1$ and $CC2$ required to move valve V to state S is an action of the subsystem R of the RCS.
- *action_of(CC,R)* - issuing computer command CC is an action of the subsystem R of the RCS.

As in the plumbing module, the state of devices is described by the fluent *in_state(D,S)* meaning that device D is in state S. Furthermore, a device is always in a state S if it is stuck in S.

Normally computer commands are issued to a valve only when the switch connected to the valve is in *gpc* state. If a computer command is issued when the switch is not in gpc state, the state of the valve is undefined in the basic VCM and the input is considered abnormal. This is represented by fluent *ab_input(V)*.

The input of the basic VCM consists of:

1. a collection of statements of the form *holds(in_state(D,S),0)* describing the states of switches and valves in the initial situation;

2. the description of possible malfunctions of switches and valves;

3. the sequence of actions which defines the past history of events up to moment T.

Notice that fluents of the form *ab_input(V)* cannot be part of the description of the initial situation which is enforced by constraint:

```
:- controls(Sw,V),
   holds(ab_input(V),0).
```

The effects of actions performed on normally functioning devices are defined by two dynamic causal laws. The first law says that if flipping a properly working switch Sw to a state S causes it to move to this state. The corresponding rule looks as follows:

```
holds(in_state(Sw,S),T+1) :-
                          occurs(flip(Sw,S),T),
                          not stuck(Sw).
```

The second dynamic causal law states that, if switch Sw controlling valve V is in state *gpc*, V is working properly, and all computer commands required to move V

to some state S were issued at time T, then V will also be in state S at the next moment of time.

```
holds(in_state(V,S),T+1) :-
                    controls(Sw,V),
                    holds(in_state(Sw,gpc),T),
                    occurs(CC,T),
                    commands(CC,V,S),
                    not stuck(V),
                    not bad_circuitry(V).
```

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 above rule is applied independently of the functioning conditions of the circuit, since it is related only to the switch itself.) It is important to consider the case when a computer command is issued to control a valve and cannot be effectuated because of the current conditions. For example, 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)*. The addition of the following rule to the description allows to handle this situation.

```
holds(ab_input(V),T+1) :-
                    controls(Sw,V),
```

```
                    holds(in_state(Sw,S1,T),
                    occurs(CC,T),
                    commands(CC,V,S2),
                    state_of(S1,v_switch),
                    neq(S1,gpc),
                    neq(S1,S2),
                    not bad_circuitry(V).
```

This rule expresses that the input of valve V is abnormal at time $T+1$, i.e. the state of V is undefined in the basic level of the VCS. When fluent *ab_input(V)* is true, negation as failure is used to stop the application of the static causal law (shown below). 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.

The static connection between switches and valves is expressed by a static causal law. It says that, under normal conditions, if switch Sw controlling valve V is in some state S (open or closed[6]), at time T, then V is also in state S at time T.

```
holds(in_state(V,S),T) :-
                    controls(Sw,V),
                    holds(in_state(Sw,S),T),
                    state_of(S,v_switch),
```

[6] 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.

```
neq(S,gpc),
not holds(ab_input(V),T),
not stuck(V),
not bad_circuitry(V).
```

It is assumed that a device D is always on a state S if it stuck at S, as defined by rule

```
holds(in_state(D,S),0) :- stuck(D,S).
```

and that D is stuck if it is stuck in some state.

```
stuck(D) :- stuck(D,S).
```

Impossibility conditions are described by constraints. The VCM description includes such a constraint to express that it is not possible to move a switch to a state it is already in.

```
:- holds(in_state(Sw,S),T),
   state_of(S,v_switch),
   occurs(flip(Sw,S),T).
```

This constraint eliminates any models where an action $flip$ tries to move a switch Sw, which is in state S, to the same state S. Constraints of this type play an important role in increasing efficiency of the module by reducing the search space for plans.

Another constraint included in the basic level of the VCS specifies that a device can only be in one state at a time, as follows

```
:- of_type(D,Dev),
   state_of(S,Dev),
   holds(in_state(D,S),T),
   ¬holds(in_state(D,S),T).
```

As usual, default rules are used to represent the inertia axiom.

```
holds(in_state(D,S),T+1) :-
                          holds(in_state(D,S),T),
                          state_of(S,Dev),
                          not ¬holds(in_state(D,S),T+1).


¬holds(in_state(D,S),T) :-
                          holds(in_state(D,S1),T),
                          state_of(S,Dev),
                          state_of(S1,Dev),
                          neq(S,S1).
```

The output of the VCM is a description of the state of valves and switches at the current moment of time.

The basic VCM consists of approximately 15 rules.

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.

This module, too, defines an lp-function. It takes as input the same information as the basic VCM, together with faults on power buses, control buses and electrical circuits. The extended VCM returns 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. Large part of this module was developed independently to address a different collection of tasks [14, 15] and can be found in Chapter 3. The corresponding module used by the USA-Advisor is described in a separate section.

In the extended VCM, a switch Sw and a set of computer commands CC control a valve V via an electrical circuit C, that connects both Sw and CC to V. These connections are represented by relations *controls(Sw,V,C)* and *commands(CC,V,S,C)*. Note that each valve and each switch are connected to one circuit only. However, several valves (usually two) may be connected to the same circuit and are thus controlled by the same switch. This explains a somewhat unexpected presence of parameter C

in these relations.

The state of a valve in the extended VMC is determined by the signals present on its two input wires, 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 causal law defines this behavior.

```
holds(in_state(V,S1),T) :-
                    input(W1,V),
                    input(W2,V),
                    input_of_type(W1,S1),
                    input_of_type(W2,S2),
                    holds(value(W1,1),T),
                    holds(value(W2,0),T),
                    neq(S1,S2),
                    not stuck(V).
```

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

At this level, the representation of a switch is extended by a collection of its 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.[7] There are a few different types of switches in the RCS system. Those that control valves are called *v_switches*

[7]Note that different pairs may be associated to the same state.

and represented by relation *of_type(Sw,v_switch)*. Possible states for v_switches are expressed by relation *state_of(S,v_switch)*, and include *open*, *closed*, and *gpc*. When a switch Sw is in position (or state) S, an electrical connection is established between input Wi and output Wo of the pair(s) corresponding to S and represented in A-Prolog by statement *connects(S,Sw,Wi,Wo)*. This relation expressess that "state S of switch Sw connects input wire Wi to output wire Wo." Therefore the signal present on Wi is transferred to Wo, as expressed by the following rule.

```
holds(value(Wo,X),T) :-
                    holds(in_state(Sw,S),T),
                    connects(S,Sw,Wi,Wo),
                    holds(value(Wi,X),T).
```

Output wires Wo of all pairs corresponding to states different from S will have value 0 at time T, as defined by rule

```
holds(value(Wo,0),T) :-
                    holds(in_state(Sw,S),T),
                    connects(S1,Sw,Wi,Wo),
                    neq(S,S1).
```

We will of course also need a more detailed representation of valves. There are two types of valves in the RCS: solenoid and motor controlled valves. However, a motor controlled valve can operate in one of three ways depending on the type of electrical circuit connected to it. So, in our representation, valves can be of four types. In all cases, wires coming from an electrical circuit control the state of the valves. The present state of a valve V and the value present on its input wire connected to a power bus control the value of signals on the output wires of V.

Valves have a set of input pins, one power pin, and two output pins. They are classified according to their physical properties and to the number of input pins they have, as follows: (a) solenoid valves (which have two input pins), (b) two-pin motor-controlled (MC) valves, (c) three-pin MC valves, and (d) four-pin MC valves. The number of input pins determines the way valves are controlled. Two-pin valves have one "open" and one "closed" pin. When a signal 1 is sent to an input pin, while the other is set to 0, the valve moves to the state associated with the pin set to 1. This behavior is captured by rule

```
holds(in_state(V,S),T) :-
                    v_solenoid(V),
                    holds(value(W1,1),T),
                    input(W1,V),
                    input_of_type(W1,S),
                    neq(S,power_bus),
                    holds(value(W2,0),T),
                    input(W2,V),
                    input_of_type(W2,S1),
                    neq(S1,power_bus),
```

```
neq(S,S1),
not stuck(V,S1).
```

applicable to both solenoid and two-pin MC valves, which are identified by rules

```
v_solenoid(V) :- type_of_valve(V,solenoid).
v_solenoid(V) :- type_of_valve(V,motor2).
```

In these rules, the type, Y, of a valve, V, is given by statement *type_of_valve(V,Y)*. For instance, valve *ffha* is identified as a solenoid by statement:

type_of_valve(ffha,solenoid).

An input/output pin of a valve has a specific function associated with it. Wires connected to the input pins of valves are represented by the two relations *input(W,V)* and *input_of_type(W,Y)*, where Y is chosen in order to be able to distinguish among the different pins.[8]

Rules describing the behavior of three-pin valves are similar. Three-pin valves have one "open" pin and two "closed" (*closea* and *closeb*) pins. A three-pin valve opens if its "open" input pin is set to 1 and pin *closeb* is set to 0. The valve moves to state "closed" only when both "closed" pins are set to 1.

Four-pin valves are slightly different. They have two "open" (*opena* and *openb*) pins and two "closed" (*closea* and *closeb*) pins. In order to open the valve, only one of the "open" pins needs to be set to 1. The following two rules describe this behavior

[8]The actual naming depends on the type of valves.

```
holds(in_state(V,open),T) :-
                    type_of_valve(V,motor4),
                    input(W1,V),
                    input_of_type(W1,opena),
                    holds(value(W1,1),T),
                    not stuck(V,closed).

holds(in_state(V,open),T) :-
                    type_of_valve(V,motor4),
                    input(W1,V),
                    input_of_type(W1,openb),
                    holds(value(W1,1),T),
                    not stuck(V,closed).
```

This type of valves close with the same combination of signals as the three-pin valves: when both *closea* and *closeb* input pins are set to 1 while both input pins *opena* and *openb* are set to 0. This is described by rule

```
holds(in_state(V,closed),T) :-
                    input(W1,V),
                    input_of_type(W1,opena),
                    holds(value(W1,0),T),
                    input(W2,V),
                    input_of_type(W1,openb),
```

```
holds(value(W2,0),T),
input(W3,V),
input_of_type(W3,closea),
holds(value(W3,1),T),
input(W4,V),
input_of_type(W4,closeb),
holds(value(W4,1),T),
not stuck(V,open).
```

Power and output pins work in the same way for all types of valves. Of the two valve output pins one is labeled “open,” and the other “closed”. When a valve is in state “open,” an electrical connection is established between the power pin and the “open” output pin, while the “closed” output pin is disconnected. Wires connected to the output pins are represented by statements *output(W,V)*, which says that wire W is an output wire of valve V, and *output_of_type(W,S)*, stating that output wire W corresponds to state S. Values on output wires of both solenoid and motor controlled valves are determined by rule

```
holds(value(W,1),T) :-
        of_type(V,valve),
        holds(in_state(V,S),T),
        output(W,V),
        output_of_type(W,S),
```

```
                   input(Wp,V),
                   input_of_type(Wp,power_bus),
                   holds(value(Wp,1).
```

This rule expresses that if valve V is in state S at time T, then the value on the output wire (corresponding to S) of V is 1 at T when V is powered.

Values on ouput wires of a valve V indicate the state of V, and are therefore multually exclusive under normal behavior. If an output wire has value 1 at time T, then the value on the other output wire is 0 at T. This behavior is defined by rule

```
holds(value(W2,0),T) :-
                   of_type(V,valve),
                   output(W1,V),
                   output(W2,V),
                   neq(W1,W2),
                   holds(value(W1,1),T).
```

If a valve has no power (abnormal condition) then all its output wires have value 0, which is specified by rule

```
holds(value(W,0),T) :-
                   of_type(V,valve),
                   output(W,V),
                   input(Wp,V),
                   input_of_type(Wp,power_bus),
```

```
holds(value(Wp,0),T).
```

The behaviors described for switches and valves are valid provided that no faults are involved. If a switch is stuck in some position, flipping has no effect. If a valve is stuck in some positon, signals on the input pins are not effective. If a power or control bus is faulty, its output is constantly 0. Stuck devices are represented by *stuck(D,S)* as in the basic valve control module. Faulty power buses and control buses are described by statement *bad_device(B)*.

Given the type of a valve V, values on input wires of V at time T, malfunctioning conditions expressed by *stuck(V,S)*, and the state of V at time $T-1$, the program determines the state of V and the values present on its output wires at moment T.

The electrical circuits of the RCS are composed of both analog and digital components. Circuits are named through statements of the form *elec_circ(C)*. In the extended level of the VCM, a digital gate or component, G, can malfunction if its input/output wire W is stuck at a value X (0 or 1), defined by statement *stuck_at(W,G,X)*. If this is the case, the representation of the electrical circuit(s) these gates belong to, are also included as part of the shuttle's representation. However, it is not necessary to add the representation of circuits that are working properly. To indicate that circuit C connected to a valve V is malfunctioning we add rule

```
bad_circuitry(V) :-
                    elec_circ(C),
                    controls(Sw,V,C).
```

The behaviour of different components of electrical circuits is described within the circuit theory module.

Different power buses of both direct (DC) and alternating current (AC) provide electrical power to the RCS to allow the operation of electrical circuits, switches, valves, and other devices. These diverse sources of energy are represented by power or control buses, defined by statements *power_bus(B)* and *control_bus(B)*. Power buses generate direct current and are employed as power sources by digital devices. For example, the power pin of valves is usually connected to a power bus. Control buses generate alternate current and are used to power mechanical devices. If a bus is faulty we add statement *bad_device(B)* to the description. As before, the connection between a bus and a device (a switch or a valve) is represented by statement *output(W,B)*. Static laws express the behaviour of power/control buses as follows. If a bus B is functioning normally and W is its output wire, then the value present on W is 1 at time T. Otherwise, the value present on W is 0.

```
holds(value(W,1),T) :-
                        of_type(B,power_bus),
                        output(W,B),
                        not bad_device(B).


holds(value(W,0),T) :-
                        of_type(B,power_bus),
```

```
                output(W,B),
                bad_device(B).
```

Rules for control buses are defined in a similar way.

The space shuttle flight computer software is contained in its five general purpose computers (GPCs) which control the vehicle during specific phases of a flight. This software allows control of all RCS activity being responsible for transmiting commands for valve configuration and jet firings. If a switch is placed in GPC state, computer commands can be output to *open* or *close* the affected valves. Issuing a computer command is represented as an action that will affect a target device D by setting D to a new state. At the extended level of the VCM, issuing computer commands is expressed by a dynamic causal law that asserts value 1 on the wire W that connects the computer to a component of an electrical circuit. The rule defining this behavior is

```
holds(value(W,1),T+1) :-
                commands(CC,V,S),
                output(W,CC),
                occurs(CC,T).
```

Normally, i.e. in the absence of computer commands, a signal value 0 is assigned to the wire that connects a component of an electrical circuit to the computer, as follows

```
holds(value(W,0),T) :-
                commands(CC,V,S),
```

```
output(W,CC),

not holds(value(W,1),T).
```

Wires connected to the output pins of computer commands, as well as power buses and control buses, are identified by *output(W,E)*, where E is either a computer command, a power bus or a control bus.

The extended VCM, without the Circuit Theory module, consists of 36 rules.

5.4.3 Circuit theory module

Large portion of the Circuit Theory Module (CTM) used in the RCS was independently developed as part of the A-Circuit project, which is presented in detail in Chapter 3. Because of the modularity of our design, it has been possible to directly include the CTM in the RCS/USA-Advisor system. Some additions, however, were necessary to account for more complex circuits used in the RCS. We added the description of new electrical components that were not present in the original CTM, and more importantly the representation of stuck faults on wires of a circuit.

The Circuit Theory Module is a general description of normal and faulty behavior of components of electrical circuits with possible propagation delays and 3-valued logic. As demonstrated in Chapter 3, it can be used as a stand-alone application for simulation, computation of the maximum delay of a circuit, detection of glitches, and other tasks.

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, as shown in Figure 5.2, where gates are nodes and wires are arcs. (Note that we allow wire ($W3$) to be an input to more than one gate ($g2, g3$). We abuse the notation of graphs and represent a single wire $W3$, which is split from an electrical connection point (and hence is not a gate node) and goes to the different gates, in order to approximate the graph to a circuit schematic diagram.)

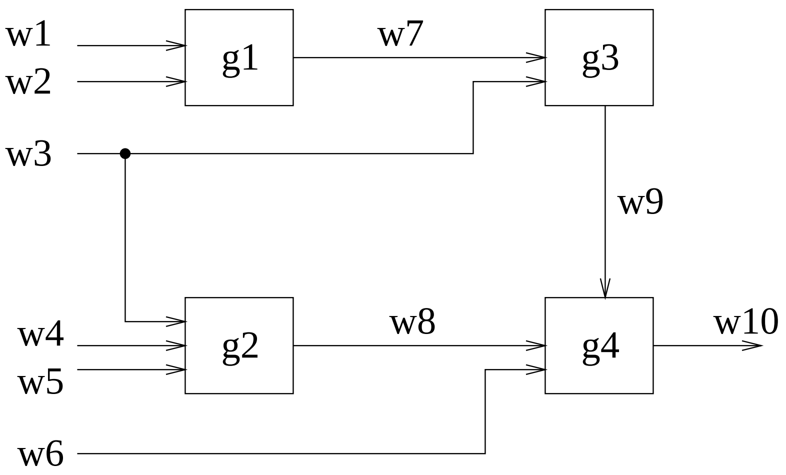


Figure 5.2: A simplified view of a circuit.

As before, 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 also expressed in 3-valued logic (0, 1, u). These signal values will be applied to input wires and propagated through the gates. Recall that if no definite value (0,

1) is present on a wire at a certain moment of time T then the value is said to be *undefined* (at T) and denoted by u.

The language $\mathcal{L}_{ckt}$ for describing electrical circuits in this module have names for gates ($g1$, $g2$, ...), wires ($w1$, $w2$, ...), signals (0, 1, u), as before, but the original gate types (*and_gate*, *or_gate*, *not_gate*,) have been expanded with: *tri_state_gate*, *td1_gate*, *niland_gate*, *rpc_gate*, *connector*, and names for wire types were also introduced: (*enable*, and *neglog*).

A *tri_state_gate* type corresponds to a *Tri-State* component, a *td1_gate* type corresponds to a *Time Delay* gate, a *Negated Input Logic AND gate* is named *niland_gate*, and a *Remote Power Controller* gate as *rpc_gate*. For uniformity of representation we also specify the points where two (converging) wires are electrically connected as a "pseudo-gate" named *connector*. This pseudo-gate behaves similar to an OR gate. This electrical connection is used when two or more wires must be connected together and become a single wire. This addition was necessary to accomodate electrical connections present in the RCS circuits, and it facilitated somewhat the translation of a circuit obtained from the graphical interface to A-Prolog.

A Tri-State gate behaves as an electrical switch which when turned on (or "enabled") allows the value on its input wire to be propagated to the output wire; while if it is not turned on the value on its output is undefined. The *enable* input wire of a Tri-State gate "enables" or turns on the component when it holds value 1. The Negated Input Logic AND gate exhibits the behavior of an AND gate whose *neglog* (negated logic) input wire is connected to an inversor, a NOT gate. The Time Delay gate, *td1_gate*, propagates the signal on its input wires at a certain time T only after a delay of 1 second. The behavior of a Remote Power Controller is similar to an AND gate.

The behavior of the new most interesting gates, in the presence of signal u, is presented in Tables 5.1(a), 5.1(b), and 5.2.

Inputs		*Output*
enable	X	
0	0	u
0	1	u
0	u	u
u	0	u
u	1	u
u	u	u
1	0	0
1	1	1
1	u	u

Inputs		*Output*
neglog	X	
0	0	0
0	1	1
0	u	u
u	0	0
u	1	u
u	u	u
1	0	0
1	1	0
1	u	0

Table 5.1: (a) Tri-State gate. (b) Negated Input Logic AND gate.

Input *Time = t*	*Output* *Time = t+1*
0	0
u	u
1	1

Table 5.2: Definition of the behavior of a Time Delay (of 1 sec) gate.

As before, circuits are named by statements of the form *elec_circ(C)*. Relations *of_type(G,GT)* and *type_of_wire(W,G,WT)* express that a gate G [wire W] is of type

GT [WT], while relation *delay(G,D)* says that delay D is associated to gate G. In order to represent types of wires we now reify wires with statement *is_wire(W)*.

The geometry of the circuit (connection among gates), is described the same way as in the A-Circuit system by representing the input and output wires of each of its gates. However, there is a slight change in the relations used. To connect the output of a gate $G1$ to an input of a gate $G2$ by a wire W, we simply indicate that wire W is the output wire of gate $G1$, *output(W,G1)*, but to specify that wire W is the input wire of gate $G2$ we now use statement *is_input(W,G2)*. The change was prompted by the introduction of faults on wires and the desire to use the original circuit theory to describe the normal behavior of gates.

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.

The dynamic behaviour of the CTM is described by fluent *value(W,X)* which expresses that the value present on wire W is X, and action *apply(W,X)*, which says that signal value X is applied to wire W. An observation of the form *occurs(apply(W,X),T)* states that action *apply(W,X)* occurred at (the situation corresponding to) moment of time T. The effect of applying a signal value to an input wire is expressed by the following dynamic causal law

```
holds(value(W,X),T) :-
                        is_input(W,G),
                        occurs(apply(W,X),T).
```

We allow for standard faults from the theory of digital circuits [105]. A gate G malfunctions if its output, or at least one of its input pins, are permanently stuck on a signal value. This is expressed by relation *stuck_at(W,G,X)* read as wire W of gate G is stuck at value X (0 or 1). 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 a gate previously discussed, the NOT gate. The rule that defines its normal behavior differs from the one previously shown in Chapter 3 only by the inclusion of condition *not is_stuck(W2,G)*, and is written as follows

```
holds(value(W2,S2),T+D) :-
                        of_type(G,not_gate),
                        delay(G,D),
                        input(W1,G),
                        output(W2,G),
                        opposite(S1,S2),
                        holds(value(W1,S1),T),
```

```
not is_stuck(W2,G).
```

This rule says that if value $S1$ holds at the input wire $W1$ of a NOT gate, with propagation delay D, at time T then the opposite value $S2$ will hold in its output wire $W2$ at moment of time $T+D$ if $W2$ is not stuck at some other value.

Let us now consider a new component: the Tri-State gate, whose behavior is defined by Table 5.1(a). 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 with delay D 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.

```
holds(value(W,X),T+D) :-
                    of_type(G,tri_state_gate),
                    delay(G,D),
                    input(W1,G),
                    input(W2,G),
                    type_of_wire(W2,G,enable),
                    neq(W1,W2),
                    holds(value(W1,X),T),
                    holds(value(W2,1),T),
                    output(W,G),
                    not is_stuck(W,G).
```

The rule defining the case when the enable wire of the Tri-State gate is not set to 1 is written as follows.

```
holds(value(W,u),T+D) :-
                    of_type(G,tri_state_gate),
                    delay(G,D),
                    input(W1,G),
                    input(W2,G),
                    type_of_wire(W2,G,enable),
                    neq(W1,W2),
                    holds(value(W1,X),T),
                    holds(value(W2,Y),T),
                    neq(Y,1),
                    output(W,G),
                    not is_stuck(W,G).
```

Still another new component is the Time Delay gate which propagates a signal present on its input wires at a certain time T only after a delay of 1 second, as shown in Table 5.2. Since we allow delays in our representation the definition of this rule is straigthforward.

```
holds(value(W,X),T+1) :-
                    of_type(G,td1_gate),
                    input(W1,G),
                    output(W,G),
```

```
        h(value(W1,X),T),
        not is_stuck(W,G).
```

Notice that condition *not is_stuck(W,G)* prevents the above rules, describing the normal behavior of some gates, from being applied when the output wire is stuck. What is not apparent is how the normal condition of the input wires is guaranteed before the application of the rule. This is partially hidden by our choice of predicates to describe inputs, and will be discussed next.

First, let us examine how input wires are now represented. Recall that we describe the input wires of a gate by relation *is_input*(W,G), which is automatically generated by the translation from the graphical representation of the circuit to A-Prolog. Under normal conditions, an input wire is not stuck at any value. We define this normal input as follows

```
input(W,G) :-
        is_input(W,G),
        not is_stuck(W,G)
```

We determine that an input wire W of a gate G is stuck if it is stuck at some value X, as follows

```
is_stuck(W,G) :- stuck_at(W,G,X).
```

Now we need to understand 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. Figure 5.3 gives a graphical representation of this idea.

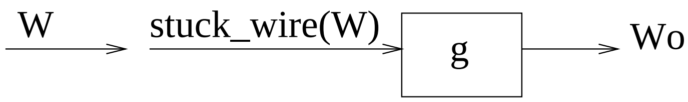


Figure 5.3: A graphical representation of a faulty input wire.

The faulty part of the wire, *stuck_wire(W)*, is stuck at value X, while the value of the normal part W is computed by normal rules depending upon its connection to the output of other gates. Thus, if the input W of a gate G is stuck at some value, then we have a "faulty wire" which is defined by rule

```
input(stuck_wire(W)) :- is_stuck(W,G).
```

The value on the bad connection side of the wire is expressed by rule

```
holds(value(stuck_wire(W),X),0) :-
                    is_input(W,G),
                    stuck_at(W,G,X).
```

So, rules for gates with faulty inputs use $stuck\_wire(W)$ as input wire. We show below an example of how this representation is used to specify a Tri-State gate with the non-enable wire stuck to X.

```
holds(value(W,X),T+D) :-
```

```
                    of_type(G,tri_state_gate),
                    delay(G,D),
                    is_input(W1,G),
                    input(W2,G),
                    neq(W1,W2),
                    holds(value(stuck_wire(W1),X),T),
                    type_of_wire(W2,G,enable),
                    holds(value(W2,1),T),
                    output(W,G),
                    not is_stuck(W,G).
```

This rule says that if a Tri-State gate G, with propagation delay D, is enabled at time T, while its other input wire $W1$ is stuck at value X at this time, then the value on the output wire W of G is X at time $T+D$. We also need some other rules to complete the representation of an enabled Tri-State gate under faulty conditions. These rules are

```
holds(value(W,X),T+D) :-
                    of_type(G,tri_state_gate),
                    delay(G,D),
                    input(W1,G),
                    is_input(W2,G),
                    neq(W1,W2),
```

```
                    h(value(W1,X),T),
                    type_of_wire(W2,G,enable),
                    h(value(stuck_wire(W2),1),T),
                    output(W,G),
                    not is_stuck(W,G).


holds(value(W,X),T1) :-
                    of_type(G,tri_state_gate),
                    delay(G,D),
                    is_input(W1,G),
                    is_input(W2,G),
                    neq(W1,W2),
                    holds(value(stuck_wire(W1),X),T),
                    type_of_wire(W2,G,enable),
                    holds(value(stuck_wire(W2),1),T),
                    output(W,G),
                    not is_stuck(W,G).
```

Lastly, when the Tri-State gate is not enabled under faulty conditions the value of its output wire is undefined.

```
holds(value(W,u),T+D) :-
```

```
                        of_type(G,tri_state_gate),
                        delay(G,D),
                        is_input(W2,G),
                        type_of_wire(W2,G,enable),
                        ¬holds(value(stuck_wire(W2),1),T),
                        output(W,G),
                        not is_stuck(W,G).
```

We show next the rule defining the behavior of a NOT gate with its input wire stuck at a certain value.

```
holds(value(W2,S2),T+D) :-
                        of_type(G,not_gate),
                        delay(G,D),
                        is_input(W1,G),
                        holds(value(stuck_wire(W1),S1),T),
                        opposite(S1,S2),
                        output(W2,G),
                        not is_stuck(W2,G).
```

It says that if the input value of a NOT gate is $S1$ at time T ,and its delay is D, then the value on its output wire is $S2$, the opposite of $S1$, at time $T+D$.

Faults on output wires are treated differently because the faults are propagated forward, i.e. an output wire will be represented by a normal and a faulty section only if this wire is an input of another gate. Since the set of faults is provided for the initial situation, we say that if the output W of a gate G is stuck at value X, then the value on W is X at the initial moment of time. This case is represented by rule

```
holds(value(W,X),0) :-
                        output(W,G),
                        stuck_at(W,G,X).
```

The inertia law is written in the form of a default as follows

```
holds(value(W,X),T+1) :-
                        signal(X),
                        is_wire(W),
                        holds(value(W,X),T),
                        not ¬holds(value(W,X),T+1).
```

It says that if the value on wire W at time T is X, and there is no reason to believe that the value on W will change at time $T+1$, then the value on W will remain X at time $T+1$. To express that there is at most one signal present on a wire at certain moment of time, we also add rule

```
¬holds(value(W,X),T) :-
                        signal(X),
                        signal(Y),
                        neq(X,Y),
                        is_wire(W),
```

```
h(value(W,Y),T).
```

The behavior of a circuit is said to be *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.

5.4.4 Planning module

As explained in Section 5.4, the USA-Advisor allows flight controllers to perform two types of tasks related to planning in the RCS domain. It

- determines whether a plan manually devised by the controllers achieves a goal; and
- finds a plan, of a length not exceeding some number of steps, N, to achieve a goal.

The Planning 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 the USA-Advisor allows for the creation of a variety of such modules.

For simplicity of presentation we start our discussion with the basic planning module (Section 5.5). It will be used to illustrate the idea of answer set planning. Section 5.6 contains an elaboration of this idea and serves as a practical planning module of the system.

5.5 The Basic Planner

The Basic Planning Module of the USA-Advisor establishes a simple search criteria used by the program to find a plan. The structure of the Basic Planning Module described in this section follows the generate and test approach from [48, 120]. The main idea of this approach consists of establishing one-to-one correspondence between plans for achieving a goal G and answer sets of a logic program P_G. This program normally consists of (a) a large part describing our knowledge about the corresponding dynamic system, and (b) a smaller part containing specification of a goal, a special rule "generating" actions, and possibly some other rules describing properties of the desired plans. The following discussion illustrates this idea. Our approach differs from the standard answer set planning approach by taking 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.

The following rules form the heart of the planner. The first rule, which is responsible for the generation of actions, states that, for each time point, T, in a given finite interval, if the goal has not been reached for one of the RCS subsystems, then an action controlling that subsystem should occur at that time.

```
1{occurs(A,T):action_of(A,R)}1 :-
                              T < lasttime,
                              subsystem(R),
                              not goal(T,R).
```

A rule of this form is called a "choice rule," and is part of the language of SMODELS [155]. It is proved that choice rules do not extend the expressive power of the language and can therefore be viewed as a shorthand for a set of logic programming standard rules. The rules, however, proved to be very convenient. First, they substantially shorten the program. Even more importantly, they allow efficient implementation which to a large degree is responsible for the efficiency of our planner.

Notice that the head of the choice rule has the form

$$L\{p(\bar{X}) : q(\bar{X})\}U.$$

It defines a subset $p \subseteq q$ of terms such that $L \leq |p| \leq U$. Normally, there are many possible sets satisfying these conditions. Hence, a program containing this type of rules has multiple answer sets, corresponding to possible choices of p.

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. The first rule below states that the overall goal has been reached if, for each subsystem, there is a time at which the goal has been reached for the subsystem.

```
goal :-
        goal(T1,left_rcs),
        goal(T2,right_rcs),
        goal(T3,fwd_rcs).


:- not goal.
```

The second rule above is a constraint that states that for a model (a solution) to exist, the overall goal must be achieved. The plan testing phase of the search is implemented by this constraint which eliminates the models that do not contain plans for the goal.

Splitting the RCS into subsystems allows us to improve the efficiency of the module substantially. For instance, for some goal, finding a plan of 5 steps takes a few seconds, as opposed to a few hours required when the representation of the RCS is not partitioned into subsystems. Notice that, since there are some dependencies between some subsystems, a very small number of extremely rare (and undesirable) plans can be missed. It's possible to modify the Planning module in order to find these plans too.

One such dependency is the connection between the Left and Right RCS' subsystems through two *crossfeed* lines (one for fuel and one for propellant) controlled by crossfeed valves. Each of the subsystems has one such valve per line. When a fault in the fuel line of one of these subsystems, say the Left RCS, does not permit preparing one of its jets for firing, the crossfeed valves are used to direct fuel from the other subsystem, in this case the Right RCS, to supply what is needed. The actions required for such operation cannot be generated for certain types of maneuvers, as we explain shortly.

First, recall from Section 5.4.1 that in our current partitioned representation of the RCS, whenever a maneuver M does not require firing any jets from one of the subsystems, say R, we specify that the maneuver portion corresponding to R is ready, by relation *done(M,R)*. In this case, during planning, no actions are generated for the subsystem which is ready, and its crossfeed valve stays closed blocking the fuel line between the subsystems. As a result, no plan using the crossfeed will be found, in this case.

The situation described above is rather rare. There are 12 possible maneuvers in the RCS domain. The partitioned representation of the RCS requires 36 rules to express these maneuvers. From these, only four can be (in extreme circumstances) affected by this limitation. The maneuvers and corresponding subsystems connected by the crossfeed lines described as “ready” in our representation are:

- $+Y$ - Right RCS,
- $-Y$ - Left RCS,

- $+yaw$ - Left RCS,
- $-yaw$ - Right RCS.

Of course if every available plan for achieving a given maneuver uses the crossfeed our system will return a misleading “no plan” answer. We dealt with the problem by following each suspected failure by an extra run of a slightly modified version of the planner. Recently a new and more elegant solution to this problem was found which is based on the extension of A-Prolog from [13].

Since the RCS contains more than 200 actions, with rather complex effects, and may require long plans, the standard planning approach described above needs to be substantially improved. This is done by addition of various forms of heuristic, domain-dependent[9], information. We refer to the Basic Planner expanded by such heuristics as Smart Planner.

5.6 Smart Planner: adding the control knowledge

In this section we discuss the expansion of the basic planner by useful heuristic information, including control knowledge. The usefulness of control knowledge for planning has been investigated in [9, 11, 99, 104], but comparatively little is known about the influence of heuristics in answer set planning (see however [27]). Such knowledge can be classified into two categories: domain dependent and domain independent.

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

Both types of heuristics work by either limiting the combinations of actions that can occur or by declaring that certain situations are illegal. In either case the heuristics help prune the search space, leading to increased efficiency, and improving plan quality by eliminating undesired plans.

Some of the control knowledge used in the USA-Advisor can easily be included for planning in other domains. An example of such domain independent knowledge is the statement "Do not repeat actions already performed." Note that, while this rule does not apply in all domains, in many an optimal plan will never include the same action twice. This rule can be easily encoded in A-Prolog as the following constraint:

```
:- action_of(A,R),
   not equal(T1,T2),
   occurs(A,T1),
   occurs(A,T2).
```

Next consider the following statement: "Do not perform two different types of actions which achieve the same effect." While the general idea expressed in this statement is similar to the one above, the encoding is quite different – it is domain dependent.

```
:- controls(Sw,V),
   occurs(flip(Sw,P),T),
   commands(CC,V,P),
   occurs(CC,T1).
```

Given a switch Sw that controls a valve V, this constraint eliminates any models where an action flips Sw to position P is later followed by the issuing of a computer command also seeking to move V to P.

The different encoding is due to the fact that in the RCS domain, the only actions which have the same effect are those of using either a switch or a computer command to change the position of a valve. In this case it is much easier to encode the domain specific instance of the general rule than to write the general rule itself. However we found that the understanding of the general nature of this heuristic is indispensible for the system designer.

There are a number of domain specific heuristics in the USA-Advisor. The following example states that a switch should not be moved to the gpc (general purpose computer) position unless the next action is to issue a computer command to the valve related to that switch.

```
:- controls(Sw,V),
   occurs(flip(Sw,gpc),T),
   not issued_commands(V,T+1).
```

Note that while there are valid plans for the operation of the RCS which do not obey this rule, for each of them there is a plan containing exactly the same actions which does obey it. This allows us to further prune the search space.

The next constraint does not directly address the performance of an action. It states that, unless a valve is stuck, it is not allowed to be open if there is no pressure above it.

```
:- link(N1,N2,V),
   holds(in_state(V,open),T),
   not holds(pressurized(N1),T),
   not stuck(V),
   not holds(in_state(V,open),0).
```

This constraint is not a physical requirement but rather a preference on types of plans.

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. Although we do not claim the plans computed are optimal, the shortest sequence of actions to achieve the goal is a good candidate as the optimal plan(s). The following constraint eliminates models where more than one path to pressurize a node $N2$ is open.

```
:- link(N1,N2,V1),
   link(N1,N2,V2),
   neq(V1,V2),
   holds(in_state(V1,open),T),
   holds(in_state(V2,open),T),
   not stuck(V1,open),
```

```
    not stuck(V2,open).
```

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.

```
:- link(N1,N2,V),
    holds(in_state(V,open),T),
    not holds(pressurized_by(N1,Tk),T),
    not has_leak(V),
    not stuck(V).
```

The improvement offered by domain-dependent heuristics has not been studied mathematically here. However, our experiments show that some of the domain-dependent heuristics play a crucial role on the efficiency of the planning module. The impact of such heuristics was made clear when the time required to find a plan for tasks involving a large number of faults was reduced from hours to seconds.

The Planning Module contains approximately 35 rules of which 13 are heuristics.

The planner is by far the largest and most sophisticated answer set planner in existence. In fact we are not aware of any other successful declarative and/or otherwise provenly correct planner of this size. Below are some lessons we learned from its design and implementation.

- Since a single action of an astronaut changes the values of many interrelated fluents of the RCS the description of effects of this action becomes a nontrivial task. To solve it we need to find solutions to frame, ramification, and qualification problems [138, 66, 135]. We solved these problems by using the techniques developed in theory of action and change and the power of A-Prolog rules. The frame problem was solved by encoding the inertia axiom by a "nonmonotonic," default rule of A-Prolog. Qualification was addressed by the use of constraints. And finally, the most difficult ramification problem was solved by the use of static causal laws. It is not clear to us how and if the effects of the RCS actions could be accurately represented by more traditional STRIPS [65] like action languages like ADL [161].
- A-Prolog proved to be a language capable of specifying the initial situation, causal and other relations of the domain, as well as the heuristic information limiting the search space and improving quality of plans. This contrasts with some of the other representational approaches which require separate languages for each of these classes of statements. For instance, the encoding of heuristic information in [9, 10, 11] required a fairly sophisticated use of temporal logic.
- Domain models written in A-Prolog can also be used for tasks different from planning. We have seen one such example in Section 3.5.2. Example of their use for diagnostic purposes can be found in [12, 82]. This is done by simply

replacing the planning module with an appropriate (e.g. diagnostic) module in which the agent's actions are replaced by exogenous actions of the environment. In a sense answer set diagnostics can be viewed as "planning in the past".

- The heuristics used in the Smart Planner were easy to encode and to use. Moreover, our experiments show that they significantly improve both, quality of plans and efficiency of search.
- It was interesting to notice that many fluents of the RCS domain had natural recursive definitions, easily expressible in A-Prolog. Recursive definition however precluded the immediate use of CCALC [132] and other planners which use satisfiability solvers. It will be interesting to see if such solvers could be used after some modifications of the representation. It is probably also worth mentioning that nonmonotonicity of A-Prolog played an important role in the formalization of the domain, e.g. in specifying the inertia axiom, closed world assumptions used for describing the initial situation, and other typical default knowledge.
- The planner's ability to mix parallel and sequential plans and to efficiently search for them are the key ingredients in the success of the project.

Overall, answer set planning proved to be a good tool for our purpose. We are not aware of any other tool which would allow us to deal with complexity of effects of the RCS actions. The next section shows that the resulting system is remarkably efficient. Partly this is due to non-numerical nature of the problem. The fact that despite a large number of concurrent actions involved, the plans were comparatively short also contributed to the efficency. To expand the applicability of answer set planning and reasoning to hybrid systems, i.e. systems involving "continuous" time and numerical computations we need to substantially extend existing answer set solvers.

The complete program describing the structure and behavior of the RCS contains approximately 700 facts and 175 rules.

5.7 Experimental Results for the USA-Advisor

In this section we give an overview of our experiments with the smarter planner of the USA-Advisor. We used a 2.4GHz Pentium 4 computer with 1024MB of RAM, running the NetBSD 1.6 Operating System; SMODELS version 2.26 with input from *Lparse* version 1.0.9, and *MKAtoms*[10] version 2.1, were used to find the plans.

The number of actions contained in a plan P for an individual subsystem of the RCS R is called the *number of steps* of P (since we assume that each action takes one unit of time (or step) to be performed), and is denoted $steps(R)$. The total *number of steps* of a plan for the whole RCS is the maximum among the number of steps taken by each RCS subsystem, i.e. $N = max(steps(Forward), steps(Left), steps(Right))$. In order to allow the grounding of the program by *Lparse*, it is necessary to include the number of steps N of the plan in the call to SMODELS.

[10] *MKAtoms* is a utility that re-formats the output of SMODELS and DLV, in order to have only one atom per line. It was developed by Marcello Balduccini and is available for download from http://krlab.cs.ttu.edu/ marcy/mkatoms

The RCS can be tested on two levels of detail: basic and extended level. There are two types of tasks to be tested: checking a plan, and finding a plan. To perform these tasks, besides the modules already discussed, we need a set of rules describing the initial state of tanks, switches, and valves, called *initial situation*; and a *test instance*, a collection of system faults together with a maneuver to be performed by the shuttle. The initial situation is common to all test instances, and is shown on Figure 5.4 . Figure 5.5 shows the test instance for maneuver $-Z$ with the RCS system malfunctioning with 3 mechanical and 2 electrical faults.

The format of a call to SMODELS is determined by the level of representation and task to be performed, as follows:

1. The basic level representation does not involve electrical faults, i.e. neither the Extended Valve Control Module, the Circuit Theory Module, nor any of the descriptions of circuits of the RCS are used.

   a. Planning in this case requires a call to SMODELS of the form:

   ```
   lparse -c lasttime=N -d none rcs_basic planner
          initial_situation test_instance_XXX | smodels
   ```

   where file `rcs_basic` corresponds to the Basic Valve Control Module, and `planner` is the Smart Planner Module. *Lparse* parameter *-c lasttime=N* gives the maximum number of steps to be considered for a plan; parameter *-d none* provides an optimization that reduces the ground program by removing literals which are trivially true. (For details on options to *Lparse* refer to [189].) Since no parameters are specified for SMODELS, it will search and return a single plan (the first plan found) satisfying the goal. SMODELS would compute and return all plans found if the call included parameter "0", written as

```
lparse -c lasttime=N -d none rcs_basic planner
       initial_situation test_instance_XXX | smodels 0
```

b. Checking a plan requires a slightly different call to SMODELS. The plan to be checked is written in the form of constraints in file *plan_XXX*. SMODELS can then be called with command

```
lparse -c lasttime=N -d none rcs_basic planner
       initial_situation test_instance_XXX plan_XXX | smodels
```

2. The extended level representation is used if the problem involves both electrical and mechanical faults. To improve efficiency, the only circuit descriptions included in the call are those of faulty circuits.

   a. For planning in this case the call (corresponding to the test instance of Figures 5.4, and 5.5) has the form

```
lparse -c lasttime=N -d none rcs_basic rcs_extended
       circuit_theory fmc2 fmc4 planner
       initial_situation test_instance_XXX | smodels
```

where `rcs_extended` is the Extended Valve Control Module; `circuit_theory` is the Circuit Theory Module; and `fmc2`, `fmc4` are the descriptions of electrical circuits *fmc2* and *fmc4*, respectively.

b. For checking a plan, we add file *plan_XXX* containing the plan written in the form of constraints. The call has the form

```
lparse -c lasttime=N -d none rcs_basic rcs_extended
        circuit_theory fmc2 fmc4 planner
        initial_situation test_instance_XXX plan_XXX | smodels
```

Section "Common Part" of the initial situation shown in Figure 5.4 defines the state of tanks, switches and valves initially. It is assumed that all helium tanks are pressurized in the initial state, which is written as facts. The normal condition of switches and valves is described by default rules.

The first four lines of the section "Faults and Other Exceptions" of the test instance shown in Figure 5.5 refer to the three mechanical faults affecting the RCS. Switch *fm1* of the Forward RCS is stuck open, while two faults affect the Right RCS: valve *roi345b* is leaking while open, and switch *ri12* is stuck open. (Note that leaking valves which are closed do not really constitute a fault.) The last two lines indicate electrical faults. The first statement says that wire $w6$ of gate $g4$ of circuit *fmc4* is stuck at 0; the second fault is that wire $w28$ of gate $g8$ of circuit *fmc2* is stuck to 1. Section "Goals" contains the subgoals to be achieved by each RCS subsystem in order to prepare the shuttle for maneuver $-Z$.

The solution to the test instance from Figure 5.5 is shown in Figure 5.6. A plan with 4 steps and 12 actions was found in 2.44 seconds for the SMODELS call corresponding to *lasttime* $= 4$. In principle, it is not known how many steps a plan will have, therefore several calls may be necessary before a plan is found. In our tests, we consider only plans containing 3 or more steps, since 1 and 2-step plans can be easily obtained, even manually. However, the USA-Advisor can also be used for computing or checking these simple plans. For this example, a previous call with *lasttime* $= 3$ returned *false* in 0.57 seconds, indicating that no plan of 3 steps existed for this problem. Hence, the total time for computing a plan for this test instance was 3.01 seconds.

During the implementation of the Planner Module we conducted the following series of experiments in order to compare the performance of the basic and the smart planner:

(a) randomly generated a collection of test instances with a given number of mechanical and electrical faults;

(b) ran the basic and the smart planners in a loop with *lasttime* ranging from 3 to 10. The duration of each iteration of the loop was limited to 10 minutes.

Overall, about 500 test instances were generated in this manner, and included three

```
%%%%%%%%%%%    INITIAL SITUATION    %%%%%%%%%%%

%%%%%%%%%%%       COMMON PART       %%%%%%%%%%%


% Initially, the Helium tanks are pressurized.

holds(pressurized_by(ffh,ffh),0).

holds(pressurized_by(foh,foh),0).

holds(pressurized_by(lfh,lfh),0).

holds(pressurized_by(loh,loh),0).

holds(pressurized_by(rfh,rfh),0).

holds(pressurized_by(roh,roh),0).


% All switches are normally in state GPC initially.

holds(in_state(Sw,gpc),0) :- of_type(Sw,v_switch),
                             not ¬holds(in_state(Sw,gpc),0).


% Valves are all normally closed initially.

holds(in_state(V,closed),0) :- of_type(V,valve),
                               not ¬holds(in_state(V,open),0).
```

Figure 5.4: Initial situation common to all test instances of RCS planner.

```
%%%%%%%%%%%         INITIAL SITUATION        %%%%%%%%%%%

%%%%%%%%%%%   FAULTS and OTHER EXCEPTIONS    %%%%%%%%%%%

stuck(fm1,open).

has_leak(roi345b).

h(in_state(roi345b,open),0).

stuck(ri12,open).

stuck_at(fmc4_w6,fmc4_g4,0).

stuck_at(fmc2_w28,fmc2_g8,1).


%%%%%%%%%%%%%%%            GOALS          %%%%%%%%%%%%%%

% Maneuver to be performed: plus_z

goal(T,fwd_rcs)   :- time(T),
                     h(maneuver_of(minus_z,fwd_rcs),T).


goal(T,left_rcs)  :- time(T),
                     h(maneuver_of(minus_z,left_rcs),T).


goal(T,right_rcs) :- time(T),
                     h(maneuver_of(minus_z,right_rcs),T).
```

Figure 5.5: Test instance for RCS planner with 3 mechanical and 2 electrical faults.

```
smodels version 2.26. Reading...done

Answer: 1

Stable Model:
             occurs(flip(fha,open),0)
             occurs(flip(ri345b,closed),0)
             occurs(cc(closea_li12,closeb_lfi12),0)

             occurs(flip(fi345,open),1)
             occurs(opena_rfhb,1)
             occurs(opena_lfha,1)

             occurs(flip(fm3,open),2)
             occurs(flip(li345b,open),2)
             occurs(flip(ri345a,open),2)

             occurs(flip(fm4,open),3)
             occurs(flip(lm4,open),3)
             occurs(flip(rm4,open),3)

True
Duration: 2.440
Number of choice points: 20
Number of wrong choices: 0
Number of atoms: 28221
Number of rules: 80256
Number of picked atoms: 26060
Number of forced atoms: 601
Number of truth assignments: 2130760
Size of searchspace (removed): 593 (790)
Total: 3.010
```

Figure 5.6: Solution for test instance shown in Figure 5.5.

```
%%%%%%%%%%%    PLAN TO BE CHECKED    %%%%%%%%%%%

:- not occurs(flip(fha,open),0).

:- not occurs(flip(ri345b,closed),0).

:- not occurs(cc(closea_li12,closeb_lfi12),0).

:- not occurs(flip(fi345,open),1).

:- not occurs(opena_rfhb,1).

:- not occurs(opena_lfha,1).

:- not occurs(flip(fm3,open),2).

:- not occurs(flip(li345b,open),2).

:- not occurs(flip(ri345a,open),2).

:- not occurs(flip(fm4,open),3).

:- not occurs(flip(lm4,open),3).

:- not occurs(flip(rm4,open),3).
```

Figure 5.7: Plan file corresponding to test instance shown in Figure 5.5.

mechanical and two electrical faults - the most interesting situation from the standpoint of the USA experts. The Smart Planner was able to find the plans or discover their absence in less than 22 seconds. The Basic Planner required substantially more time (in some cases the difference exceeded 2 orders of magnitude). On average the Smart Planner was about 10 times faster.

The second series of experiments dealt with our deliberate attempt to crash the system. We selected a number of test instances which seemed to correspond to especially difficult situations. Even though the size of the grounded program, the length of plans, and the number of actions involved are substantially larger than those in the initial experiments, the time is still quite acceptable (USA wanted planning times of less than 15 minutes). In contrast, the basic planner was not able to find solutions to any of these problems - we stopped the planner after 24 hours of work.

It is interesting to note that achieving this performance required all of the Smart Planner heuristics - removal of some of them gave a small improvements on a few test instances, but on others tests the performance was worsened by more than an order of magnitude. A Pentium II 450MHz system was used in these initial trials. More detailed results on these experiments appear in [158, 18, 157].

To further test the Smart Planner we conducted a series of experiments based on the random generation of 2000 test instances, distributed in blocks of 200 instances, containing the following number of faults:

- Block 1: 3 mechanical and 0 electrical;

- Block 2: 3 mechanical and 2 electrical;
- Block 3: 5 mechanical and 0 electrical;
- Block 4: 5 mechanical and 3 electrical;
- Block 5: 8 mechanical and 0 electrical;
- Block 6: 8 mechanical and 5 electrical;
- Block 7: 10 mechanical and 0 electrical;
- Block 8: 10 mechanical and 3 electrical;
- Block 9: 10 mechanical and 5 electrical;
- Block 10: 10 mechanical and 7 electrical.

The tests performed with these instances used the smart planner in a loop with *lasttime* ranging from 3 to 10; and as before, the duration of each iteration of the loop was limited to 10 minutes. Our choice of 10 minutes is guided by the expectation of flight controllers to have a result in less than 15 minutes. The number of steps and the time limit can always be increased, however it becomes increasingly harder to find plans for instances with such high number of faults.

The overall results for the 2000 experiments are summarized in Table 5.3. Here the name of an instance group indicates the number of mechanical and electrical faults in that block of experiments, e.g. *ins-10-7* means that all 200 test instances in this block have 10 mechanical and 7 electrical faults. The first column of Table 5.3 indicates the different instance groups tested; the second column gives the maximum number of actions performed, and the third indicates the maximum number of steps needed, for all plans found in a specific instance group. The maximum time, in seconds, to find a plan with N steps, without considering previous unsuccessful computations with $3 \leq$ *lasttime* $< N$, is given in the fourth column. The maximum total time, in seconds, to find a plan with N steps, which includes the time required for previous unsuccessful computations with $3 \leq$ *lasttime* $< N$, is presented in the fifth column. This correspond to the worst-case scenario. Notice that in these experiments, few difficult test instances required several minutes to compute a plan, or to indicate a plan did not exist, while the majority of the test instances was solved in seconds. The values in the sixth column confirm this obsertation. It gives the maximum average total time, in seconds, to find a plan with N steps, which includes the time required for previous unsuccessful computations with $3 \leq$ *lasttime* $< N$. The last column shows the number of test instances, per block of experiments, for which no plan was found. It is important to point out that for all these instances the planner indicated the absence of a plan, i.e. the planner was able to conclude that no plan exists in the time allowed for the computation.

Some other important information regarding these experiments are: (a) the number of ground rules in the tests ranges from 50,000 to 285,000 with an average of 160,000 rules, and (b) the number of ground atoms ranges from 15,000 to 70,000 with an

Instance groups	Max-#actions	Max-#steps	Max-time (seconds)	Max-total time (secs)	Avg-total time (secs)	#no-plan found
ins-3-0	18	6	17.020	608.300	4.459	7
ins-3-2	15	5	5.560	19.170	3.760	60
ins-5-0	15	7	75.810	687.320	5.930	30
ins-5-3	18	7	35.720	753.460	16.618	103
ins-8-0	18	6	79.270	610.770	11.034	69
ins-8-5	16	6	13.130	114.590	8.460	140
ins-10-0	18	7	465.420	1213.000	20.478	99
ins-10-3	18	6	41.750	615.280	16.856	147
ins-10-5	12	6	8.790	105.560	9.447	163
ins-10-7	12	4	108.100	108.680	10.439	181

Table 5.3: Overall results for 2000 RCS experiments.

average of 34,000 atoms. Graphs and tables with detailed information about the test instances used in these experiments are presented in Appendix A.

Finally, we highlight the fact that, on average, the maximum total time required to find a plan, for all test instances for which such a plan existed, was less than 21 seconds.

5.8 Summary

In this chapter we described a medium size decision support system written in A-Prolog. This application requires modeling of the operation of a fairly complex sub-system of the space shuttle at a level suitable for use by shuttle flight controllers. It is expected that deployment of this system, for use in the space program, will begin in August of 2003. The system, while based on a representation of the Reaction Control System described on previous work [203, 31], represents a substantial advance over its predecessor (which was developed in Prolog.) The RCS/USA-Advisor is implemented in the declarative language A-Prolog and uses methodologies and search engines based on a new programming paradigm, *answer set programming.*

From the scientific standpoint, this work can be of interest to two groups of people, those interested in answer set programming and those interested in planning. We hope both groups will be glad to learn about the existence of a comparatively large and practical software system written in A-Prolog. The former group can also learn about advantages of A-Prolog with respect to standard Prolog, evident even in the case of plan checking.

An important methodological lesson we learned from this exercise is the importance of careful initial design. For instance, introduction of junction nodes in the model of the Plumbing Module of the RCS substantially simplified the resulting program. We are also satisfied with our use of the Java interface for selecting modules necessary for solving a given problem, and integrating these modules into a final A-Prolog program. Structuring most modules as lp-functions contributed to the reusability and proof of correctness of the integration. Such proof is especially important due to the critical nature of the RCS. Consider the following situation: suppose you have lp-functions f and g correctly implementing the plumbing and basic VCM modules of the system; integration of these modules leads to the creation of new lp-function $h = f \circ g$. It is known that, due to nonmonotonicity of A-Prolog, logic programming representation of this function cannot always be obtained by combining together rules of f and g. In our case, however, a general theorem [72] can be used to check if this is indeed the case.

The people from planning may find it interesting to see a system of substantial size built on theory of action and change. In particular, we were somewhat surprised by the importance of static causal laws in our model. We are not sure that the use of STRIPS-like languages containing only dynamic causal laws is sufficient for a concise representation of the RCS, and especially of the extended VCM.

The use of A-Prolog allowed us to deal with recursive causal laws, which may pose a problem to more classical planning methods. (Partial solution to this problem is suggested in [59], where the authors use CCALC ([133]) to reduce the computation of answer sets to the computation of models of some propositional formula. They give a sufficient condition of the correctness of such transformation. Unfortunately, the idea does not apply here, since the corresponding graph is not acyclic.)

Recent work in planning drew attention to the problem of finding a language which would allow a declarative and efficient representation of heuristic information [10, 99, 104, 68]. We believe that this dissertation demonstrates that a large amount of such information can be naturally expressed in A-Prolog. Moreover, its use dramatically improves efficiency of the planner (which is not always the case for satisfiability based planners.)

Finally, it may be interesting to see how modularity allows planning to be performed in different levels. It is easy, for instance, to modify our planning module to search for manual plans, i.e., those not including computer commands. The new planner will be much more efficient and, in many cases, sufficient for the flight controllers' needs. We have plans of applying these techniques to modeling other systems of the space shuttle.

Chapter 6

Conclusions

"Energy and persistence conquer all things."

Benjamin Franklin (1706-1790)

The purpose of this work is to answer the following two questions:

1. Is it possible to represent a real world problem of reasonable size involving complex effects of actions with the A-Prolog language?

2. Are the available inference engines for A-Prolog able to compute the solutions for such a domain in a reasonably efficient manner?

We have addressed both questions and succeeded in demonstrating that the answer to both is positive. It is important to point out that we have developed the largest and by far most complex application of answer set programming to date. Other planners, to the best of our knowledge, have substantial difficulty in representing domains dealing with state constraints and recursion. The results obtained in this project are so positive that there are indications of their use beyond this application.

A sign of this trend is the present work under development by United Space Alliance programmers to extend our system to other subsystems of the space shuttle.

Another important point is that, in principle, the Theory of Circuits, as well as other parts of our program, which can be viewed as a Theory of Switches, a Theory of Valves, etc., can be re-utilized in the design of other control applications, e.g. systems with mechanical and/or electrical modules.

Even though we have not included the proofs for all our theorems in the dissertation, we have been able to show that some of our programs are provenly correct. This was possible thanks to the general level of knowledge about mathematical properties of the A-Prolog language.

It was also demonstrated that the most sophisticated and powerful inference engines for answer set programming available at the moment have limitations that still need to be addressed. Answer set programming works and allows planning in domains where parallelism and a great deal of knowledge are available. It does not work well when planning involves long plans, and it does not work well with domains which require numerical computations. This work, in this sense, is also mportant because it made clear what the current limitations are. An important contribution of this work is to prove that the language is powerful enough to represent and reason about effects of actions in certain classes of domains. We believe that future work will allow the expansion of this class of programs for those requiring numerical computation and/or that are only partially grounded.

6.1 Lessons Learned

It was believed for some time that A-Prolog is capable of representing default knowledge as well as various forms of knowledge incompleteness. Quite recently, it was noticed that A-Prolog is also suitable for modeling reasoning of agents in dynamic domains. Even more recently, it was understood that methodologies of declarative programming developed in these two areas can be used in many other interesting domains. In this work, the A-Prolog language was used to demonstrate the applicability of this methodology by solving the problem of reasoning in a dynamic domain, including electrical and mechanical system modules.

When modeling complex domains, the syntactic restrictions of A-Prolog can make some rules appear non-natural – in our case, the three rules used in the GD program from Section 3.5.2, in order to exhaustively generate possible input vectors for the circuit. There is some work currently being done on an extension of A-Prolog to deal with sets [95] which is expected to overcome this problem.

We also would like to stress the following software engineering lessons learned from this work:

1. The syntax and semantic of A-Prolog, as well as its mathematical theory, allowed us to quickly build a concise and modular solution to a comparatively non-trivial problem.

2. The solution was constructed in parallel with the development of the proof of

its correctness. Declarativeness of A-Prolog greatly facilitated this process.

3. Reasoning and constraint satisfaction algorithms built in the A-Prolog inference engine proved to be sufficiently efficient for implementing interesting new algorithms for simulation and analysis of digital circuits, planning, plan checking, and even diagnosis. Comparison of their efficiency with respect to other known algorithms remains to be investigated. The preliminary results, especially for planning, are very encouraging.

4. Declarative programs in A-Prolog were nicely integrated with each other and with the Java-based graphical interface allowing a user–friendly interaction with the system.

We believe that the integration of programs written in different languages, with different programming paradigms, will be a trademark of future knowledge-intensive systems.

6.2 Future Work

In this dissertation we developed a decision support system for the space shuttle's flight controllers based on the Reaction Control System. This is a complex domain which is worth further investigation. Our work was mainly concentrated on planning tasks; it would be interesting to examine the issues involving other reasoning tasks. Diagnosis of faulty components from the different modules, e.g. switches, valves, and digital gates, is an obvious candidate task. The idea of reducing the problem of finding a diagnosis for a faulty system to finding stable models of a logic program was first proposed in [62]. Recent work [12, 69, 82] in this area seems to be easily applicable to this domain as far as our preliminary tests indicated. In this context, diagnosis can be viewed as "planning in the past." The relationship between several reasoning tasks, e.g. planning, diagnosis, abduction, is still not entirely clear. We consider A-Prolog the best candidate for specification of these reasoning tasks. Much work remains to be done towards methodologies of use of A-Prolog and development of algorithms for different reasoning tasks.

There are a number of unanswered questions related to the RCS domain; we discuss some of these issues next.

The nature of the RCS system allowed us to create a partitioned representation and to develop an efficient planner based on the parallel computation of independent subgoals. The existence of dependencies among different subparts of a system would, in principle, prevent the use of such technique. We can deal with partial dependencies among subsystems of the RCS by utilizing a different partition of the system. There exists ongoing research on consistency-restoring rules [13] that seems to overcome the limitations imposed by these dependencies. More needs to be done in this direction. This line of research seems to be closely connected to the specification of prioritized defaults [80] – rules establishing preferences among choices available to a reasoning system. This relationship remains to be investigated.

The planner implemented in this dissertation contains a large amount of control knowledge which was easily described in the A-Prolog language. The use of this knowledge dramatically improved the efficiency of the planner. We intend to apply what we have learned from this experiment to other domains and analyze whether this is always possible or for which classes of problems this is mostly adequate. The modular design of our program resulted in planning with different programs, which corresponded to different levels of detail of the domain. To the best of our knowledge, this technique was not used before. Further investigation of its applicability to different classes of programs should be done. Moreover, there are many other approaches for planning, and their correspondence to our approach is still not clear.

It would also be interesting to investigate the reusability of the several modules developed for the RCS, e.g. Plumbing, Valve, and Circuit Theory, in different applications involving similar knowledge.

We have experimented with a version of the RCS/USA-Advisor which uses the DLV inference engine [41, 55]. The results were slower than those obtained with SMODELS. This is due, in part, to the fact that our program does not contain disjunctive rules which can be efficiently computed by DLV. Currently there are no standard benchmarks that could be utilized for comparing the efficiency of A-Prolog inference engines. A real world application of the size of the RCS can be an interesting test bed for this goal, but the differences between these engines must be examined in a much broader spectrum.

There are many interesting open problems directly related to the implementation of A-Prolog engines. Some of these problems, including the ones under investigation by various research groups, are:

1. Modification of algorithms and reasoning engines in order to allow the computation of longer plans and efficient handling of programs corresponding to heavily numerical domains;

2. Development and implementation of new inference rules, in the spirit of the EER rule [142], that would improve the efficiency of the computation of stable models;

3. Development of algorithms allowing the partial grounding [56, 89] of the logic programs given as input to these inference engines;

4. Development of algorithms allowing the parallel computation of stable models. There are several research groups working on parallel engines, for details see [67, 164, 165].

In this dissertation, we have demonstrated the applicability of the A-Prolog language to the representation of defaults and multiple interesting aspects of reasoning about actions and their effects. Several extensions to the language are being developed, e.g. sets and consistency-restoring rules, and much remains to be investigated.

Clearly, both the A-Prolog language and the answer set programming paradigm have experienced an explosive development on the last five years. It was a privilege to see it happen and we hope that this work allows the reader to share the excitement of the many existing possibilities and also gives a glimpse of much that is still to come.

Appendix A

RCS Experiments' Results

> *"It is of great importance that the general public be given the opportunity to experience, consciously and intelligently, the efforts and results of scientific research. It is not sufficient that each result be taken up, elaborated, and applied by few specialists in the field. Restricting the body of knowledge to a small group deadens the philosophical spirit of a people and leads to spiritual poverty."*
>
> *Albert Einstein (1879-1955)*

This appendix presents the detailed results for 2000 experiments performed with the RCS. These experiments were divided in blocks of 200 test instances as explained in Section 5.7. The results for each block of experiments are given in two types of graphs:

1. The "Total Time" graph provides the sum of the time spent on each call to SMODELS until a plan was found, or the last step of the loop with $lasttime = 10$ was processed.

2. The "Number of Steps" graph shows the number of steps contained in each plan found. If the number of steps equals to 0 then no plan was found for that

instance.

A label of the form *results/res-3-2*, appearing on the upper right corner of the graphs, indicates the number of mechanical and electrical faults in the experiments, e.g. *res3-2* means that the instances have 3 mechanical and 2 electrical faults. Appendix A also contains tables summarizing the most important data with respect to the experiments. In these tables, the first column is the test instance number, the second gives the number of RCS subsystems involved in the maneuver (1, 2, or 3), the third is the number of time steps needed, the fourth is the total number of actions performed, the fifth and sixth are the number of rules and atoms used by SMODELS in the grounded code for that test case, and the seventh column is the time, in seconds, needed to find a plan (this time refers only to the time step when the plan was found). The test instance of Figures 5.5, and 5.6 corresponds to instance 8 of Table A.3.

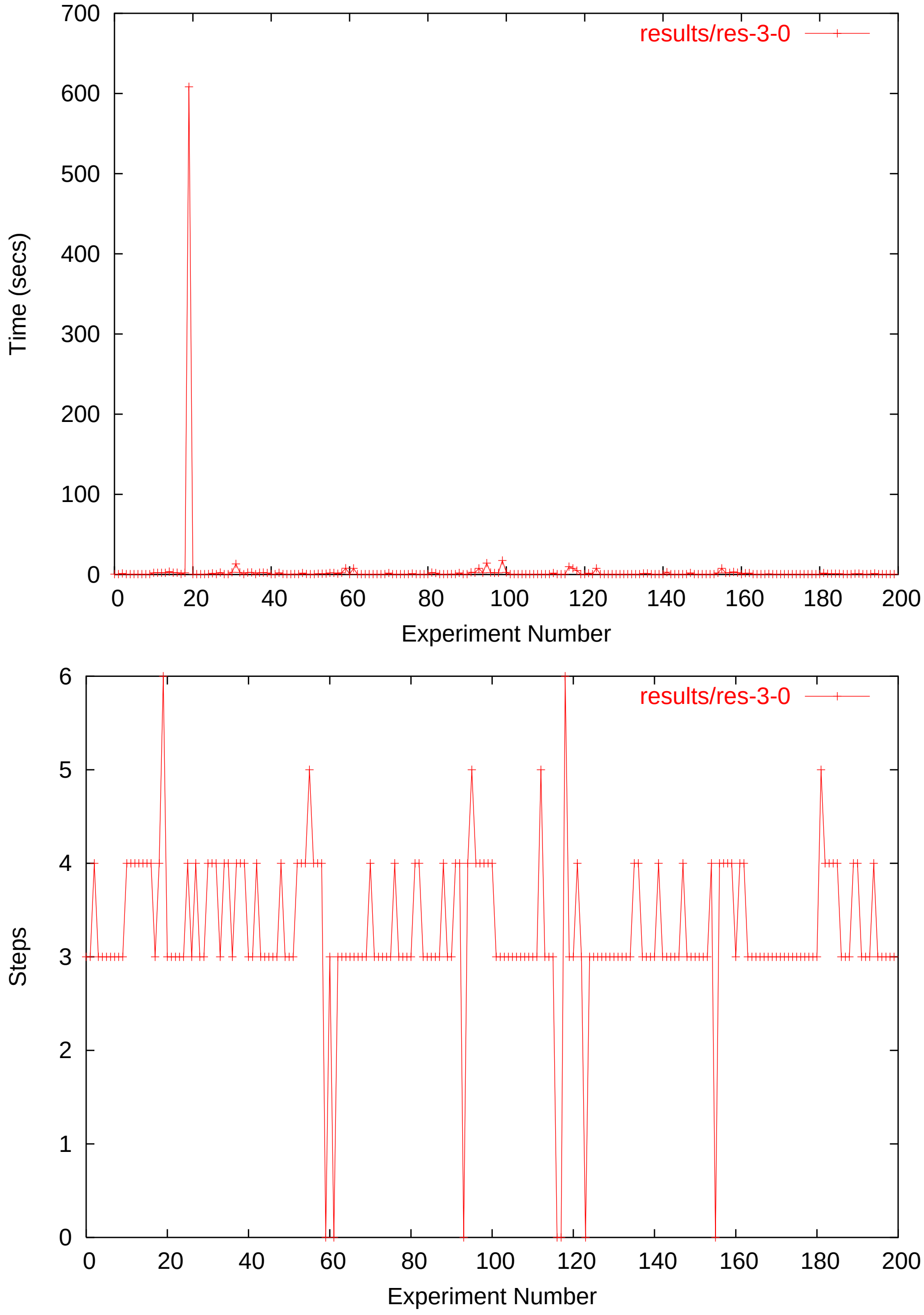


Figure A.1: Results for experiments with 3 mechanical and 0 electrical faults.

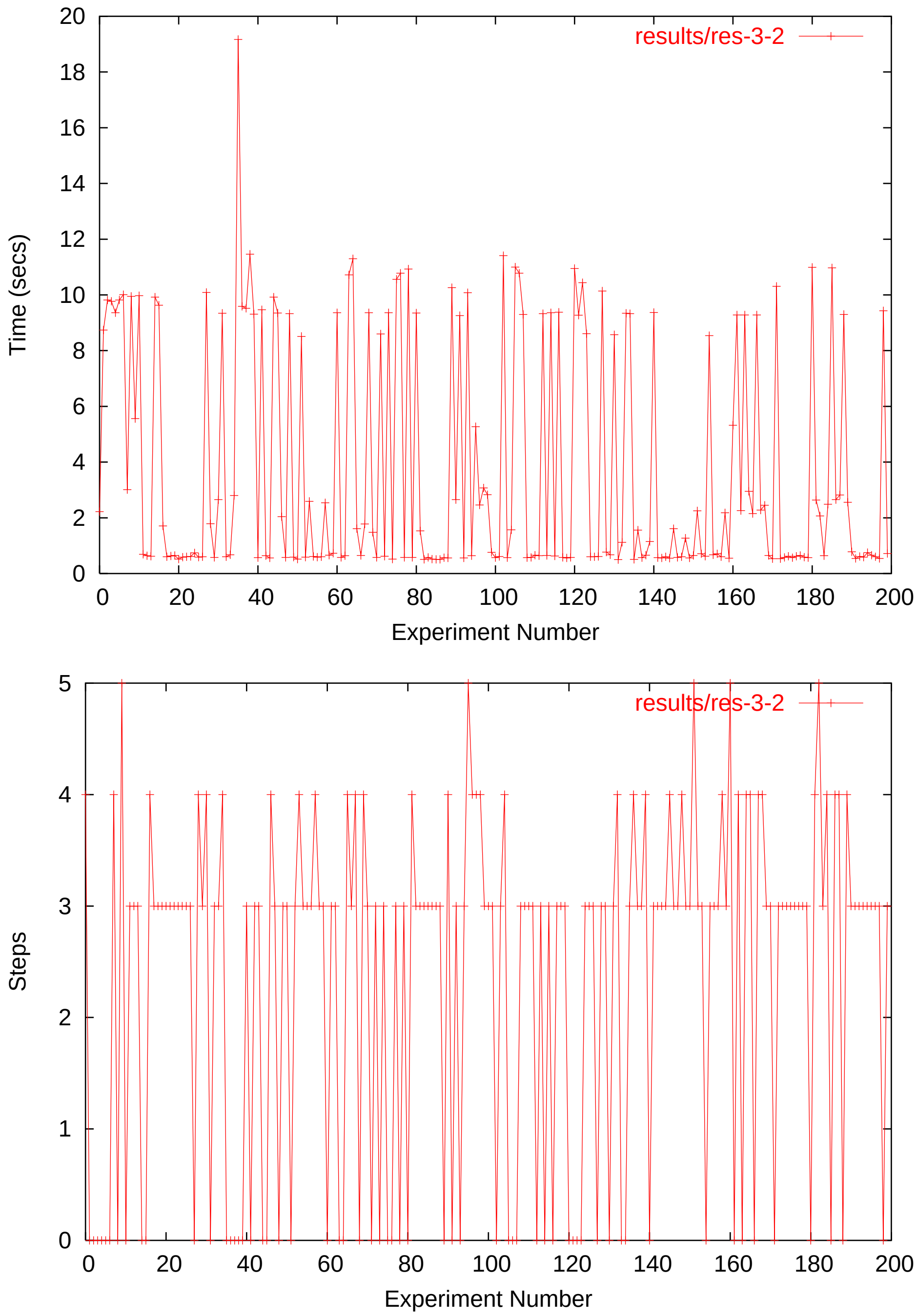


Figure A.2: Results for experiments with 3 mechanical and 2 electrical faults.

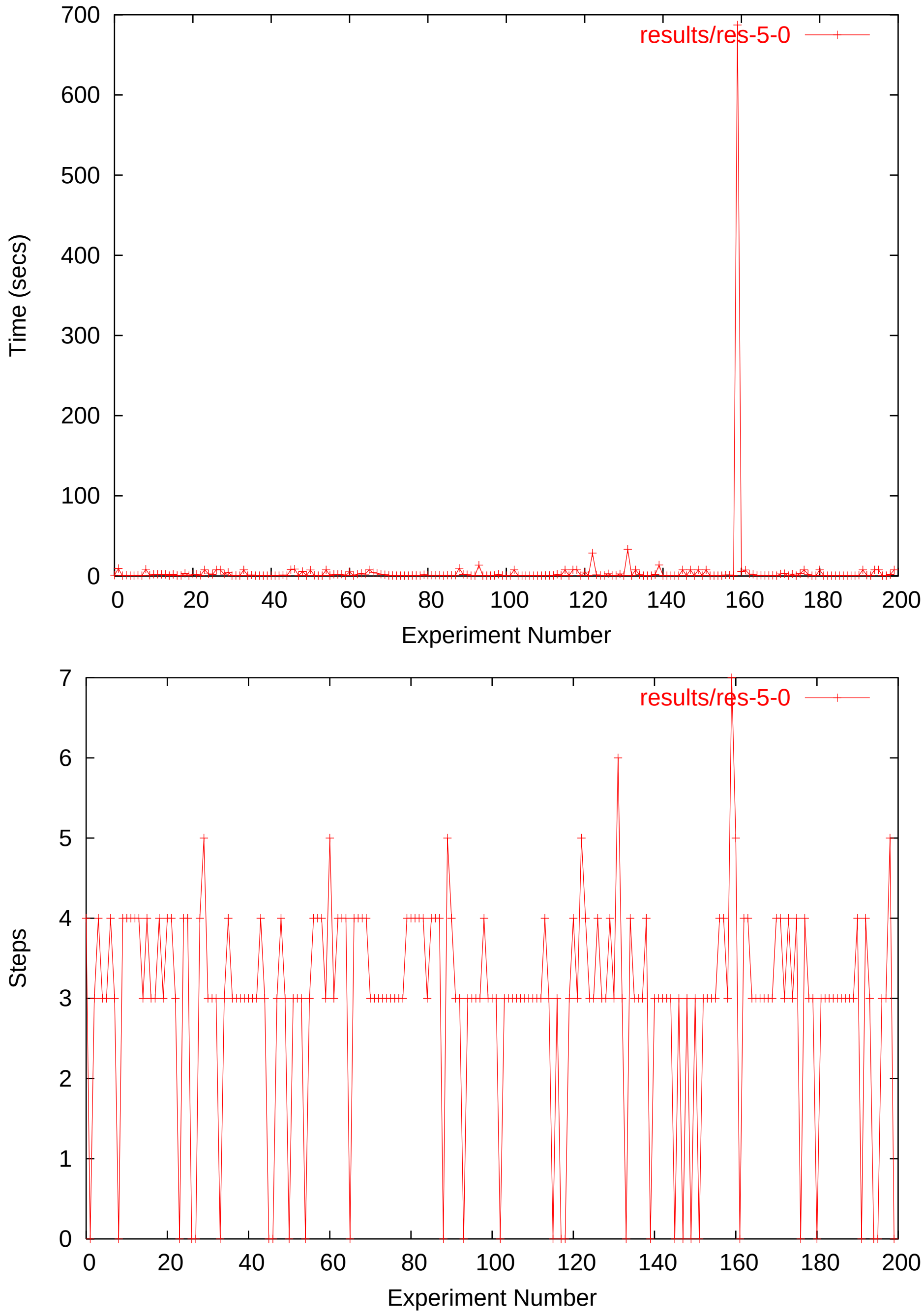


Figure A.3: Results for experiments with 5 mechanical and 0 electrical faults.

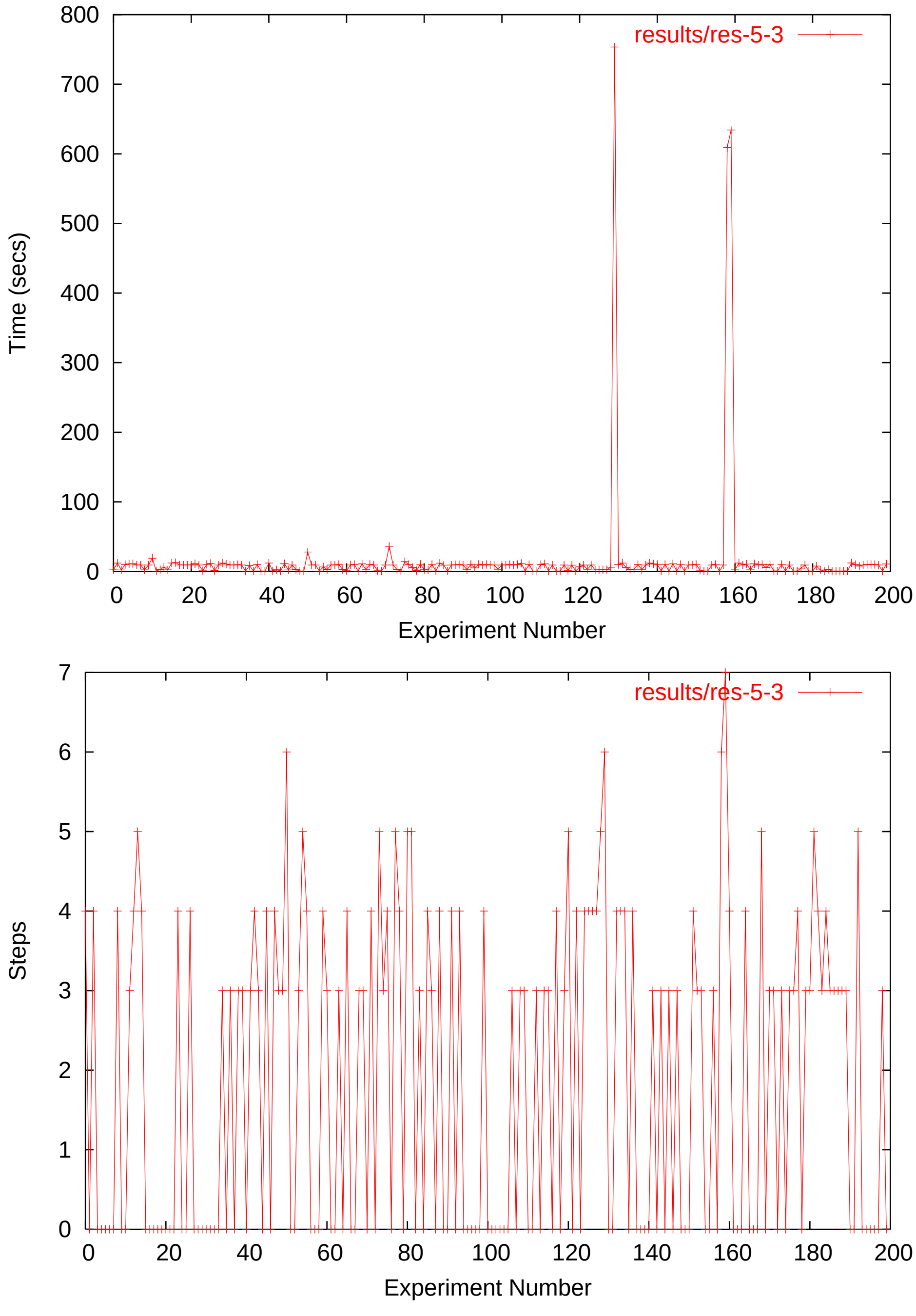


Figure A.4: Results for experiments with 5 mechanical and 3 electrical faults.

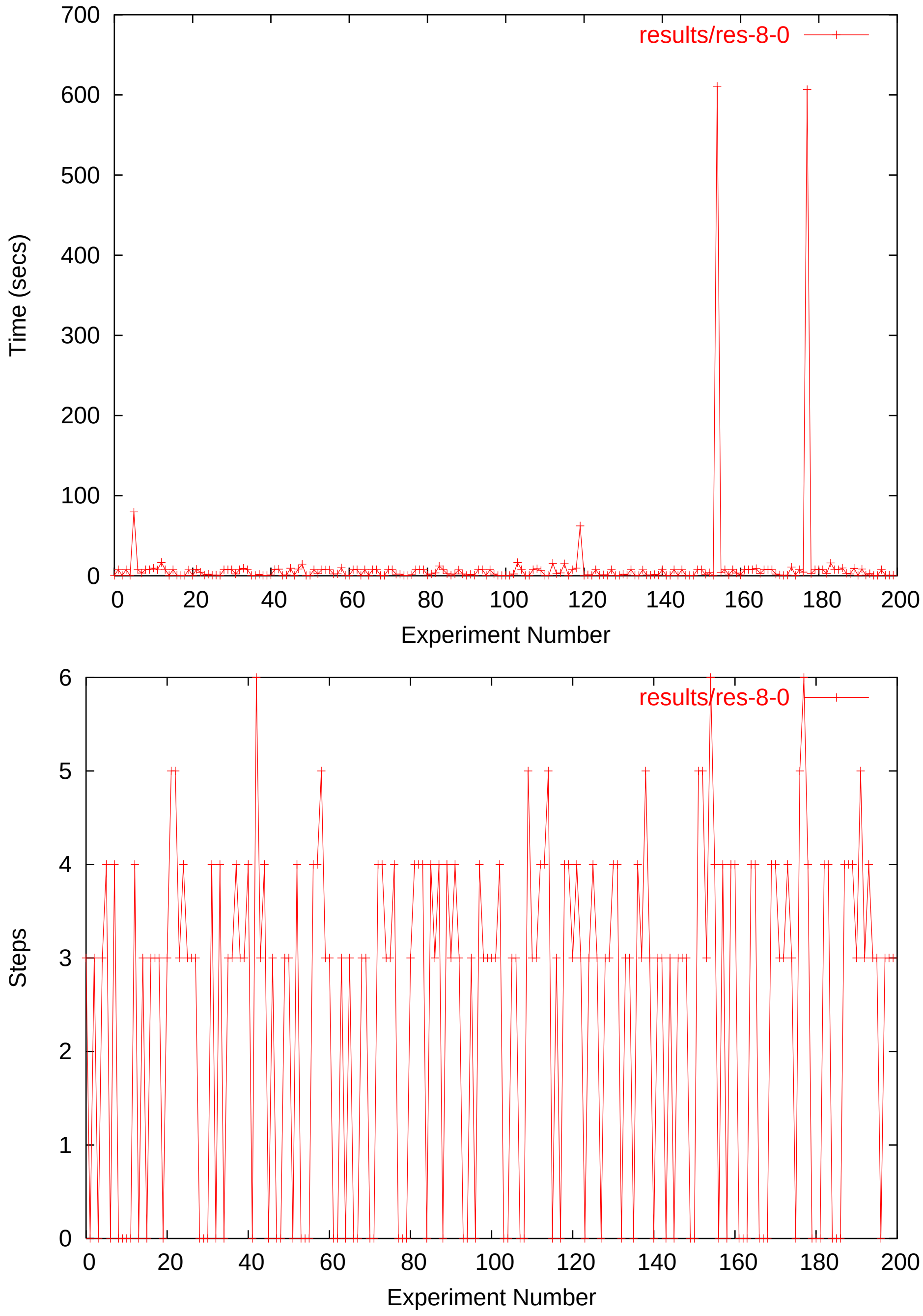


Figure A.5: Results for experiments with 8 mechanical and 0 electrical faults.

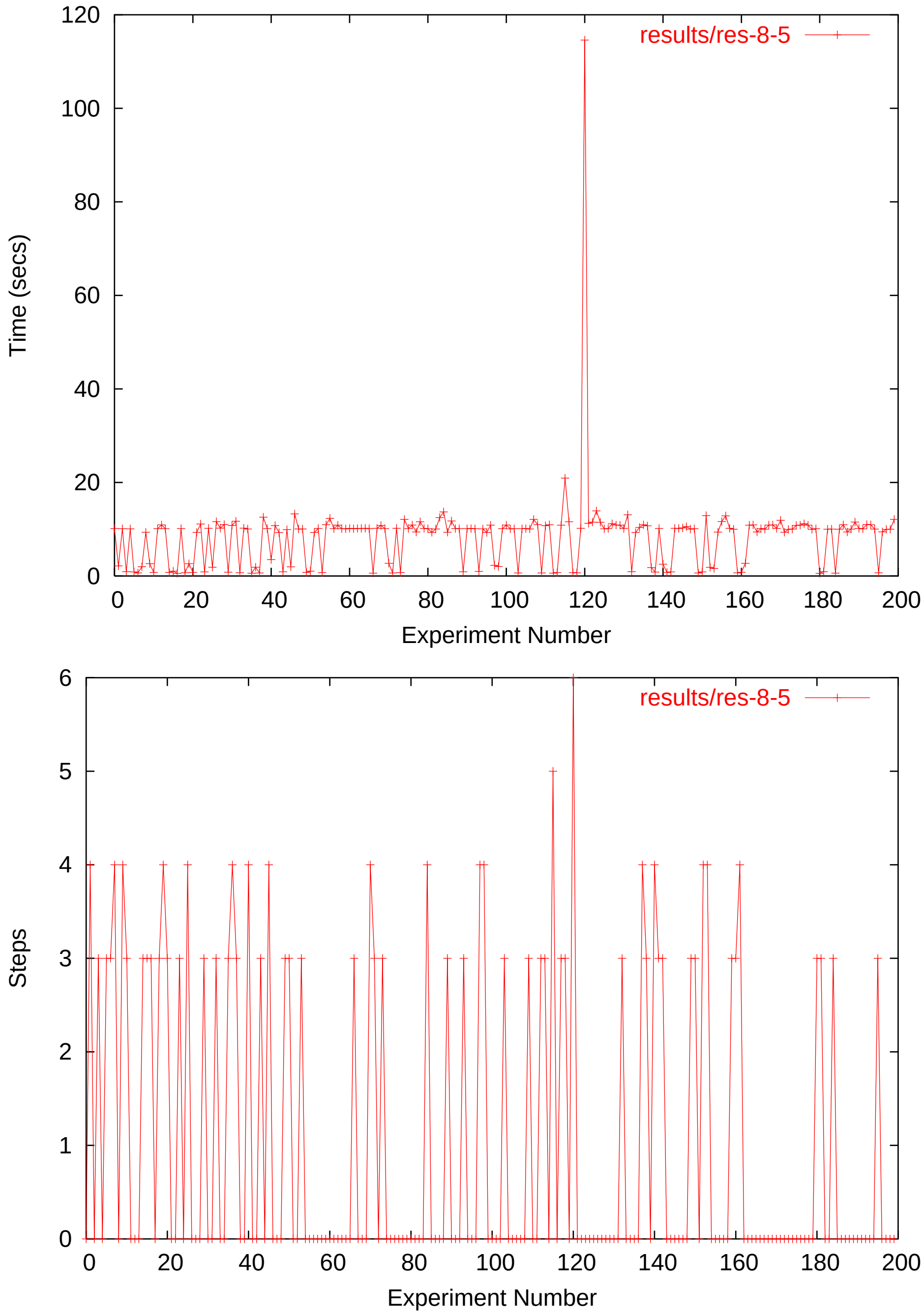
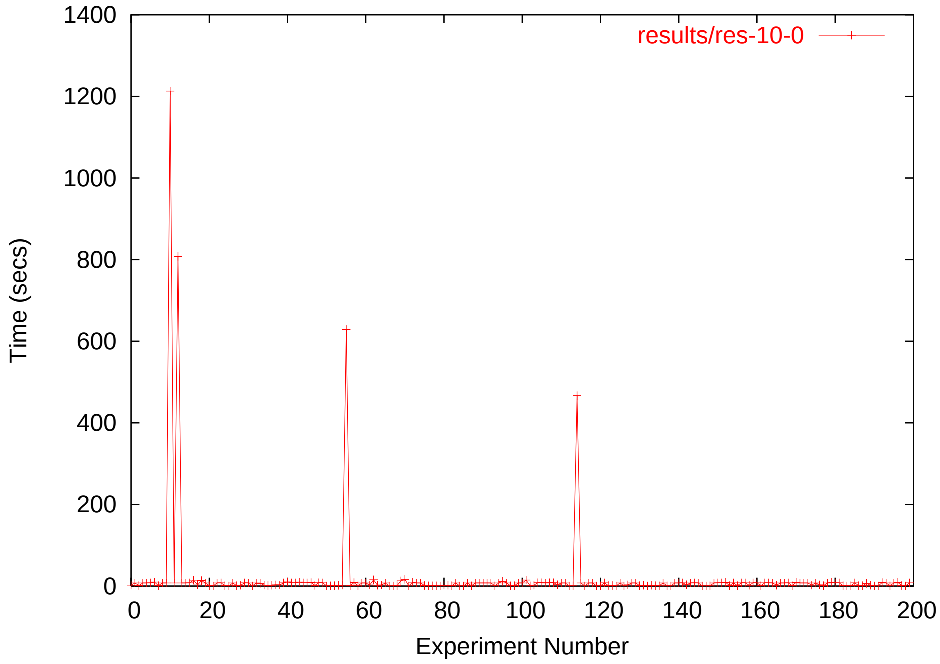


Figure A.6: Results for experiments with 8 mechanical and 5 electrical faults.



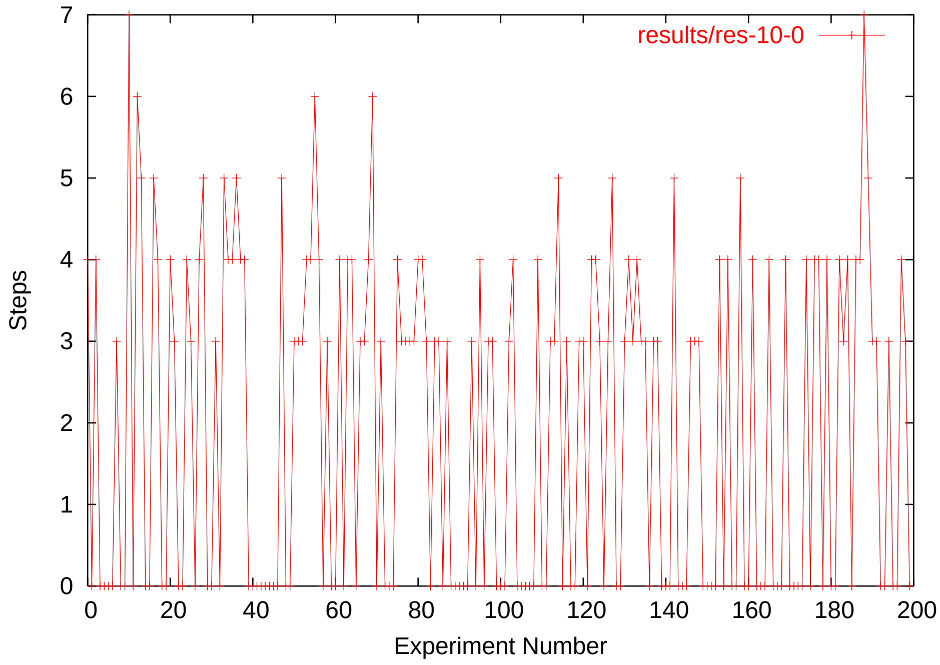


Figure A.7: Results for experiments with 10 mechanical and 0 electrical faults.

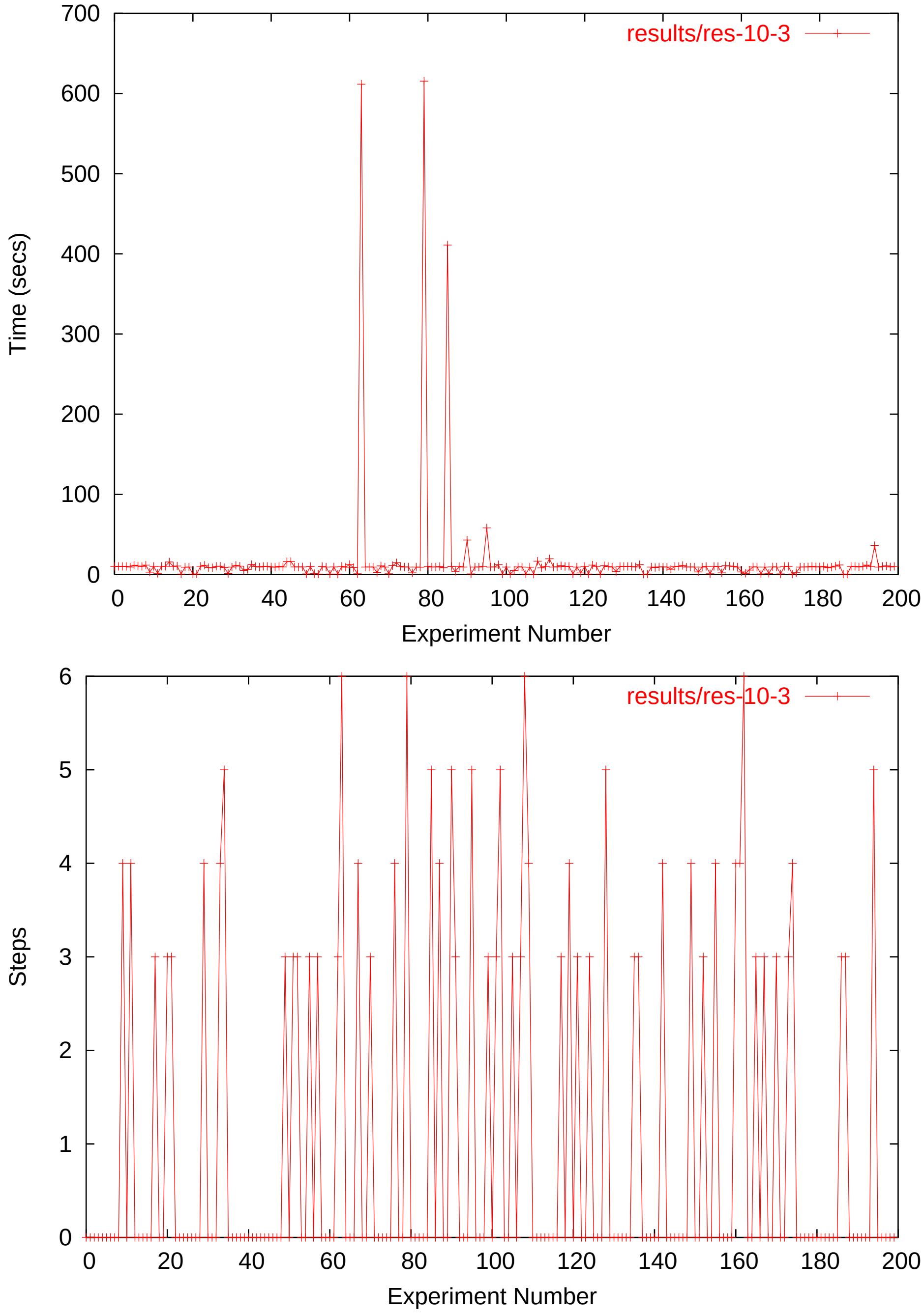
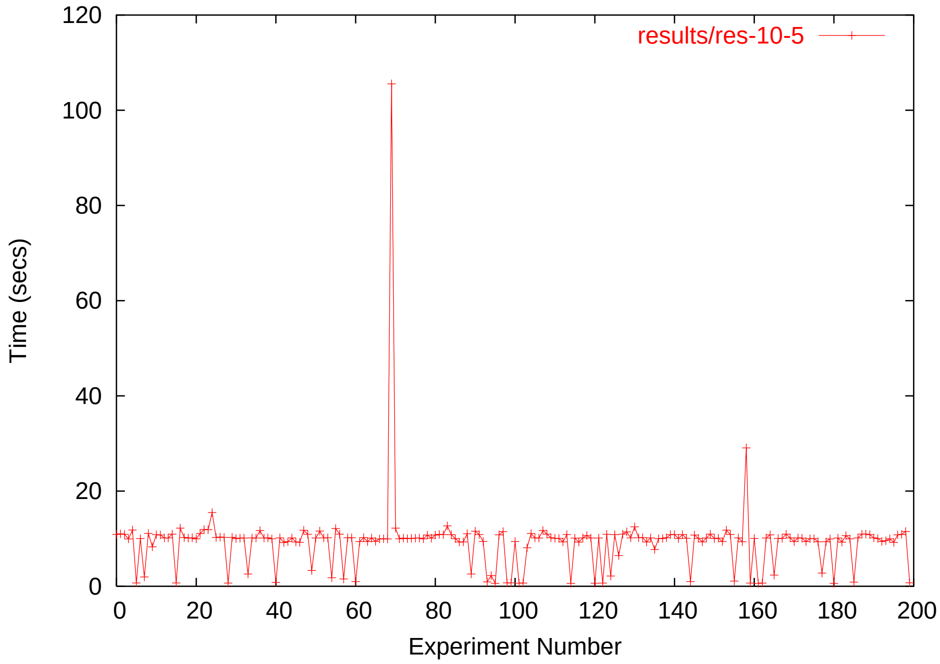


Figure A.8: Results for experiments with 10 mechanical and 3 electrical faults.



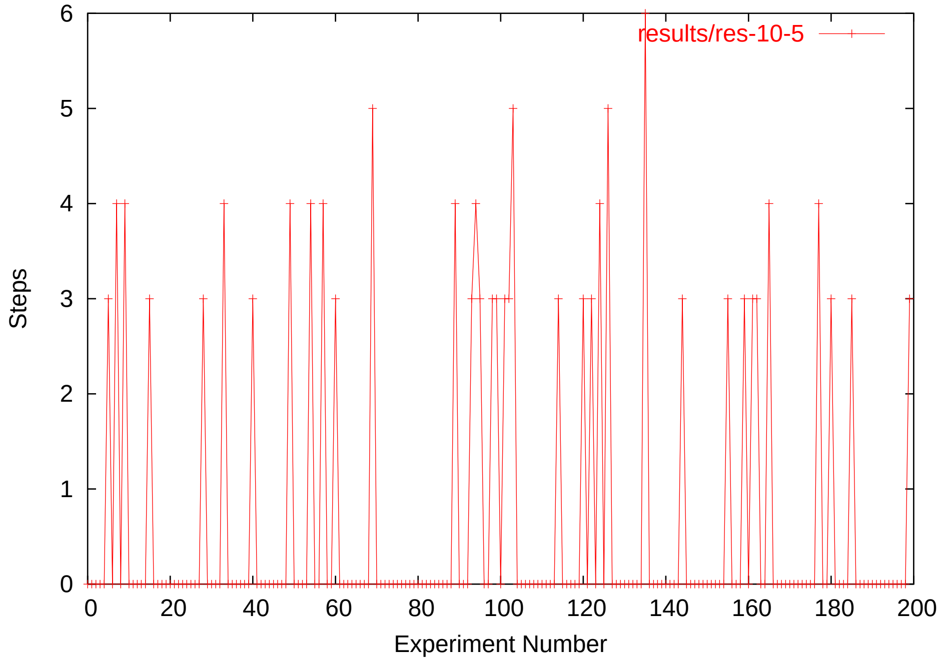


Figure A.9: Results for experiments with 10 mechanical and 5 electrical faults.

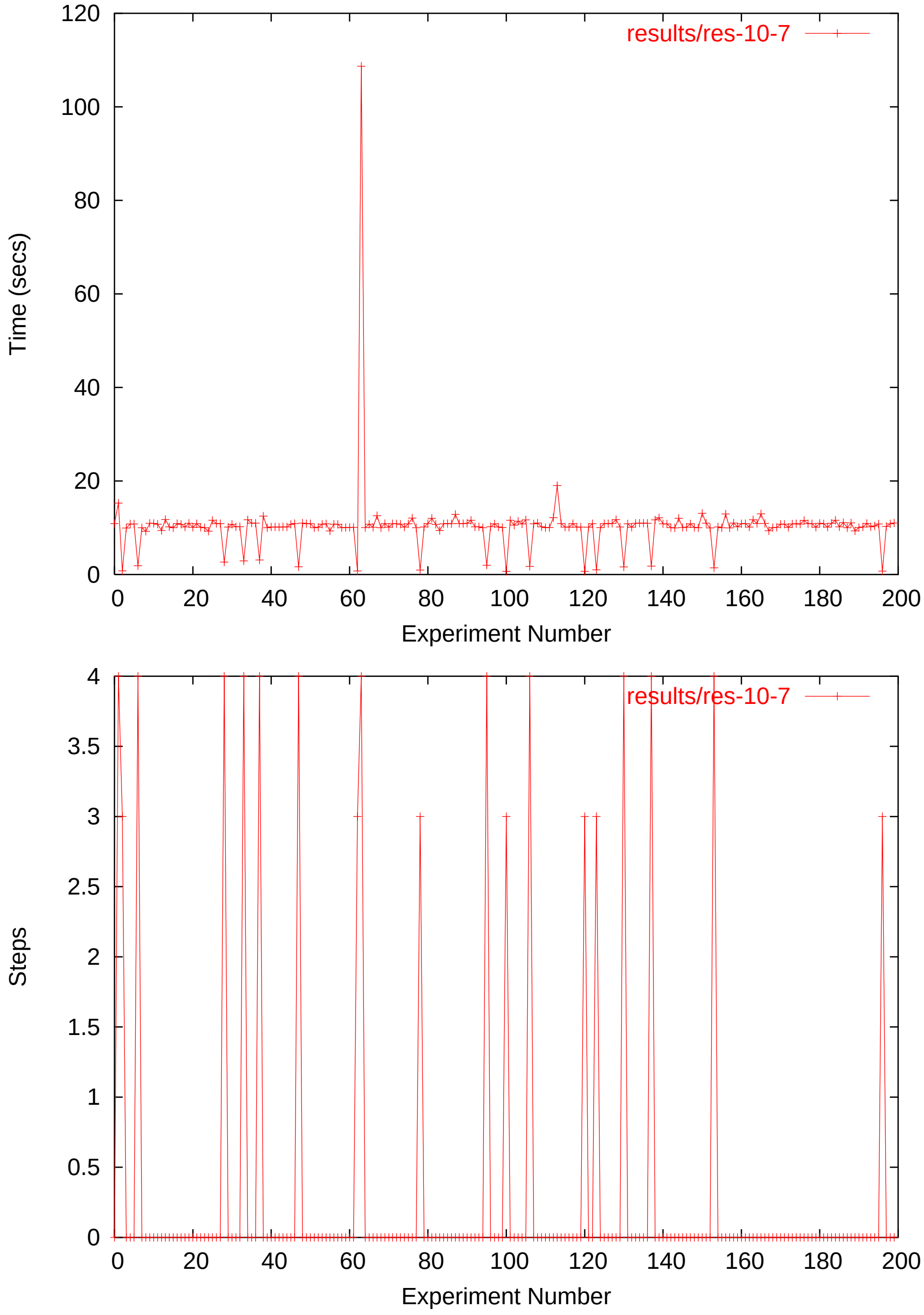


Figure A.10: Results for experiments with 10 mechanical and 7 electrical faults.

Table A.1: Results for experiments with 3 mech. and 0 elect. faults: cases 1-100.

Inst	R	S	A	Rules	Atoms	Time
1	2	3	6	47271	14709	0.430
2	2	3	6	47286	14709	0.450
3	2	4	8	63718	18122	0.970
4	2	3	6	47274	14710	0.440
5	2	3	6	47271	14710	0.440
6	2	3	6	47278	14710	0.450
7	2	3	6	47295	14709	0.460
8	2	3	6	47295	14709	0.450
9	2	3	6	47282	14709	0.440
10	2	3	6	47275	14709	0.450
11	3	4	12	63691	18123	1.690
12	3	4	12	63691	18123	1.640
13	3	4	12	63700	18123	1.680
14	3	4	12	63692	18123	1.740
15	3	4	12	63707	18123	3.120
16	3	4	12	63661	18123	1.830
17	3	4	12	63680	18123	1.840
18	3	3	9	47274	14710	0.620
19	3	4	12	63707	18123	1.780
20	3	6	18	103478	25049	7.290
21	2	3	6	47275	14710	0.550
22	2	3	6	47275	14709	0.450
23	2	3	6	47287	14710	0.500
24	2	3	6	47286	14709	0.440
25	2	3	6	47287	14710	0.660
26	2	4	8	63686	18123	1.080
27	2	3	6	47287	14710	0.440
28	3	4	12	63718	18122	2.060
29	2	3	6	47295	14709	0.560
30	2	3	6	47282	14709	0.450
31	3	4	12	63707	18123	2.360
32	3	4	12	63666	18123	12.770
33	3	4	11	63677	18123	2.270
34	3	3	9	47287	14710	0.490
35	3	4	12	63707	18123	2.030
36	3	4	12	63696	18124	2.240
37	3	3	9	47287	14710	0.480
38	3	4	12	63696	18124	1.810
39	3	4	11	63696	18124	1.960
40	3	4	12	63707	18123	1.560
41	3	3	9	47275	14710	0.560
42	3	3	9	47295	14709	0.470
43	3	4	12	63685	18124	1.750
44	3	3	9	47287	14710	0.710
45	3	3	9	47275	14710	0.450
46	3	3	9	47278	14710	0.470
47	3	3	9	47278	14709	0.730
48	3	3	9	47272	14709	0.470
49	3	4	12	63689	18124	1.380
50	3	3	9	47286	14709	0.580

Inst	R	S	A	Rules	Atoms	Time
51	1	3	3	47274	14710	0.420
52	1	3	3	47247	14710	0.470
53	1	4	4	63671	18124	0.610
54	1	4	4	63685	18124	0.590
55	1	4	4	63718	18122	0.640
56	1	5	5	82373	21570	0.860
57	3	4	12	63681	18124	1.630
58	1	4	4	63687	18123	0.640
59	3	4	12	63707	18123	1.660
60	1	0	0	215364	39301	1.690
61	2	3	6	47295	14709	0.440
62	2	0	0	215422	39299	1.710
63	2	3	6	47267	14710	0.500
64	2	3	6	47265	14710	0.560
65	2	3	6	47263	14710	0.530
66	2	3	6	47279	14709	0.440
67	2	3	6	47247	14710	0.450
68	2	3	6	47295	14709	0.560
69	2	3	6	47303	14708	0.440
70	2	3	6	47263	14709	0.450
71	3	4	12	63707	18123	1.460
72	2	3	6	47262	14709	0.440
73	2	3	6	47301	14707	0.450
74	2	3	6	47287	14710	0.460
75	2	3	6	47258	14710	0.450
76	2	3	6	47295	14709	0.530
77	2	4	8	63696	18123	0.820
78	2	3	6	47295	14709	0.440
79	2	3	6	47266	14710	0.450
80	2	3	6	47295	14709	0.450
81	2	3	6	47287	14710	0.450
82	3	4	12	63691	18123	2.130
83	2	4	8	63682	18123	1.330
84	2	3	6	47287	14710	0.450
85	2	3	6	47303	14708	0.490
86	2	3	6	47278	14710	0.560
87	2	3	6	47287	14710	0.650
88	2	3	6	47270	14710	0.470
89	2	4	7	63707	18123	1.560
90	2	3	6	47287	14710	0.560
91	3	3	9	47271	14709	0.470
92	3	4	11	63680	18124	2.290
93	3	4	12	63682	18123	1.910
94	3	0	0	215393	39300	1.670
95	3	4	12	63677	18123	1.670
96	3	5	14	82373	21570	4.580
97	3	4	11	63696	18124	1.990
98	3	4	12	63696	18124	1.570
99	3	4	11	63696	18124	1.740
100	3	4	11	63707	18123	17.020

Table A.2: Results for experiments with 3 mech. and 0 elect. faults: cases 101-200.

Inst	R	S	A	Rules	Atoms	Time
101	3	4	12	63677	18123	2.240
102	2	3	6	47270	14710	0.440
103	2	3	6	47275	14709	0.450
104	2	3	6	47250	14710	0.460
105	2	3	6	47270	14710	0.550
106	2	3	6	47278	14709	0.440
107	2	3	6	47295	14709	0.440
108	2	3	6	47267	14710	0.520
109	2	3	6	47295	14709	0.430
110	2	3	6	47287	14710	0.420
111	1	3	3	47275	14709	0.450
112	1	3	3	47278	14710	0.450
113	1	5	5	82359	21571	0.850
114	1	3	3	47287	14710	0.440
115	1	3	3	47263	14710	0.430
116	1	3	3	47287	14710	0.430
117	1	0	0	215422	39299	2.400
118	1	0	0	215341	39301	1.710
119	1	6	6	103495	25048	2.560
120	3	3	9	47275	14709	0.480
121	2	3	6	47287	14710	0.550
122	2	4	8	63691	18123	1.160
123	2	3	6	47267	14710	0.460
124	2	0	0	215393	39300	1.710
125	2	3	6	47267	14710	0.440
126	2	3	6	47287	14710	0.560
127	2	3	6	47279	14709	0.450
128	2	3	6	47287	14710	0.450
129	2	3	6	47234	14710	0.470
130	2	3	6	47295	14709	0.440
131	2	3	6	47269	14708	0.460
132	2	3	6	47282	14709	0.460
133	3	3	9	47303	14708	0.470
134	2	3	6	47258	14710	0.430
135	2	3	6	47311	14707	0.440
136	2	4	8	63707	18123	0.830
137	2	4	8	63700	18123	0.940
138	2	3	6	47301	14707	0.450
139	2	3	6	47265	14709	0.460
140	2	3	6	47282	14709	0.550
141	2	3	6	47295	14709	0.440
142	2	4	8	63718	18122	1.820
143	2	3	6	47295	14709	0.530
144	2	3	6	47287	14710	0.560
145	2	3	6	47287	14710	0.460
146	2	3	6	47303	14708	0.460
147	2	3	6	47282	14709	0.460
148	3	4	12	63686	18123	1.610
149	2	3	6	47287	14710	0.460
150	2	3	6	47295	14709	0.550

Inst	R	S	A	Rules	Atoms	Time
151	3	3	9	47278	14710	0.570
152	3	3	9	47283	14708	0.460
153	3	3	9	47295	14709	0.550
154	3	3	9	47277	14709	0.480
155	3	4	12	63696	18124	1.640
156	3	0	0	215393	39300	1.700
157	3	4	12	63676	18124	1.580
158	3	4	12	63660	18124	1.650
159	3	4	12	63707	18123	2.840
160	3	4	12	63662	18123	1.820
161	2	3	6	47295	14709	0.470
162	2	4	8	63691	18123	1.250
163	2	4	8	63707	18122	1.400
164	2	3	6	47287	14710	0.460
165	2	3	6	47295	14709	0.560
166	2	3	6	47261	14710	0.460
167	2	3	6	47295	14709	0.460
168	3	3	9	47278	14710	0.780
169	2	3	6	47274	14710	0.470
170	2	3	6	47287	14710	0.450
171	2	3	6	47271	14710	0.530
172	2	3	6	47265	14710	0.480
173	2	3	6	47257	14710	0.480
174	2	3	6	47283	14708	0.440
175	2	3	6	47275	14709	0.440
176	2	3	6	47295	14709	0.450
177	2	3	6	47287	14710	0.450
178	2	3	6	47303	14708	0.540
179	2	3	6	47287	14710	0.440
180	2	3	6	47295	14709	0.440
181	1	3	3	47278	14709	0.490
182	1	5	5	82293	21571	0.860
183	1	4	4	63696	18124	0.590
184	1	4	4	63650	18124	0.650
185	1	4	4	63729	18121	0.610
186	1	4	4	63696	18124	0.620
187	1	3	3	47287	14710	0.410
188	1	3	3	47295	14709	0.480
189	1	3	3	47263	14710	0.480
190	1	4	4	63707	18123	0.590
191	2	4	8	63718	18122	0.820
192	2	3	6	47303	14708	0.450
193	2	3	6	47275	14709	0.450
194	2	3	6	47287	14710	0.580
195	2	4	8	63682	18123	0.910
196	2	3	6	47263	14710	0.450
197	2	3	6	47271	14710	0.440
198	2	3	6	47271	14709	0.440
199	2	3	6	47295	14709	0.430
200	2	3	6	47287	14710	0.540

Table A.3: Results for experiments with 3 mech. and 2 elect. faults: cases 1-100.

Inst	R	S	A	Rules	Atoms	Time
1	3	4	12	72682	24809	1.760
2	3	0	0	235154	53588	1.950
3	3	0	0	251085	60822	2.100
4	3	0	0	251323	60821	2.010
5	3	0	0	251049	60822	2.000
6	3	0	0	251217	60822	2.150
7	3	0	0	251320	60821	2.090
8	3	4	12	80256	28221	2.440
9	3	0	0	251183	60821	2.110
10	3	5	15	102138	33571	4.070
11	2	0	0	251199	60821	2.140
12	2	3	6	60578	22903	0.690
13	2	3	6	60549	22902	0.640
14	2	3	6	60536	22903	0.620
15	2	0	0	251203	60822	2.070
16	2	0	0	251289	60820	2.040
17	2	4	8	72738	24810	1.220
18	2	3	6	60415	22904	0.600
19	2	3	6	60533	22902	0.630
20	2	3	6	54639	20128	0.650
21	2	3	6	54573	20128	0.520
22	2	3	6	60510	22902	0.590
23	2	3	6	60498	22904	0.600
24	2	3	6	60572	22903	0.610
25	2	3	6	60522	22902	0.740
26	2	3	6	60558	22903	0.590
27	2	3	6	60566	22902	0.600
28	2	0	0	251252	60820	2.000
29	2	4	8	72814	24809	1.320
30	2	3	6	60576	22902	0.580
31	3	4	12	72764	24811	2.180
32	3	0	0	251365	60820	1.990
33	3	0	0	251213	60821	1.980
34	3	3	9	60451	22904	0.670
35	3	4	12	72847	24809	2.320
36	3	0	0	251290	60821	5.560
37	3	0	0	251243	60820	2.080
38	3	0	0	251311	60820	2.020
39	3	0	0	251212	60820	2.500
40	3	0	0	251333	60820	1.980
41	2	3	6	60482	22903	0.570
42	2	0	0	251246	60821	2.020
43	2	3	6	60485	22902	0.640
44	2	3	6	60470	22903	0.560
45	2	0	0	251129	60822	2.130
46	2	3	6	60514	22902	0.670
47	2	4	8	80150	28219	1.510
48	2	3	6	60608	22901	0.580
49	2	0	0	251264	60820	2.000
50	2	3	6	60515	22903	0.590
51	3	3	9	60544	22900	0.700
52	3	0	0	234869	53589	1.840
53	3	3	9	60554	22902	0.590
54	3	4	12	80191	28220	2.040
55	3	3	9	60570	22904	0.610
56	3	3	9	60545	22903	0.590
57	3	3	9	60535	22904	0.600
58	3	4	12	80201	28221	1.940
59	3	3	9	54545	20130	0.660
60	3	3	9	60487	22902	0.730
61	2	0	0	251277	60820	2.000
62	2	3	6	60560	22903	0.580
63	2	3	6	60558	22903	0.640
64	2	0	0	251172	60819	2.340
65	2	0	0	251169	60822	2.570
66	2	4	8	80132	28220	1.090
67	2	3	6	60573	22902	0.650
68	2	4	7	80215	28220	1.240
69	2	0	0	251354	60820	2.010
70	2	4	8	72748	24810	1.010
71	2	3	6	60511	22903	0.580
72	2	0	0	235146	53589	1.860
73	2	3	6	60538	22900	0.620
74	2	0	0	234953	53588	1.860
75	2	3	6	54523	20129	0.520
76	2	0	0	251338	60823	2.380
77	2	0	0	251304	60821	2.460
78	2	3	6	60588	22901	0.580
79	2	0	0	251212	60820	2.510
80	2	3	6	60461	22904	0.580
81	2	0	0	251138	60822	2.000
82	2	4	8	80180	28221	1.010
83	2	3	6	54517	20129	0.510
84	2	3	6	54592	20129	0.510
85	2	3	6	54565	20130	0.520
86	2	3	6	54496	20130	0.510
87	2	3	6	54570	20128	0.510
88	2	3	6	60553	22902	0.570
89	2	3	4	54561	20130	0.560
90	2	0	0	234987	53589	2.360
91	3	4	12	80144	28222	2.130
92	3	0	0	251367	60821	1.980
93	3	0	0	251231	60820	1.990
94	3	0	0	251138	60821	2.160
95	3	3	9	60597	22901	0.640
96	3	5	15	102154	33572	3.860
97	3	4	12	80179	28221	1.930
98	3	4	11	80070	28222	2.550
99	3	4	12	72709	24811	2.360
100	3	3	9	60511	22903	0.760

Table A.4: Results for experiments with 3 mech. and 2 elect. faults: cases 101-200.

Inst	R	S	A	Rules	Atoms	Time	Inst	R	S	A	Rules	Atoms	Time
101	2	3	6	60482	22903	0.570	151	3	3	9	60497	22903	0.650
102	2	3	6	60446	22902	0.610	152	3	3	9	60553	22902	0.600
103	2	0	0	251367	60821	2.590	153	3	3	9	60517	22903	0.710
104	2	3	6	60573	22901	0.570	154	3	3	9	60593	22902	0.620
105	2	4	8	72730	24808	1.170	155	3	0	0	235015	53589	1.840
106	2	0	0	251236	60822	2.540	156	3	3	9	60438	22903	0.670
107	2	0	0	251268	60821	2.440	157	3	3	9	60578	22903	0.710
108	2	3	6	54546	20127	0.510	158	3	3	9	60548	22902	0.600
109	2	3	6	60566	22902	0.570	159	3	4	12	80155	28220	1.670
110	2	3	6	60558	22903	0.580	160	3	3	9	54580	20129	0.550
111	2	3	6	60495	22903	0.560	161	3	5	15	102128	33572	3.970
112	2	3	6	60534	22903	0.640	162	3	0	0	251189	60820	1.970
113	2	0	0	251174	60823	2.000	163	3	4	12	72773	24810	1.790
114	2	3	6	60494	22903	0.650	164	3	0	0	251291	60822	1.990
115	2	0	0	251238	60821	2.000	165	3	4	11	72806	24809	2.490
116	2	3	6	60560	22901	0.630	166	3	4	12	72751	24810	1.680
117	2	0	0	251415	60822	2.000	167	3	0	0	251410	60820	1.990
118	2	3	6	60528	22902	0.580	168	3	4	12	72706	24811	1.810
119	2	3	6	60527	22904	0.560	169	3	4	12	80106	28221	1.930
120	2	3	6	60474	22902	0.580	170	3	3	9	60597	22902	0.650
121	3	0	0	251344	60821	2.450	171	2	3	6	54545	20130	0.530
122	3	0	0	251286	60820	1.980	172	2	0	0	235041	53588	1.850
123	3	0	0	234999	53588	1.830	173	2	3	6	54609	20129	0.530
124	3	0	0	251213	60821	1.990	174	2	3	6	60577	22901	0.580
125	3	3	9	60559	22902	0.600	175	2	3	6	54510	20131	0.620
126	3	3	9	60511	22901	0.600	176	2	3	6	60474	22902	0.570
127	3	3	9	60522	22903	0.610	177	2	3	6	54514	20131	0.620
128	3	0	0	251019	60823	2.180	178	2	3	6	60547	22904	0.650
129	3	3	9	60602	22902	0.770	179	2	3	6	60549	22903	0.590
130	3	3	9	60542	22901	0.670	180	2	3	6	60548	22902	0.570
131	2	0	0	235125	53589	1.860	181	3	0	0	251223	60821	2.410
132	2	3	6	54502	20131	0.500	182	3	4	12	80090	28221	2.090
133	2	3	6	60500	22902	0.570	183	3	5	14	102066	33570	6.080
134	2	0	0	251323	60821	1.990	184	3	3	9	60526	22903	0.640
135	2	0	0	251362	60819	2.000	185	3	4	12	72712	24810	2.020
136	2	3	6	54576	20130	0.510	186	3	0	0	251086	60823	2.390
137	2	4	8	80256	28220	1.040	187	3	4	12	80213	28220	2.130
138	2	3	6	60486	22904	0.570	188	3	4	11	72731	24811	2.350
139	2	3	6	60526	22902	0.660	189	3	0	0	251246	60821	1.980
140	2	3	6	60522	22901	0.670	190	3	4	12	72672	24811	2.090
141	2	0	0	251293	60821	2.010	191	3	3	9	54540	20129	0.780
142	2	3	6	60413	22904	0.570	192	3	3	9	54561	20130	0.540
143	2	3	6	60566	22902	0.560	193	3	3	9	60580	22902	0.610
144	2	3	6	60527	22904	0.600	194	3	3	9	60455	22902	0.590
145	2	3	6	54548	20130	0.550	195	3	3	9	60582	22902	0.750
146	2	4	8	80185	28218	1.090	196	3	3	9	60523	22903	0.660
147	2	3	6	60618	22901	0.580	197	3	3	9	60558	22900	0.600
148	2	3	6	54581	20129	0.600	198	3	3	9	54548	20128	0.540
149	2	3	6	60590	22902	0.650	199	3	0	0	251329	60819	1.980
150	2	3	6	60514	22901	0.560	200	3	3	9	60550	22902	0.720

Table A.5: Results for experiments with 5 mech. and 0 elect. faults: cases 1-100.

Inst	R	S	A	Rules	Atoms	Time
1	1	4	4	63669	18127	0.570
2	1	0	0	215258	39303	2.320
3	2	3	6	47269	14711	0.430
4	1	4	4	63662	18127	0.590
5	2	3	6	47265	14712	0.470
6	1	3	3	47235	14714	0.440
7	1	4	4	63637	18128	0.640
8	2	3	6	47291	14712	0.530
9	1	0	0	215261	39304	1.950
10	1	4	4	63646	18126	0.620
11	3	4	11	63700	18126	1.730
12	3	4	12	63668	18127	1.740
13	3	4	12	63685	18126	1.590
14	3	4	12	63689	18127	1.530
15	3	3	9	47275	14712	0.530
16	3	4	12	63659	18127	1.550
17	3	3	9	47291	14712	0.600
18	3	3	9	47274	14713	0.690
19	3	4	12	63700	18126	2.790
20	3	3	9	47299	14711	0.530
21	3	4	11	63623	18128	1.890
22	3	4	12	63643	18127	1.800
23	3	3	9	47291	14712	0.860
24	3	0	0	215368	39303	1.670
25	3	4	12	63648	18127	2.230
26	3	4	12	63623	18128	1.520
27	3	0	0	215313	39303	1.670
28	3	0	0	215313	39303	1.680
29	3	4	12	63678	18128	1.850
30	3	5	15	82349	21574	3.630
31	2	3	6	47253	14713	0.440
32	2	3	6	47250	14713	0.580
33	2	3	6	47282	14712	0.460
34	2	0	0	215339	39304	1.700
35	2	3	6	47222	14714	0.460
36	2	4	7	63664	18127	0.920
37	2	3	6	47289	14711	0.480
38	2	3	6	47255	14714	0.500
39	2	3	6	47251	14714	0.450
40	2	3	6	47275	14712	0.450
41	2	3	6	47238	14714	0.530
42	2	3	6	47238	14714	0.450
43	2	3	6	47254	14713	0.450
44	2	4	8	63691	18125	0.940
45	2	3	6	47250	14713	0.450
46	2	0	0	215397	39302	1.710
47	2	0	0	215282	39304	1.940
48	2	3	6	47282	14712	0.510
49	2	4	8	63673	18127	1.690
50	2	3	6	47246	14712	0.440
51	3	0	0	215368	39303	1.690
52	3	3	9	47238	14714	0.840
53	3	3	9	47291	14712	0.560
54	3	3	9	47274	14712	0.570
55	3	0	0	215339	39304	1.670
56	3	3	9	47291	14712	0.660
57	3	4	12	63643	18126	1.650
58	3	4	12	63664	18126	1.590
59	3	4	12	63711	18125	1.900
60	3	3	9	47225	14714	0.460
61	3	5	13	82363	21573	4.180
62	3	3	9	47253	14713	0.460
63	3	4	11	63647	18127	1.900
64	3	4	12	63700	18125	3.080
65	3	4	11	63689	18127	1.920
66	3	0	0	215293	39304	1.690
67	3	4	12	63722	18124	3.740
68	3	4	12	63689	18126	3.260
69	3	4	12	63662	18127	1.810
70	2	4	8	63648	18127	1.180
71	2	3	6	47242	14714	0.610
72	2	3	6	47266	14714	0.670
73	2	3	6	47231	14714	0.550
74	2	3	6	47259	14712	0.440
75	2	3	6	47242	14714	0.450
76	2	3	6	47250	14714	0.460
77	2	3	6	47291	14712	0.680
78	2	3	6	47283	14713	0.620
79	2	3	6	47282	14712	0.510
80	2	4	8	63673	18127	1.300
81	1	4	4	63672	18126	0.610
82	1	4	4	63621	18128	0.620
83	1	4	4	63666	18125	0.620
84	1	4	4	63638	18128	0.610
85	2	3	6	47271	14712	0.470
86	1	4	4	63659	18127	0.600
87	1	4	4	63637	18128	0.640
88	1	4	4	63722	18124	0.620
89	1	0	0	215278	39303	2.350
90	1	5	5	82316	21574	0.950
91	2	4	8	63653	18127	1.220
92	2	3	6	47253	14713	0.510
93	2	3	6	47279	14711	0.490
94	2	0	0	215339	39304	3.840
95	2	3	6	47266	14714	0.460
96	2	3	6	47246	14713	0.470
97	2	3	6	47283	14713	0.460
98	2	3	6	47251	14712	0.610
99	2	4	6	63653	18127	1.730
100	2	3	4	47274	14713	0.610

Table A.6: Results for experiments with 5 mech. and 0 elect. faults: cases 101-200.

Inst	R	S	A	Rules	Atoms	Time	Inst	R	S	A	Rules	Atoms	Time
101	2	3	6	47281	14712	0.460	151	2	3	6	47259	14713	0.480
102	2	3	6	47283	14713	0.430	152	2	0	0	215162	39304	1.690
103	2	0	0	215305	39304	1.710	153	2	3	6	47283	14713	0.550
104	2	3	4	47262	14714	0.510	154	2	3	6	47263	14714	0.530
105	2	3	4	47255	14714	0.540	155	2	3	6	47275	14714	0.460
106	2	3	6	47278	14712	0.570	156	2	3	6	47241	14714	0.430
107	2	3	6	47271	14713	0.540	157	2	4	8	63722	18124	0.940
108	2	3	6	47274	14713	0.510	158	2	4	8	63648	18127	0.830
109	2	3	6	47274	14712	0.450	159	2	3	6	47283	14713	0.440
110	2	3	6	47254	14713	0.440	160	2	7	12	127030	28567	75.810
111	3	3	9	47242	14714	0.590	161	3	5	15	82347	21572	4.620
112	3	3	9	47259	14713	0.650	162	3	0	0	215384	39301	1.670
113	3	3	8	47238	14714	0.660	163	3	4	12	63673	18127	2.120
114	3	4	12	63627	18127	1.730	164	3	4	12	63678	18127	1.650
115	3	3	9	47223	14713	0.670	165	3	3	9	47233	14714	0.710
116	3	0	0	215313	39303	1.690	166	2	3	6	47291	14712	0.830
117	3	3	9	47263	14713	0.500	167	3	3	9	47291	14712	0.650
118	3	0	0	215368	39303	1.700	168	3	3	9	47283	14713	0.590
119	3	0	0	215368	39303	1.670	169	2	3	6	47283	14713	0.750
120	3	3	9	47283	14713	0.550	170	2	3	6	47261	14713	0.520
121	3	4	12	63659	18126	4.490	171	3	4	12	63664	18127	2.260
122	3	3	9	47283	14713	0.620	172	3	4	12	63664	18126	2.560
123	3	5	15	82352	21572	6.260	173	2	3	6	47238	14714	0.440
124	2	4	8	63673	18127	0.990	174	3	4	12	63640	18128	1.970
125	3	3	9	47291	14712	0.580	175	2	3	6	47285	14712	0.500
126	3	3	9	47275	14714	0.600	176	3	4	11	63659	18127	2.580
127	3	4	11	63689	18127	2.330	177	3	0	0	215368	39303	1.670
128	3	3	9	47257	14712	0.580	178	3	4	12	63648	18127	1.720
129	3	3	9	47275	14714	0.710	179	2	3	6	47282	14712	0.450
130	3	4	11	63658	18127	2.140	180	2	3	6	47281	14712	0.550
131	2	3	6	47263	14713	0.500	181	2	0	0	215368	39303	1.680
132	2	6	12	103381	25054	31.300	182	2	3	6	47262	14713	0.440
133	2	3	6	47255	14714	0.500	183	2	3	6	47283	14713	0.420
134	2	0	0	215306	39304	1.680	184	2	3	6	47291	14712	0.430
135	2	4	8	63668	18127	1.260	185	2	3	6	47261	14713	0.450
136	2	3	6	47291	14712	0.490	186	2	3	6	47271	14713	0.440
137	2	3	6	47234	14713	0.430	187	2	3	6	47242	14714	0.500
138	2	3	6	47267	14713	0.510	188	2	3	6	47283	14713	0.500
139	2	4	8	63700	18126	1.310	189	2	3	6	47266	14714	0.430
140	2	0	0	215259	39304	3.910	190	2	3	6	47283	14713	0.450
141	2	3	6	47291	14712	0.460	191	1	4	4	63662	18128	0.600
142	2	3	6	47262	14714	0.570	192	1	0	0	215339	39304	1.690
143	2	3	6	47299	14711	0.450	193	1	4	4	63633	18127	0.600
144	2	3	6	47263	14713	0.500	194	1	3	3	47274	14713	0.450
145	2	3	6	47266	14713	0.510	195	1	0	0	215231	39305	1.690
146	2	0	0	215270	39304	1.670	196	1	0	0	215316	39304	1.690
147	2	3	6	47274	14713	0.430	197	1	3	3	47267	14713	0.420
148	2	0	0	215261	39304	1.680	198	1	3	3	47266	14714	0.440
149	2	3	6	47274	14713	0.430	199	1	5	5	82347	21573	0.800
150	2	0	0	215313	39303	1.680	200	1	0	0	215294	39304	1.690

Table A.7: Results for experiments with 5 mech. and 3 elect. faults: cases 1-100.

Inst	R	S	A	Rules	Atoms	Time
1	3	4	12	80175	28226	1.860
2	3	0	0	267514	68059	2.710
3	3	4	12	80065	28229	1.740
4	3	0	0	267311	68060	2.120
5	3	0	0	267507	68060	2.290
6	3	0	0	251228	60827	2.490
7	3	0	0	251300	60825	1.990
8	3	0	0	251224	60826	1.980
9	3	4	11	80052	28228	1.980
10	3	0	0	251088	60825	1.990
11	3	0	0	267514	68059	5.310
12	3	3	9	60509	22908	0.600
13	3	4	12	80070	28229	2.200
14	3	5	15	101998	33579	4.880
15	3	4	11	80156	28227	1.980
16	3	0	0	267300	68060	2.630
17	3	0	0	267240	68060	2.840
18	3	0	0	251174	60828	1.990
19	3	0	0	251070	60825	1.990
20	3	0	0	251185	60823	1.990
21	1	0	0	251096	60825	2.000
22	1	0	0	267249	68059	2.480
23	1	0	0	251187	60826	2.020
24	1	4	4	72678	24815	0.760
25	1	0	0	251149	60829	2.240
26	3	0	0	267517	68060	2.500
27	1	4	4	80108	28224	0.750
28	1	0	0	251215	60826	2.020
29	3	0	0	267486	68059	2.580
30	1	0	0	251310	60825	2.420
31	2	0	0	251031	60826	2.000
32	3	0	0	251300	60825	2.000
33	2	0	0	251021	60828	2.130
34	2	0	0	250798	60829	1.990
35	2	3	6	60447	22908	0.600
36	2	0	0	235023	53594	1.850
37	2	3	6	60532	22905	0.720
38	2	0	0	267316	68058	2.140
39	2	3	6	66386	25683	0.740
40	2	3	6	60553	22908	0.620
41	3	0	0	267290	68059	2.750
42	3	3	9	66521	25679	0.910
43	3	4	12	87514	31636	2.090
44	3	3	9	60477	22909	0.680
45	3	0	0	251224	60826	2.530
46	3	4	12	80250	28225	1.800
47	3	0	0	251125	60827	1.980
48	3	4	12	80137	28224	2.240
49	3	3	9	60434	22909	0.830
50	3	3	9	60437	22910	0.580

Inst	R	S	A	Rules	Atoms	Time
51	3	6	18	126340	38958	14.050
52	3	0	0	251041	60827	1.980
53	3	0	0	251272	60827	1.990
54	3	3	9	60399	22909	0.610
55	3	5	15	110977	37624	4.980
56	3	4	12	80184	28226	2.240
57	3	0	0	250992	60826	1.980
58	3	0	0	251109	60825	2.000
59	3	0	0	267197	68059	2.120
60	3	4	12	80078	28226	2.240
61	3	3	9	60422	22909	1.110
62	3	0	0	250955	60828	1.980
63	3	0	0	251077	60828	2.160
64	3	3	9	60491	22908	0.600
65	3	0	0	267397	68060	2.320
66	3	4	12	80069	28226	2.050
67	1	0	0	267496	68056	2.140
68	3	0	0	251311	60825	1.980
69	3	3	9	60502	22909	1.040
70	3	3	9	60375	22911	0.630
71	3	0	0	251266	60825	1.990
72	3	4	12	80143	28227	35.720
73	3	0	0	251249	60826	1.990
74	1	5	5	101990	33577	1.560
75	3	3	9	66395	25683	0.840
76	3	4	12	80036	28226	13.740
77	3	0	0	267188	68061	2.120
78	3	5	13	102029	33578	4.180
79	1	4	4	80248	28226	0.750
80	3	0	0	250920	60828	2.220
81	1	5	5	102003	33579	1.460
82	1	5	5	102044	33578	1.030
83	1	0	0	251310	60825	2.270
84	3	3	9	60445	22908	0.760
85	3	0	0	267081	68061	2.550
86	3	4	12	80098	28225	7.580
87	3	3	9	60478	22908	0.760
88	3	0	0	251238	60824	2.020
89	3	4	11	87574	31634	9.050
90	3	0	0	250999	60829	2.140
91	3	0	0	251191	60826	2.000
92	3	4	12	87547	31639	2.050
93	1	0	0	267240	68060	2.210
94	3	4	12	80131	28226	4.720
95	3	0	0	234902	53595	2.280
96	3	0	0	251055	60828	2.000
97	3	0	0	267345	68058	2.130
98	1	0	0	251174	60828	1.990
99	3	0	0	251090	60826	1.990
100	3	4	12	87549	31634	2.570

Table A.8: Results for experiments with 5 mech. and 3 elect. faults: cases 101-200.

Inst	R	S	A	Rules	Atoms	Time
101	2	0	0	251181	60827	2.000
102	1	0	0	251049	60825	2.010
103	2	0	0	267197	68060	2.150
104	2	0	0	251172	60826	2.010
105	2	0	0	267093	68061	2.140
106	2	0	0	267385	68060	2.660
107	2	3	6	66447	25682	0.660
108	2	0	0	251067	60828	2.180
109	2	3	6	66498	25679	0.620
110	2	3	6	60447	22910	0.570
111	2	0	0	250994	60827	2.120
112	2	0	0	250958	60830	2.510
113	2	3	6	60480	22909	0.640
114	2	0	0	251093	60827	2.000
115	2	3	6	60501	22908	0.650
116	2	3	6	66440	25683	0.700
117	1	0	0	251164	60824	1.980
118	2	4	8	80113	28227	1.060
119	3	0	0	251096	60825	1.970
120	2	3	6	60489	22909	0.670
121	3	5	15	110852	37624	5.240
122	3	0	0	251126	60827	2.020
123	3	4	12	80018	28228	2.670
124	3	0	0	251121	60825	1.990
125	3	4	12	72692	24815	1.860
126	3	4	12	72707	24816	1.960
127	3	4	12	80060	28225	2.010
128	3	4	12	80134	28226	2.160
129	3	5	13	110831	37625	4.570
130	3	6	18	126231	38962	9.500
131	3	0	0	251147	60826	2.180
132	3	0	0	267438	68059	2.640
133	3	4	12	80139	28224	5.120
134	3	4	12	87587	31637	2.330
135	3	4	12	80083	28228	2.620
136	3	0	0	251282	60826	2.170
137	3	4	12	80031	28225	2.070
138	3	0	0	250768	60829	1.980
139	3	0	0	267428	68059	2.640
140	3	0	0	251113	60826	2.380
141	2	0	0	267373	68059	2.150
142	2	3	5	66471	25682	0.710
143	2	0	0	267353	68060	2.150
144	2	3	6	54521	20136	0.650
145	2	0	0	251065	60825	2.530
146	2	3	5	60560	22906	0.690
147	2	0	0	267254	68060	2.150
148	2	3	5	60560	22908	0.630
149	3	0	0	251151	60829	1.980
150	2	0	0	251206	60827	2.020

Inst	R	S	A	Rules	Atoms	Time
151	2	0	0	267298	68060	2.150
152	2	4	8	80200	28225	1.380
153	2	3	6	60485	22910	0.690
154	2	3	6	66404	25681	0.620
155	3	0	0	251128	60827	1.990
156	2	0	0	267411	68060	2.150
157	2	3	6	60536	22907	0.630
158	2	0	0	250936	60827	2.020
159	3	6	14	126275	38961	7.810
160	2	7	14	141233	39057	16.560
161	3	4	12	80186	28227	1.880
162	3	0	0	267299	68061	2.540
163	3	0	0	251179	60826	1.990
164	3	0	0	251061	60828	2.400
165	3	4	12	80195	28225	1.820
166	3	0	0	267460	68058	2.360
167	3	0	0	251204	60826	2.000
168	3	0	0	267296	68060	2.130
169	3	5	15	102026	33577	4.460
170	3	0	0	251041	60827	2.170
171	2	3	6	60492	22908	0.690
172	2	3	6	60541	22907	0.680
173	2	0	0	267496	68060	2.150
174	2	3	6	60525	22911	0.760
175	2	0	0	251195	60827	2.020
176	2	3	6	60378	22911	0.590
177	2	3	6	66504	25683	0.760
178	2	4	8	87468	31637	4.300
179	3	0	0	250985	60827	1.990
180	2	3	6	54499	20136	0.700
181	2	3	6	66359	25684	0.630
182	2	5	10	110865	37623	3.880
183	2	4	8	72639	24816	1.720
184	2	3	6	60463	22910	0.580
185	3	4	12	80214	28224	2.370
186	2	3	6	60449	22907	0.570
187	2	3	6	60486	22909	0.830
188	2	3	6	60435	22908	0.580
189	2	3	6	60446	22908	0.840
190	2	3	6	66475	25683	0.720
191	2	0	0	267198	68060	2.770
192	2	0	0	251031	60828	2.100
193	3	5	14	101917	33578	6.400
194	3	0	0	250980	60826	1.990
195	3	0	0	251011	60828	2.160
196	3	0	0	267394	68060	2.130
197	3	0	0	267472	68059	2.130
198	2	0	0	250787	60829	2.010
199	2	3	6	60439	22908	0.590
200	3	0	0	267488	68060	2.320

Table A.9: Results for experiments with 8 mech. and 0 elect. faults: cases 1-100.

Inst	R	S	A	Rules	Atoms	Time
1	2	3	6	47247	14718	0.490
2	2	0	0	215345	39307	1.680
3	2	3	6	47262	14716	0.520
4	2	0	0	215151	39308	1.690
5	2	3	6	47244	14718	0.430
6	3	4	12	63647	18132	79.270
7	3	0	0	215374	39306	1.670
8	3	4	12	63657	18132	1.900
9	2	0	0	215289	39307	1.680
10	3	0	0	215143	39309	1.690
11	3	0	0	215220	39309	2.310
12	3	0	0	215112	39310	1.670
13	3	4	12	63624	18133	16.170
14	2	0	0	215253	39309	1.680
15	2	3	6	47226	14719	0.500
16	2	0	0	215242	39309	1.690
17	2	3	6	47228	14719	0.730
18	2	3	6	47250	14716	0.540
19	2	3	6	47251	14718	0.450
20	3	0	0	215177	39309	1.660
21	2	3	6	47224	14719	0.530
22	3	5	15	82292	21578	5.680
23	2	5	10	82289	21579	3.450
24	2	3	6	47248	14717	0.890
25	3	4	12	63579	18133	1.530
26	2	3	6	47236	14719	0.710
27	2	3	6	47272	14717	0.790
28	2	3	6	47234	14717	0.570
29	2	0	0	215282	39308	1.680
30	2	0	0	215316	39308	1.690
31	2	0	0	215316	39308	1.690
32	2	4	8	63684	18131	1.810
33	2	0	0	215137	39310	1.680
34	2	4	8	63677	18129	1.990
35	2	0	0	215253	39309	1.680
36	2	3	6	47265	14719	0.500
37	2	3	6	47212	14717	0.550
38	2	4	8	63635	18133	1.350
39	2	3	6	47253	14719	0.520
40	2	3	6	47265	14717	0.440
41	1	4	4	63704	18128	0.590
42	2	0	0	215176	39309	1.710
43	2	6	12	103353	25059	4.250
44	2	3	6	47236	14719	0.560
45	1	4	4	63594	18134	0.590
46	1	0	0	215161	39308	2.360
47	2	3	6	47247	14718	0.460
48	1	0	0	215311	39306	1.950
49	2	0	0	215198	39309	4.260
50	2	3	6	47227	14718	0.450

Inst	R	S	A	Rules	Atoms	Time
51	2	3	6	47235	14717	0.570
52	2	0	0	215308	39306	1.710
53	3	4	12	63673	18131	2.200
54	3	0	0	215316	39308	1.660
55	3	0	0	215310	39305	1.660
56	3	0	0	215307	39305	1.670
57	3	4	12	63610	18133	1.810
58	3	4	12	63685	18129	2.070
59	3	5	11	82315	21577	3.660
60	3	3	9	47223	14718	0.720
61	2	3	6	47269	14717	0.530
62	2	0	0	215232	39309	1.680
63	2	0	0	215258	39310	1.680
64	2	3	6	47237	14720	0.470
65	2	0	0	215314	39307	1.690
66	2	3	6	47245	14719	0.440
67	2	0	0	215224	39310	1.690
68	2	0	0	215238	39308	1.680
69	2	3	6	47240	14717	0.500
70	2	3	6	47201	14718	0.500
71	2	0	0	215170	39310	1.680
72	2	0	0	215248	39308	1.660
73	2	4	8	63654	18131	1.450
74	2	4	7	63622	18133	1.520
75	2	3	6	47217	14720	0.520
76	2	3	6	47240	14720	0.470
77	2	4	8	63613	18131	0.970
78	2	0	0	215254	39309	1.690
79	2	0	0	215316	39308	1.680
80	2	0	0	215293	39308	1.690
81	3	3	9	47245	14719	0.810
82	3	4	11	63615	18134	1.890
83	3	4	12	63595	18132	3.250
84	3	4	12	63632	18132	11.930
85	2	0	0	215316	39308	1.680
86	3	4	12	63688	18128	2.170
87	3	3	9	47249	14718	0.590
88	3	4	11	63660	18131	1.840
89	3	0	0	215188	39309	1.670
90	3	4	12	63610	18134	1.790
91	2	3	6	47207	14718	0.490
92	2	4	8	63647	18131	1.380
93	2	3	6	47256	14719	0.600
94	2	0	0	215272	39308	1.680
95	2	0	0	215261	39308	1.680
96	2	3	6	47218	14718	0.470
97	2	0	0	215345	39307	1.680
98	2	4	7	63598	18132	1.720
99	2	3	6	47287	14715	0.440
100	2	3	6	47215	14719	0.440

Table A.10: Results for experiments with 8 mech. and 0 elect. faults: cases 101-200.

Inst	R	S	A	Rules	Atoms	Time
101	2	3	6	47232	14718	0.430
102	2	3	6	47264	14718	0.420
103	3	4	11	63662	18132	1.840
104	3	0	0	215322	39307	4.980
105	3	0	0	215203	39310	1.680
106	3	3	9	47216	14719	0.460
107	2	3	6	47265	14719	0.470
108	3	0	0	215339	39306	1.670
109	3	0	0	215343	39306	1.990
110	3	5	15	82323	21577	3.610
111	2	3	6	47185	14719	0.430
112	3	3	9	47232	14718	0.730
113	3	4	12	63595	18133	15.070
114	3	4	10	63579	18133	2.190
115	2	5	9	82328	21578	2.080
116	3	0	0	215264	39309	4.370
117	3	3	9	47257	14718	0.450
118	2	0	0	215232	39309	1.700
119	3	4	12	63612	18133	9.190
120	3	4	12	63680	18129	61.780
121	2	3	6	47256	14719	0.460
122	2	4	8	63632	18131	0.870
123	2	3	6	47203	14719	0.430
124	2	0	0	215316	39308	1.690
125	2	3	6	47265	14719	0.610
126	2	4	8	63657	18129	0.940
127	2	3	6	47212	14719	0.500
128	2	0	0	215199	39309	1.680
129	2	3	4	47229	14718	0.470
130	2	3	4	47220	14719	0.510
131	3	4	12	63626	18132	1.490
132	1	4	4	63695	18130	0.630
133	3	0	0	215232	39309	1.670
134	3	3	9	47273	14718	0.650
135	3	3	9	47230	14718	0.440
136	3	0	0	215187	39309	1.670
137	1	4	4	63691	18127	0.630
138	3	3	9	47213	14720	0.600
139	1	5	5	82321	21577	0.790
140	3	3	9	47213	14719	0.830
141	2	0	0	215229	39309	1.680
142	2	3	6	47245	14717	0.480
143	2	3	6	47215	14720	0.450
144	2	0	0	215344	39306	1.690
145	2	3	6	47219	14718	0.530
146	2	0	0	215077	39310	1.670
147	2	3	6	47239	14716	0.620
148	2	3	6	47256	14717	0.510
149	2	3	6	47252	14719	0.570
150	1	0	0	215237	39308	1.700

Inst	R	S	A	Rules	Atoms	Time
151	3	0	0	215230	39308	1.660
152	1	5	5	82243	21581	0.840
153	3	5	14	82297	21579	2.940
154	3	3	9	47210	14719	0.540
155	3	6	18	103324	25059	9.640
156	3	4	12	63610	18131	3.450
157	1	0	0	215236	39308	1.700
158	1	4	4	63637	18132	0.610
159	3	0	0	215314	39307	1.680
160	3	4	12	63663	18130	3.060
161	1	4	4	63679	18130	0.600
162	3	0	0	215232	39309	1.680
163	3	0	0	215197	39309	1.670
164	3	0	0	215271	39308	1.680
165	3	4	11	63659	18131	8.630
166	3	4	11	63695	18130	2.390
167	3	0	0	215254	39309	1.680
168	3	0	0	215267	39307	1.670
169	1	0	0	215285	39309	1.700
170	3	4	12	63595	18132	1.780
171	1	4	4	63673	18132	0.610
172	3	3	9	47257	14718	0.450
173	3	3	9	47221	14719	0.740
174	3	4	12	63673	18132	10.510
175	3	3	9	47195	14720	0.500
176	3	0	0	215207	39308	1.690
177	3	5	15	82311	21578	3.600
178	3	6	18	103420	25058	5.620
179	3	4	11	63567	18132	2.340
180	3	0	0	215226	39308	1.670
181	3	0	0	215285	39308	1.680
182	3	0	0	215240	39308	1.680
183	3	4	12	63674	18128	2.470
184	3	4	12	63676	18129	15.680
185	3	0	0	215345	39307	1.670
186	3	0	0	215332	39307	1.680
187	3	0	0	215157	39310	2.420
188	3	4	12	63591	18133	2.400
189	3	4	12	63653	18131	1.860
190	3	4	12	63684	18131	3.580
191	2	3	6	47202	14717	0.460
192	3	5	14	82357	21576	4.250
193	2	3	5	47212	14719	0.800
194	3	4	12	63666	18130	2.360
195	2	3	6	47260	14718	0.430
196	2	3	5	47241	14718	0.570
197	2	0	0	215287	39309	1.680
198	2	3	6	47245	14719	0.640
199	2	3	6	47264	14718	0.580
200	2	3	6	47256	14718	0.700

Table A.11: Results for experiments with 8 mech. and 5 elect. faults: cases 1-100.

Inst	R	S	A	Rules	Atoms	Time
1	2	0	0	267156	68067	2.140
2	2	4	8	87328	31649	1.570
3	2	0	0	267076	68069	2.110
4	2	3	6	66346	25693	0.920
5	2	0	0	266792	68072	2.140
6	2	3	6	66326	25691	0.870
7	2	3	6	66450	25691	0.650
8	2	4	8	87455	31645	1.420
9	2	0	0	250806	60835	2.000
10	2	4	8	87501	31644	1.660
11	2	3	6	66436	25689	0.750
12	2	0	0	266981	68071	2.130
13	2	0	0	283121	75301	2.290
14	2	0	0	267232	68066	2.130
15	2	3	6	66323	25691	0.770
16	2	3	6	60346	22919	1.040
17	2	3	3	60461	22917	0.550
18	2	0	0	266898	68071	2.140
19	2	3	6	72335	28464	0.700
20	2	4	8	87521	31646	1.970
21	2	3	6	66324	25693	0.760
22	2	0	0	250876	60833	2.010
23	1	0	0	267046	68070	2.440
24	2	3	6	66420	25689	0.880
25	1	0	0	266643	68070	2.150
26	2	4	8	87510	31644	1.310
27	1	0	0	283267	75302	2.440
28	1	0	0	267053	68069	2.150
29	1	0	0	283350	75299	2.310
30	2	3	6	66413	25690	0.800
31	2	0	0	283183	75301	2.260
32	2	0	0	282949	75304	2.490
33	2	3	6	66390	25689	0.690
34	2	0	0	267201	68070	2.150
35	2	0	0	266973	68068	2.120
36	2	3	6	60379	22921	0.590
37	2	4	8	87538	31643	1.290
38	2	3	6	66376	25691	0.650
39	2	0	0	283146	75303	2.830
40	2	0	0	267052	68065	2.150
41	2	4	8	87467	31642	2.880
42	2	0	0	267146	68067	2.280
43	2	0	0	250695	60837	1.970
44	2	3	6	72296	28465	0.910
45	2	0	0	250973	60836	2.120
46	2	4	8	87351	31648	1.370
47	2	0	0	283332	75301	3.000
48	2	0	0	267010	68070	2.140
49	1	0	0	266835	68070	2.150
50	2	3	6	66401	25691	0.750

Inst	R	S	A	Rules	Atoms	Time
51	3	3	8	66369	25691	1.030
52	3	0	0	250560	60839	1.990
53	1	0	0	267196	68066	2.150
54	3	3	9	66357	25693	0.740
55	1	0	0	267067	68070	2.420
56	1	0	0	283181	75304	2.750
57	3	0	0	266932	68069	2.130
58	3	0	0	283379	75301	2.280
59	3	0	0	267187	68067	2.140
60	1	0	0	266695	68069	2.130
61	2	0	0	267191	68068	2.150
62	1	0	0	267013	68069	2.140
63	2	0	0	266965	68069	2.150
64	2	0	0	267123	68070	2.150
65	2	0	0	266958	68072	2.150
66	2	0	0	267174	68069	2.140
67	2	3	6	66305	25692	0.620
68	2	0	0	267142	68065	2.150
69	2	0	0	267119	68069	2.290
70	2	0	0	266882	68072	2.140
71	2	4	7	87508	31646	1.620
72	2	3	6	66388	25690	0.650
73	2	0	0	267234	68066	2.160
74	2	3	6	66370	25691	0.740
75	1	0	0	283294	75300	2.640
76	1	0	0	267011	68069	2.130
77	1	0	0	283313	75301	2.280
78	2	0	0	250853	60837	2.000
79	2	0	0	283228	75302	2.420
80	2	0	0	267133	68069	2.150
81	3	0	0	267349	68068	2.140
82	3	0	0	251049	60836	1.990
83	3	0	0	266701	68071	2.130
84	3	0	0	266948	68066	2.890
85	3	4	12	87444	31646	13.130
86	3	0	0	250865	60833	1.990
87	3	0	0	267402	68068	2.510
88	1	0	0	267159	68072	2.150
89	3	0	0	267206	68065	2.130
90	2	3	6	66297	25694	0.910
91	3	0	0	267165	68070	2.130
92	3	0	0	267138	68068	2.140
93	3	0	0	266991	68068	2.130
94	3	3	8	66348	25691	0.980
95	3	0	0	267006	68071	2.130
96	3	0	0	250576	60838	1.990
97	3	0	0	250701	60836	2.360
98	2	4	8	87382	31647	1.690
99	2	4	8	87458	31644	1.450
100	3	0	0	266940	68068	2.130

Table A.12: Results for experiments with 8 mech. and 5 elect. faults: cases 101-200.

Inst	R	S	A	Rules	Atoms	Time
101	3	0	0	282966	75304	2.280
102	2	0	0	267201	68070	2.130
103	2	0	0	267321	68065	2.130
104	2	3	6	60379	22921	0.640
105	3	0	0	267054	68070	2.140
106	3	0	0	267385	68065	2.150
107	2	0	0	267300	68066	2.120
108	3	0	0	267434	68068	2.720
109	3	0	0	267031	68068	2.340
110	2	3	6	66376	25693	0.640
111	2	0	0	266979	68069	2.270
112	2	0	0	283420	75301	2.300
113	2	3	6	66271	25692	0.630
114	2	3	6	66352	25693	0.740
115	2	0	0	283186	75302	2.270
116	2	5	10	119683	41679	4.420
117	2	0	0	283578	75302	2.430
118	2	3	6	66408	25688	0.710
119	2	3	6	66407	25691	0.680
120	2	0	0	267459	68068	2.150
121	3	6	16	136374	43655	10.190
122	3	0	0	266984	68069	2.460
123	3	0	0	283404	75299	2.400
124	3	0	0	283370	75296	3.140
125	3	0	0	267272	68067	2.430
126	2	0	0	267052	68065	2.150
127	3	0	0	267065	68068	2.140
128	2	0	0	267116	68064	2.580
129	3	0	0	283508	75302	2.280
130	2	0	0	267136	68067	2.330
131	2	0	0	267143	68070	2.150
132	2	0	0	283399	75298	2.900
133	2	3	6	72296	28465	0.930
134	2	0	0	250728	60835	1.990
135	2	0	0	267252	68070	2.140
136	2	0	0	283500	75301	2.290
137	2	0	0	283259	75300	2.240
138	2	4	8	79945	28238	1.240
139	2	3	6	66349	25690	0.850
140	2	0	0	266811	68070	2.150
141	2	4	8	94990	35056	1.870
142	2	3	6	66256	25694	0.750
143	2	3	6	72380	28464	0.880
144	2	0	0	267127	68069	2.160
145	2	0	0	267002	68070	2.140
146	2	0	0	267045	68071	2.250
147	2	0	0	267342	68066	2.220
148	2	0	0	266931	68070	2.120
149	2	0	0	267195	68069	2.130
150	2	3	6	66408	25691	0.650

Inst	R	S	A	Rules	Atoms	Time
151	2	3	6	66253	25693	0.860
152	2	0	0	283564	75301	2.890
153	2	4	8	87506	31648	1.240
154	2	4	8	87419	31648	1.030
155	2	0	0	250669	60838	2.010
156	2	0	0	283290	75301	2.460
157	2	0	0	283383	75301	2.900
158	2	0	0	267271	68064	2.150
159	3	0	0	267044	68070	2.110
160	2	3	6	66278	25691	0.670
161	3	3	9	60316	22918	0.780
162	3	4	12	87507	31646	2.140
163	2	0	0	267036	68069	2.300
164	2	0	0	283170	75303	2.290
165	2	0	0	251099	60834	2.010
166	2	0	0	266959	68067	2.140
167	3	0	0	267012	68071	2.110
168	3	0	0	267260	68066	2.320
169	2	0	0	283209	75302	2.280
170	2	0	0	267157	68067	2.130
171	3	0	0	267287	68066	2.750
172	3	0	0	251040	60837	2.000
173	3	0	0	267056	68066	2.100
174	3	0	0	266935	68070	2.100
175	3	0	0	283418	75300	2.270
176	3	0	0	283209	75298	2.260
177	3	0	0	251132	60836	2.520
178	3	0	0	283191	75302	2.270
179	3	0	0	267156	68068	2.120
180	3	0	0	267035	68066	2.140
181	2	3	6	60397	22917	0.560
182	2	3	6	72274	28466	0.930
183	3	0	0	266931	68070	2.100
184	3	0	0	267257	68068	2.120
185	2	3	6	60370	22921	0.630
186	3	0	0	267193	68066	2.110
187	2	0	0	283413	75300	2.300
188	2	0	0	251225	60834	2.020
189	3	0	0	266976	68069	2.120
190	2	0	0	282937	75302	2.440
191	2	0	0	267218	68068	2.130
192	2	0	0	250911	60838	2.160
193	2	0	0	283301	75303	2.310
194	2	0	0	283575	75302	2.300
195	3	0	0	267155	68070	2.120
196	2	3	6	60342	22920	0.680
197	2	0	0	250950	60836	2.010
198	3	0	0	266976	68071	2.110
199	3	0	0	267185	68069	2.100
200	2	0	0	266965	68070	2.770

Table A.13: Results for experiments with 10 mech. and 0 elect. faults: cases 1-100.

Inst	R	S	A	Rules	Atoms	Time
1	3	4	12	63625	18132	1.940
2	3	0	0	215212	39311	1.680
3	2	4	8	63633	18133	0.790
4	3	0	0	215192	39314	1.670
5	3	0	0	215234	39310	1.670
6	2	0	0	215233	39311	1.700
7	3	0	0	215291	39311	2.370
8	3	3	9	47243	14720	0.920
9	3	0	0	215153	39312	1.670
10	2	0	0	215262	39312	1.670
11	3	7	18	127093	28573	11.580
12	3	0	0	215104	39314	1.680
13	3	6	17	103377	25061	8.830
14	3	5	15	82291	21580	4.960
15	3	0	0	215123	39311	1.670
16	2	0	0	215167	39313	1.690
17	3	5	15	82208	21582	5.490
18	3	4	11	63644	18136	2.080
19	3	0	0	215233	39311	3.960
20	3	0	0	215161	39312	1.670
21	1	4	4	63583	18135	0.610
22	2	3	6	47250	14720	0.430
23	2	0	0	215082	39314	1.660
24	2	0	0	215320	39310	1.680
25	1	4	4	63590	18136	0.610
26	1	3	3	47243	14721	0.420
27	1	0	0	215181	39311	1.670
28	1	4	4	63619	18136	0.640
29	1	5	5	82273	21582	0.910
30	1	0	0	215216	39312	1.700
31	3	0	0	215147	39310	1.670
32	2	3	6	47245	14722	0.440
33	2	0	0	215104	39314	1.680
34	3	5	15	82269	21581	4.230
35	3	4	12	63676	18133	2.000
36	2	4	8	63576	18132	0.890
37	1	5	5	82345	21580	1.010
38	3	4	12	63631	18134	2.500
39	3	4	12	63575	18135	2.060
40	1	0	0	215087	39315	1.690
41	1	0	0	215234	39311	2.590
42	1	0	0	215279	39311	1.690
43	1	0	0	215049	39314	1.690
44	1	0	0	215199	39310	2.490
45	1	0	0	215205	39311	1.700
46	1	0	0	215114	39314	1.700
47	1	0	0	215144	39313	1.930
48	1	5	5	82229	21581	0.810
49	1	0	0	215003	39314	1.770
50	1	0	0	215161	39312	1.690

Inst	R	S	A	Rules	Atoms	Time
51	1	3	3	47242	14722	0.400
52	3	3	9	47204	14724	0.600
53	3	3	9	47269	14721	0.880
54	3	4	12	63594	18137	1.170
55	3	4	12	63603	18135	1.890
56	3	6	17	103381	25062	6.200
57	1	4	4	63579	18135	0.640
58	1	0	0	215205	39311	1.930
59	3	3	9	47217	14722	0.520
60	3	0	0	215224	39311	1.680
61	3	0	0	215291	39311	1.670
62	3	4	12	63562	18137	2.180
63	3	0	0	215181	39314	4.540
64	3	4	12	63602	18136	1.690
65	3	4	10	63656	18134	2.160
66	3	0	0	215188	39313	1.660
67	1	3	3	47226	14719	0.440
68	3	3	9	47211	14723	0.460
69	1	4	4	63610	18135	0.660
70	3	6	18	103344	25061	5.380
71	2	0	0	215262	39312	4.740
72	2	3	6	47237	14722	0.490
73	3	0	0	215105	39312	2.330
74	2	0	0	215263	39309	1.680
75	2	0	0	215239	39312	1.680
76	2	4	8	63635	18135	0.990
77	3	3	9	47186	14724	0.540
78	2	3	6	47248	14720	0.530
79	2	3	5	47258	14720	0.560
80	2	3	6	47237	14723	0.480
81	3	4	12	63656	18134	1.830
82	3	4	12	63564	18135	1.560
83	3	3	9	47199	14724	0.890
84	3	0	0	215195	39312	1.690
85	2	3	6	47285	14719	0.450
86	2	3	6	47244	14722	0.490
87	2	0	0	215207	39312	1.680
88	2	3	6	47239	14721	0.520
89	2	0	0	215104	39314	1.690
90	2	0	0	215150	39312	1.680
91	3	0	0	215190	39310	1.670
92	3	0	0	215263	39310	1.680
93	3	0	0	215207	39312	1.670
94	3	3	9	47195	14721	0.500
95	3	0	0	215152	39312	1.670
96	3	4	12	63639	18135	2.100
97	3	0	0	215233	39313	1.680
98	3	3	9	47221	14722	0.580
99	3	3	9	47158	14724	0.910
100	3	0	0	215147	39314	1.680

Table A.14: Results for experiments with 10 mech. and 0 elect. faults: cases 101-200.

Inst	R	S	A	Rules	Atoms	Time
101	3	0	0	215163	39312	1.680
102	3	0	0	215168	39310	4.470
103	3	3	9	47219	14724	0.610
104	3	4	12	63537	18137	1.800
105	3	0	0	215133	39313	1.780
106	3	0	0	215118	39312	1.770
107	3	0	0	215210	39310	1.700
108	3	0	0	215139	39311	1.770
109	3	0	0	215271	39308	1.810
110	3	4	12	63557	18137	2.960
111	2	0	0	215195	39312	1.690
112	2	0	0	215205	39312	1.690
113	2	3	6	47163	14724	0.480
114	2	3	6	47241	14723	0.550
115	3	5	10	82271	21583	465.420
116	2	0	0	215199	39313	1.680
117	2	3	6	47192	14722	0.440
118	2	0	0	215291	39311	1.680
119	2	0	0	215194	39312	1.690
120	2	3	6	47229	14720	0.430
121	2	3	6	47235	14721	0.540
122	2	0	0	215121	39313	1.680
123	2	4	8	63597	18136	0.830
124	2	4	8	63615	18134	1.010
125	2	3	6	47222	14721	0.480
126	2	0	0	215133	39313	1.680
127	2	3	6	47257	14721	0.490
128	2	5	9	82220	21582	1.560
129	2	0	0	215257	39311	1.690
130	2	0	0	215129	39312	1.690
131	2	3	5	47248	14722	0.950
132	2	4	8	63625	18135	1.360
133	2	3	6	47204	14723	0.450
134	2	4	8	63636	18135	1.390
135	2	3	6	47215	14720	0.820
136	2	3	5	47180	14724	0.550
137	2	0	0	215168	39311	1.680
138	2	3	6	47212	14723	0.510
139	2	3	6	47224	14720	0.430
140	2	0	0	215155	39313	1.680
141	2	0	0	215252	39309	1.680
142	2	0	0	215320	39310	1.700
143	2	5	9	82318	21581	1.790
144	2	0	0	215148	39311	1.680
145	2	0	0	215156	39313	1.710
146	2	0	0	215218	39312	1.670
147	2	3	6	47244	14720	0.440
148	2	3	6	47222	14721	0.460
149	2	3	6	47243	14722	0.650
150	2	0	0	215249	39312	1.710

Inst	R	S	A	Rules	Atoms	Time
151	1	0	0	215088	39313	1.700
152	1	0	0	215106	39312	1.690
153	1	0	0	215304	39309	2.010
154	1	4	4	63625	18135	0.580
155	1	0	0	215225	39311	1.700
156	1	4	4	63592	18135	0.610
157	1	0	0	215228	39312	1.700
158	1	0	0	215257	39311	1.700
159	1	5	5	82311	21580	0.890
160	1	0	0	215261	39311	1.760
161	3	0	0	215234	39311	1.730
162	1	4	4	63666	18134	0.660
163	1	0	0	215149	39311	1.690
164	1	0	0	215082	39311	1.700
165	3	0	0	215349	39309	1.680
166	3	4	12	63599	18137	1.470
167	3	0	0	215253	39310	1.670
168	3	0	0	215202	39311	1.680
169	1	0	0	215178	39313	1.690
170	1	4	4	63593	18137	0.620
171	3	0	0	215173	39312	2.020
172	3	0	0	215268	39311	1.670
173	3	0	0	215269	39311	1.680
174	3	0	0	215160	39314	1.680
175	3	4	11	63639	18136	1.910
176	3	0	0	215081	39314	1.670
177	3	4	12	63614	18135	2.050
178	1	4	4	63619	18134	0.620
179	3	0	0	215213	39311	1.670
180	3	4	11	63605	18135	9.150
181	1	0	0	214952	39315	1.690
182	1	0	0	215291	39311	1.750
183	1	4	4	63580	18133	0.590
184	1	3	3	47249	14721	0.460
185	1	4	4	63603	18136	0.620
186	1	0	0	215134	39313	1.690
187	1	4	4	63699	18132	0.610
188	1	4	4	63692	18132	0.620
189	1	7	7	126964	28575	1.930
190	1	5	5	82299	21580	0.890
191	1	3	3	47207	14722	0.460
192	1	3	1	47275	14719	0.430
193	1	0	0	215137	39314	1.930
194	2	0	0	215155	39313	1.680
195	2	3	6	47179	14724	0.590
196	2	0	0	215160	39312	1.680
197	1	0	0	215042	39312	2.030
198	2	4	8	63614	18136	0.880
199	2	3	6	47211	14720	0.460
200	1	0	0	215206	39311	1.700

Table A.15: Results for experiments with 10 mech. and 3 elect. faults: cases 1-100.

Inst	R	S	A	Rules	Atoms	Time
1	3	0	0	267086	68063	2.130
2	3	0	0	267229	68068	2.150
3	3	0	0	267102	68068	2.120
4	3	0	0	267275	68067	2.140
5	3	0	0	251071	60835	2.000
6	3	0	0	267432	68068	2.700
7	1	0	0	250875	60833	2.450
8	1	0	0	267273	68066	2.150
9	3	0	0	267245	68068	2.650
10	3	4	11	87471	31646	2.280
11	1	0	0	267175	68067	2.150
12	2	4	8	80105	28236	0.940
13	1	0	0	250790	60836	2.330
14	2	0	0	267304	68066	2.160
15	2	0	0	267104	68069	4.190
16	2	0	0	266856	68069	2.140
17	2	0	0	267139	68068	2.280
18	2	3	6	60467	22918	0.600
19	2	0	0	234715	53606	1.970
20	1	0	0	251127	60836	2.010
21	2	3	6	66417	25689	0.650
22	1	3	3	60474	22917	0.580
23	2	0	0	267311	68066	2.150
24	2	0	0	267152	68067	2.610
25	2	0	0	234653	53602	1.870
26	1	0	0	234551	53602	1.850
27	2	0	0	267217	68068	2.150
28	1	0	0	251104	60834	2.350
29	2	0	0	234930	53600	1.870
30	1	4	4	80026	28235	0.850
31	1	0	0	250951	60837	1.980
32	3	0	0	267113	68069	2.410
33	3	0	0	267246	68065	2.150
34	3	4	11	80087	28235	4.460
35	3	5	14	101909	33585	5.040
36	3	0	0	267291	68066	2.800
37	3	0	0	267162	68067	2.130
38	3	0	0	250978	60835	2.000
39	3	0	0	267191	68067	2.130
40	3	0	0	267284	68068	2.130
41	2	0	0	250858	60835	2.010
42	3	0	0	250863	60838	1.960
43	3	0	0	267233	68066	2.120
44	2	0	0	251019	60835	2.000
45	3	0	0	250756	60837	4.430
46	2	0	0	267369	68066	4.380
47	2	0	0	251034	60833	2.020
48	3	0	0	250949	60835	1.990
49	2	0	0	251010	60835	2.010
50	2	3	5	66412	25688	0.770
51	3	0	0	267178	68067	2.110
52	3	3	9	60514	22918	1.120
53	2	3	6	66448	25691	0.650
54	3	0	0	267222	68065	2.140
55	2	0	0	250800	60836	2.000
56	2	3	6	66419	25689	0.710
57	2	0	0	250922	60835	2.010
58	3	3	9	60489	22916	0.630
59	2	0	0	267434	68066	2.150
60	3	0	0	251124	60835	1.980
61	3	0	0	250979	60834	3.060
62	3	0	0	234913	53602	1.870
63	3	3	9	60424	22921	0.950
64	3	6	15	126229	38968	10.160
65	3	0	0	251054	60836	1.970
66	3	0	0	250824	60836	2.000
67	3	0	0	251116	60835	2.010
68	3	4	12	72530	24825	2.080
69	3	0	0	251138	60834	2.350
70	3	0	0	250796	60836	1.980
71	3	3	9	60463	22917	0.920
72	3	0	0	251026	60835	2.510
73	3	0	0	251056	60834	3.990
74	3	0	0	267186	68069	2.130
75	3	0	0	250989	60836	1.980
76	3	0	0	250924	60837	2.000
77	3	4	12	79953	28235	1.560
78	3	0	0	250828	60834	1.990
79	3	0	0	250664	60838	1.970
80	3	6	18	126216	38966	11.640
81	3	0	0	267110	68068	2.140
82	3	0	0	251200	60832	1.980
83	3	0	0	250965	60836	2.000
84	3	0	0	267304	68067	2.120
85	3	0	0	234836	53601	1.820
86	3	5	15	93043	29536	5.770
87	3	0	0	267118	68070	2.110
88	3	4	12	80094	28234	3.370
89	3	0	0	267040	68070	2.140
90	3	0	0	250889	60837	2.010
91	3	5	15	110959	37634	41.750
92	3	3	8	66397	25689	0.800
93	3	0	0	251233	60830	1.990
94	3	0	0	251004	60833	2.010
95	3	0	0	267267	68067	2.140
96	3	5	12	101871	33586	3.880
97	3	0	0	250951	60835	1.990
98	3	0	0	251159	60833	1.990
99	3	0	0	267272	68067	2.760
100	3	3	9	60444	22917	0.680

Table A.16: Results for experiments with 10 mech. and 3 elect. faults: cases 101-200.

Inst	R	S	A	Rules	Atoms	Time
101	3	0	0	250971	60835	1.980
102	2	3	6	60472	22913	0.580
103	2	5	10	101866	33586	3.130
104	2	0	0	250769	60837	2.010
105	2	0	0	250806	60835	1.980
106	2	3	6	60502	22916	0.620
107	3	0	0	250518	60838	1.970
108	2	3	6	60365	22919	0.560
109	2	6	12	126224	38969	4.670
110	3	4	12	87554	31643	7.810
111	2	0	0	267257	68066	2.150
112	2	0	0	267228	68067	5.540
113	2	0	0	251084	60832	2.010
114	2	0	0	251107	60836	2.020
115	3	0	0	250899	60839	2.450
116	2	0	0	250740	60838	2.140
117	2	0	0	267221	68067	2.150
118	3	3	9	60454	22918	0.670
119	2	0	0	251147	60837	1.990
120	2	4	8	80086	28235	1.440
121	2	0	0	267075	68068	2.150
122	2	3	6	66470	25689	0.610
123	2	0	0	267253	68067	2.640
124	2	0	0	267306	68068	2.140
125	2	3	6	60425	22919	0.690
126	2	0	0	266973	68070	2.310
127	2	0	0	267386	68068	2.120
128	2	0	0	250966	60836	1.990
129	2	5	10	102010	33583	1.980
130	2	0	0	267066	68068	2.150
131	2	0	0	267233	68067	2.140
132	2	0	0	267504	68063	2.160
133	2	0	0	267338	68063	2.120
134	2	0	0	251160	60835	2.010
135	2	0	0	267242	68066	2.720
136	2	3	6	60512	22913	0.560
137	2	3	6	60470	22919	0.640
138	2	0	0	250834	60836	2.010
139	2	0	0	234902	53602	1.860
140	2	0	0	250980	60834	2.020
141	3	0	0	251016	60833	2.000
142	3	0	0	250853	60838	1.990
143	3	4	12	87521	31641	6.000
144	3	0	0	267230	68070	2.150
145	3	0	0	267259	68068	2.140
146	3	0	0	251277	60833	2.490
147	3	0	0	250570	60838	1.970
148	3	0	0	250841	60838	1.990
149	3	0	0	250756	60837	2.010
150	3	4	12	72703	24825	2.500

Inst	R	S	A	Rules	Atoms	Time
151	2	0	0	251191	60833	2.000
152	3	0	0	267283	68068	2.140
153	3	3	9	60524	22918	0.930
154	3	0	0	267010	68065	2.130
155	2	0	0	267119	68065	2.120
156	3	4	12	87490	31642	1.520
157	3	0	0	267116	68068	2.340
158	2	0	0	267352	68067	2.250
159	3	0	0	267093	68067	2.130
160	3	0	0	250738	60837	1.990
161	2	4	8	80072	28232	2.760
162	2	4	8	80121	28235	0.970
163	2	6	12	136530	43656	3.180
164	2	0	0	251048	60836	2.000
165	2	0	0	250980	60834	2.010
166	2	3	6	66329	25691	0.850
167	2	0	0	251155	60832	1.990
168	2	3	6	66470	25684	1.410
169	2	0	0	250914	60837	2.020
170	2	0	0	250921	60836	2.010
171	2	3	6	60534	22913	0.750
172	2	0	0	267060	68069	2.140
173	2	0	0	267210	68067	2.130
174	2	3	6	60473	22918	0.640
175	2	4	8	87442	31645	1.680
176	2	0	0	251124	60833	1.990
177	2	0	0	250875	60835	1.970
178	2	0	0	250928	60834	2.000
179	2	0	0	266902	68070	2.150
180	2	0	0	250593	60837	2.010
181	2	0	0	251063	60835	2.010
182	2	0	0	267305	68067	2.120
183	2	0	0	234661	53604	1.850
184	2	0	0	250978	60836	2.000
185	2	0	0	251216	60834	2.430
186	2	0	0	267049	68068	2.740
187	2	3	6	54488	20143	0.630
188	2	3	6	60413	22918	0.550
189	2	0	0	267110	68069	2.140
190	2	0	0	267443	68064	2.150
191	2	0	0	251042	60836	2.020
192	2	0	0	267261	68067	2.150
193	2	0	0	267332	68064	2.660
194	2	0	0	267236	68065	2.150
195	2	5	10	101867	33586	30.110
196	2	0	0	250912	60837	2.010
197	2	0	0	267044	68068	2.160
198	2	0	0	267430	68065	2.270
199	2	0	0	250679	60837	2.010
200	2	0	0	267504	68066	2.120

Table A.17: Results for experiments with 10 mech. and 5 elect. faults: cases 1-100.

Inst	R	S	A	Rules	Atoms	Time
1	2	0	0	267109	68071	2.310
2	2	0	0	283239	75305	2.310
3	2	0	0	266931	68073	2.330
4	3	0	0	266935	68070	2.110
5	2	0	0	283232	75306	2.480
6	2	3	6	66495	25694	0.690
7	3	0	0	267081	68072	2.100
8	2	4	8	87521	31651	1.370
9	2	0	0	283394	75305	2.320
10	3	4	11	87330	31649	7.720
11	3	0	0	267126	68071	2.300
12	3	0	0	283396	75304	2.270
13	3	0	0	266972	68068	2.150
14	3	0	0	266824	68073	2.140
15	3	0	0	283285	75304	2.290
16	3	3	9	66251	25698	0.700
17	3	0	0	266996	68069	2.810
18	3	0	0	266806	68072	2.170
19	3	0	0	267050	68072	2.120
20	3	0	0	267191	68074	2.140
21	3	0	0	266710	68074	2.090
22	2	0	0	283321	75303	2.340
23	2	0	0	267101	68072	2.670
24	2	0	0	283168	75306	2.510
25	2	0	0	267055	68074	3.940
26	2	0	0	267232	68069	2.160
27	2	0	0	266908	68070	2.160
28	2	0	0	266904	68069	2.190
29	2	3	6	60388	22919	0.680
30	2	0	0	267043	68072	2.160
31	3	0	0	267120	68072	2.120
32	3	0	0	267078	68073	2.140
33	3	0	0	267211	68071	2.150
34	3	4	12	79995	28240	2.020
35	3	0	0	267328	68070	2.130
36	3	0	0	266834	68073	2.150
37	3	0	0	283114	75304	2.460
38	3	0	0	266884	68071	2.140
39	3	0	0	266899	68073	2.120
40	3	0	0	267011	68075	2.100
41	3	3	9	72298	28467	0.810
42	3	0	0	267057	68074	2.150
43	3	0	0	250725	60843	1.960
44	3	0	0	250537	60844	2.010
45	3	0	0	267000	68074	2.140
46	3	0	0	251100	60840	1.990
47	3	0	0	250925	60838	1.980
48	3	0	0	283110	75309	2.480
49	3	0	0	283138	75305	2.290
50	3	4	12	80014	28239	2.770

Inst	R	S	A	Rules	Atoms	Time
51	2	0	0	267164	68075	2.150
52	2	0	0	267149	68072	2.600
53	2	0	0	266985	68070	2.150
54	2	0	0	267060	68072	2.150
55	2	4	8	94818	35058	1.170
56	2	0	0	266814	68074	2.740
57	2	0	0	283266	75305	2.290
58	2	4	8	87371	31652	0.960
59	2	0	0	267015	68072	2.150
60	2	0	0	266978	68069	2.150
61	2	3	6	72354	28469	0.980
62	2	0	0	250711	60838	2.010
63	2	0	0	267022	68073	2.150
64	2	0	0	250650	60843	2.000
65	2	0	0	266748	68071	2.140
66	2	0	0	250721	60841	2.000
67	3	0	0	266974	68071	2.080
68	3	0	0	266639	68072	2.100
69	3	0	0	266955	68071	2.100
70	2	5	10	110776	37635	3.490
71	3	0	0	267024	68072	2.770
72	3	0	0	266897	68072	2.090
73	3	0	0	266586	68075	2.130
74	3	0	0	266859	68071	2.100
75	3	0	0	267294	68071	2.120
76	3	0	0	267175	68067	2.140
77	3	0	0	267050	68076	2.130
78	3	0	0	266699	68074	2.090
79	3	0	0	283142	75305	2.230
80	3	0	0	266848	68076	2.120
81	3	0	0	283301	75305	2.250
82	3	0	0	283526	75300	2.280
83	3	0	0	283170	75303	2.270
84	3	0	0	283181	75304	2.930
85	3	0	0	266960	68073	2.300
86	2	0	0	266839	68072	2.110
87	2	0	0	250763	60840	1.990
88	3	0	0	250664	60842	2.000
89	3	0	0	266631	68076	2.360
90	3	4	10	79991	28239	2.040
91	2	0	0	283252	75306	2.440
92	2	0	0	283646	75303	2.280
93	2	0	0	251042	60837	2.010
94	2	3	6	66383	25695	0.920
95	2	4	7	87588	31648	1.640
96	2	3	6	66339	25696	0.620
97	2	0	0	283394	75304	2.280
98	2	0	0	267054	68071	2.680
99	2	3	6	66344	25696	0.730
100	2	3	6	66290	25697	0.730

Table A.18: Results for experiments with 10 mech. and 5 elect. faults: cases 101-200.

Inst	R	S	A	Rules	Atoms	Time
101	2	0	0	250682	60838	1.990
102	2	3	6	66396	25695	0.650
103	2	3	6	66372	25694	0.660
104	2	5	8	110795	37634	5.660
105	2	0	0	266876	68076	2.480
106	2	0	0	267432	68070	2.150
107	2	0	0	266850	68069	2.150
108	2	0	0	283042	75305	2.470
109	2	0	0	283361	75301	2.290
110	2	0	0	266770	68075	2.160
111	2	0	0	266688	68076	2.140
112	2	0	0	266781	68073	2.130
113	3	0	0	250812	60839	2.000
114	3	0	0	283397	75306	2.280
115	2	3	6	66295	25695	0.620
116	3	0	0	266944	68075	2.130
117	2	0	0	250676	60842	1.980
118	2	0	0	267064	68072	2.150
119	2	0	0	267171	68073	2.250
120	3	0	0	266742	68073	2.150
121	2	3	6	66360	25698	0.650
122	2	0	0	266865	68074	2.150
123	2	3	6	66405	25694	0.680
124	2	0	0	283128	75305	2.290
125	2	4	8	87448	31648	1.330
126	2	0	0	283508	75302	2.270
127	2	5	10	110748	37634	4.670
128	2	0	0	283228	75306	2.300
129	2	0	0	266955	68073	2.530
130	2	0	0	266932	68072	2.150
131	2	0	0	283344	75306	2.820
132	2	0	0	267210	68075	2.170
133	2	0	0	267057	68074	2.160
134	2	0	0	251167	60836	1.980
135	2	0	0	267123	68069	2.150
136	2	6	12	136462	43660	4.880
137	2	0	0	266834	68073	2.120
138	2	0	0	266935	68070	2.140
139	2	0	0	266921	68073	2.160
140	2	0	0	283288	75304	2.280
141	3	0	0	283337	75306	2.270
142	2	0	0	267208	68071	2.120
143	2	0	0	283549	75305	2.260
144	3	0	0	266927	68073	2.130
145	3	3	9	66259	25696	0.980
146	2	0	0	267047	68070	2.300
147	2	0	0	267398	68070	2.130
148	3	0	0	250528	60843	1.990
149	3	0	0	267046	68075	2.140
150	3	0	0	283240	75306	2.280
151	2	0	0	267114	68072	2.110
152	3	0	0	266941	68073	2.120
153	3	0	0	250745	60840	2.010
154	2	0	0	266803	68072	2.640
155	3	0	0	282917	75304	2.300
156	3	3	8	66415	25694	1.100
157	3	0	0	267031	68072	2.150
158	3	0	0	250820	60837	2.000
159	3	0	0	250611	60842	8.790
160	3	3	9	66234	25697	0.680
161	2	0	0	267237	68067	2.120
162	2	3	6	54458	20149	0.620
163	2	3	6	60439	22922	0.660
164	2	0	0	266717	68074	2.150
165	2	0	0	283432	75302	2.250
166	2	4	8	94896	35056	1.690
167	2	0	0	267164	68073	2.130
168	2	0	0	267073	68072	2.130
169	2	0	0	283228	75303	2.290
170	2	0	0	267092	68072	2.140
171	3	0	0	250703	60840	2.000
172	3	0	0	267253	68073	2.140
173	3	0	0	266961	68069	2.140
174	3	0	0	250873	60840	2.000
175	3	0	0	266912	68074	2.100
176	3	0	0	266971	68074	2.130
177	3	0	0	250733	60843	1.990
178	3	4	11	80070	28238	2.250
179	3	0	0	250914	60841	2.000
180	3	0	0	267142	68068	2.100
181	2	3	6	54392	20151	0.620
182	2	0	0	266703	68073	2.160
183	3	0	0	250798	60840	1.990
184	2	0	0	266991	68070	2.230
185	3	0	0	267035	68070	2.100
186	2	3	6	66294	25696	0.860
187	2	0	0	267443	68072	2.160
188	2	0	0	283156	75304	2.290
189	2	0	0	283249	75303	2.300
190	3	0	0	267068	68075	2.300
191	2	0	0	267305	68074	2.150
192	3	0	0	267223	68071	2.120
193	2	0	0	250509	60840	2.000
194	2	0	0	250724	60843	2.030
195	3	0	0	266594	68076	2.100
196	3	0	0	250800	60839	1.960
197	3	0	0	267113	68072	2.320
198	2	0	0	283515	75303	2.280
199	2	0	0	283275	75310	2.440
200	2	3	6	60533	22919	0.720

Table A.19: Results for experiments with 10 mech. and 7 elect. faults: cases 1-100.

Inst	R	S	A	Rules	Atoms	Time	Inst	R	S	A	Rules	Atoms	Time
1	2	0	0	283130	75308	2.290	51	3	0	0	283034	75308	2.280
2	2	4	8	79747	28248	14.670	52	3	0	0	266785	68078	2.120
3	2	3	6	72154	28475	0.790	53	3	0	0	266795	68077	2.140
4	3	0	0	266756	68076	2.110	54	3	0	0	283198	75309	2.260
5	2	0	0	266710	68077	2.280	55	3	0	0	282819	75309	2.270
6	3	0	0	283178	75308	2.260	56	3	0	0	250569	60845	1.980
7	2	4	7	87411	31656	1.300	57	3	0	0	283008	75309	2.260
8	3	0	0	267052	68075	2.100	58	3	0	0	283280	75304	2.240
9	3	0	0	250768	60844	1.980	59	3	0	0	266666	68076	2.120
10	2	0	0	282956	75308	2.300	60	3	0	0	267006	68073	2.120
11	2	0	0	282839	75311	2.300	61	3	0	0	266786	68079	2.130
12	3	0	0	282927	75313	2.260	62	3	0	0	266695	68078	2.120
13	2	0	0	250874	60843	2.010	63	3	3	9	66227	25698	0.750
14	2	0	0	282674	75309	2.480	64	3	4	12	87233	31656	108.100
15	2	0	0	266918	68077	2.160	65	3	0	0	266703	68080	2.110
16	3	0	0	266718	68076	2.090	66	3	0	0	283092	75306	2.270
17	2	0	0	282950	75309	2.290	67	3	0	0	266977	68072	2.140
18	3	0	0	282641	75311	2.230	68	3	0	0	266867	68078	2.910
19	2	0	0	266623	68079	2.150	69	3	0	0	266677	68077	2.120
20	2	0	0	283390	75304	2.290	70	3	0	0	283177	75309	2.290
21	3	0	0	266798	68079	2.130	71	3	0	0	266809	68077	2.130
22	3	0	0	283153	75306	2.270	72	3	0	0	283298	75307	2.280
23	3	0	0	266729	68076	2.130	73	3	0	0	282766	75309	2.280
24	3	0	0	266962	68074	2.110	74	3	0	0	283614	75307	2.260
25	3	0	0	250674	60841	1.990	75	3	0	0	266883	68073	2.140
26	3	0	0	267099	68069	2.510	76	3	0	0	282684	75312	2.250
27	3	0	0	283155	75309	2.290	77	3	0	0	266815	68076	2.690
28	3	0	0	283246	75309	2.280	78	3	0	0	266821	68079	2.110
29	3	4	12	79929	28243	2.100	79	3	3	9	72141	28474	0.940
30	3	0	0	266743	68077	2.120	80	3	0	0	267187	68071	2.150
31	3	0	0	283135	75308	2.250	81	3	0	0	283246	75308	2.290
32	1	0	0	266654	68078	2.170	82	3	0	0	266637	68080	2.700
33	1	0	0	266791	68079	2.160	83	3	0	0	283330	75309	2.240
34	3	4	12	87220	31653	2.300	84	3	0	0	250744	60842	2.000
35	1	0	0	282884	75309	2.460	85	3	0	0	282842	75310	2.270
36	1	0	0	283030	75310	2.300	86	3	0	0	282944	75310	2.280
37	1	0	0	282856	75310	2.310	87	3	0	0	282741	75310	2.280
38	3	4	12	94737	35062	2.490	88	3	0	0	283312	75308	2.880
39	3	0	0	282851	75309	2.750	89	3	0	0	282803	75308	2.290
40	3	0	0	266957	68075	2.120	90	3	0	0	283051	75310	2.270
41	2	0	0	267049	68072	2.140	91	2	0	0	282698	75311	2.290
42	2	0	0	266856	68079	2.140	92	2	0	0	282971	75308	2.440
43	2	0	0	266805	68075	2.140	93	2	0	0	267037	68078	2.160
44	2	0	0	266808	68078	2.160	94	2	0	0	267099	68076	2.160
45	2	0	0	266848	68074	2.150	95	3	0	0	266703	68077	2.100
46	3	0	0	283134	75309	2.240	96	2	4	8	87283	31654	1.420
47	2	0	0	283076	75308	2.280	97	2	0	0	266877	68075	2.160
48	2	4	7	87236	31650	1.070	98	2	0	0	283260	75306	2.260
49	2	0	0	283087	75311	2.300	99	2	0	0	266880	68078	2.150
50	2	0	0	282895	75308	2.280	100	2	0	0	266783	68076	2.160

Table A.20: Results for experiments with 10 mech. and 7 elect. faults: cases 101-200.

Inst	R	S	A	Rules	Atoms	Time
101	2	3	6	60271	22928	0.670
102	2	0	0	283103	75310	2.430
103	2	0	0	266788	68079	2.250
104	2	0	0	282978	75306	2.370
105	2	0	0	282969	75311	2.270
106	2	0	0	282668	75312	2.470
107	2	4	7	94723	35061	1.100
108	2	0	0	283013	75309	2.250
109	2	0	0	283205	75311	2.320
110	2	0	0	266948	68073	2.150
111	2	0	0	266717	68077	2.120
112	2	0	0	266980	68076	2.130
113	3	0	0	267105	68079	2.730
114	3	0	0	282921	75311	5.000
115	3	0	0	283080	75310	2.280
116	3	0	0	266984	68076	2.140
117	3	0	0	266882	68078	2.150
118	3	0	0	283063	75309	2.280
119	3	0	0	266816	68076	2.130
120	2	0	0	266909	68075	2.140
121	3	3	9	66202	25698	0.660
122	3	0	0	266807	68075	2.130
123	3	0	0	283209	75306	2.280
124	3	3	8	72159	28475	1.000
125	2	0	0	266947	68075	2.120
126	1	0	0	282820	75312	2.290
127	1	0	0	283180	75307	2.290
128	1	0	0	283054	75310	2.270
129	1	0	0	283170	75306	2.540
130	3	0	0	267039	68077	2.140
131	1	4	4	94824	35064	1.000
132	1	0	0	283084	75303	2.250
133	1	0	0	266882	68074	2.140
134	1	0	0	283348	75309	2.280
135	1	0	0	282780	75311	2.310
136	1	0	0	282928	75309	2.300
137	1	0	0	282795	75310	2.300
138	1	4	4	94882	35066	1.150
139	1	0	0	283191	75306	2.450
140	1	0	0	283157	75308	2.660
141	1	0	0	282925	75313	2.270
142	3	0	0	283044	75306	2.260
143	3	0	0	266908	68075	2.110
144	3	0	0	266873	68075	2.120
145	3	0	0	267109	68079	2.750
146	3	0	0	266479	68077	2.110
147	3	0	0	266994	68073	2.130
148	3	0	0	266741	68075	2.290
149	3	0	0	266581	68077	2.130
150	3	0	0	266887	68079	2.110

Inst	R	S	A	Rules	Atoms	Time
151	3	0	0	283029	75312	2.970
152	1	0	0	282787	75311	2.310
153	3	0	0	266858	68075	2.110
154	1	4	4	87390	31652	0.860
155	1	0	0	250582	60849	2.240
156	3	0	0	266621	68079	2.130
157	3	0	0	283004	75309	2.900
158	3	0	0	266389	68080	2.100
159	1	0	0	282953	75304	2.310
160	1	0	0	266701	68081	2.150
161	3	0	0	283162	75302	2.280
162	3	0	0	283359	75309	2.280
163	3	0	0	266975	68077	2.140
164	3	0	0	282869	75310	2.470
165	3	0	0	283382	75307	2.280
166	3	0	0	283085	75309	2.940
167	3	0	0	283231	75310	2.290
168	3	0	0	250546	60847	2.000
169	3	0	0	266726	68076	2.130
170	3	0	0	266745	68074	2.130
171	3	0	0	282878	75312	2.240
172	3	0	0	283137	75308	2.270
173	3	0	0	266818	68075	2.140
174	2	0	0	283215	75306	2.270
175	3	0	0	283427	75305	2.280
176	2	0	0	283175	75307	2.250
177	2	0	0	283132	75307	2.440
178	3	0	0	283284	75307	2.280
179	3	0	0	282702	75311	2.280
180	3	0	0	266830	68075	2.130
181	1	0	0	282713	75312	2.300
182	2	0	0	283113	75308	2.270
183	1	0	0	267063	68071	2.140
184	1	0	0	282692	75312	2.280
185	1	0	0	283196	75310	2.440
186	1	0	0	266896	68075	2.150
187	1	0	0	282960	75308	2.340
188	2	0	0	266681	68077	2.120
189	1	0	0	282841	75312	2.310
190	2	0	0	250728	60843	1.990
191	2	0	0	267136	68071	2.130
192	2	0	0	266680	68075	2.140
193	2	0	0	282907	75310	2.310
194	2	0	0	266699	68078	2.170
195	2	0	0	266741	68080	2.190
196	2	0	0	282820	75310	2.270
197	2	3	6	66200	25700	0.730
198	2	0	0	266932	68077	2.170
199	2	0	0	283055	75311	2.250
200	2	0	0	283432	75307	2.300

Bibliography

[1] M. Agnes, editor in chief. *Webster's New World College Dictionary.* IDG Books Worldwide, Inc., Fourth Edition, 2001.

[2] J.J. Alferes, R. Li, and L.M. Pereira. Concurrent Actions and Changes in the Situation Calculus. In H. Geffner, editor, *Proceedings of IBERAMIA 94*, pp. 93–104, McGraw Hill, 1994.

[3] J.J. Alferes and L.M. Pereira. *Reasoning with Logic Programming.* LNAI Volume 1111, Springer-Verlag, 1996.

[4] G. Antoniou. *Nonmonotonic Reasoning.* The MIT Press, 1997.

[5] K.R. Apt, H.A. Blair, and A. Walker. Towards a theory of declarative knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pp. 89–148, Morgan Kaufmann, Los Altos, California, 1988.

[6] K.R. Apt and R. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, vols. 19–20, pp. 9–71, 1994.

[7] K.R. Apt, V.W. Marek, M. Truszczyński, and D.S. Warren, editors. *The Logic Programming Paradigm - A 25-Year Perspective.* Springer, 1999.

[8] Y. Babovich. Cmodels system. Available from http://www.cs.utexas.edu/users/tag/cmodels.html

[9] F. Baccus and F. Kabanza. Using Temporal Logic to Control Search in a Forward Chaining Planner. In M. Ghallab and A. Milano, editors, *New Directions in Planning,* pp. 141–153, IOS Press, 1996.

[10] F. Bacchus and F. Kabanza. Planning for Temporally Extended Goals. In *Annals of Mathematics and Artificial Intelligence,* 22(1-2), pp. 5–27, 1998.

[11] F. Baccus and F. Kabanza. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence*, vol. 16, pp. 123–191, 2000.

[12] M. Balduccini, J. Galloway, and M. Gelfond. Diagnosing physical systems in A-Prolog. In *Proceedings of the 25th National Conference on Artificial Intelligence (AAAI-01) - Workshop on Answer Set Programming*, pp. 77–83, 2001.

[13] M. Balduccini and M. Gelfond. Logic Programs with Consistency-Restoring Rules. In P. Doherty, J. McCarthy, and M. Williams, editors, *Proceedings of the International Symposium on Logical Formalization of Commonsense Reasoning - AAAI Spring 2003 Symposium*, March 2003.

[14] M. Balduccini, M. Gelfond, and M. Nogueira. A-Prolog as a tool for declarative programming. In *Proceedings of the Twelfth International Conference on Software Engineering and Knowledge Engineering (SEKE-2000)*, pp. 63–72, 2000.

[15] M. Balduccini, M. Gelfond, and M. Nogueira. Digital Circuits in A-Prolog. *Technical Report*, Department of Computer Science, University of Texas at El Paso, 2000.

[16] M. Balduccini, M. Gelfond, and M. Nogueira. Reasoning about Digital Circuits in A-Prolog. *Technical Report*, The University of Texas at El Paso and Texas Tech University, 2000.

[17] M. Balduccini, M. Gelfond, M. Nogueira, and R. Watson. The USA-Advisor: A Case Study in Answer Set Planning. In *Proceedings of the Sixthth International Conference on Logic Programming and Nonmonotonic Reasoning*, pp. 439–442, September 2001.

[18] M. Balduccini, M. Gelfond, M. Nogueira, and R. Watson. Planning with the USA-Advisor. In D. Kortenkamp, editor, *Third NASA International Workshop on Planning and Scheduling for Space*, October 2002.

[19] A. Baker. Nonmonotonic reasoning in the framework of situation calculus. *Artificial Intelligence*, 49(1-3), pp. 5–23, May 1991.

[20] A. Baker and M. Ginsberg. Temporal Projection and Explanation. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*, pp. 906–911, Detroit, Michigan, 1989.

[21] C. Baral. Reasoning about Actions: Non-deterministic effects, Constraints and Qualification. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pp. 2017–2023, Montreal, Canada, 1995.

[22] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003.

[23] C. Baral and M. Gelfond. Representing concurrent actions in extended logic programming. In *Proceedings of 13th International Joint Conference on Artificial Intelligence (IJCAI-93)*, pp. 866–871, Chambery, France, 1993.

[24] C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming*, vol. 12, pp. 1–80, 1994.

[25] C. Baral and M. Gelfond. Reasoning about effects of concurrent actions. *Journal of Logic Programming*, 31(1-3), pp. 85–117, 1997.

[26] C. Baral and J. Lobo. Defeasible specification in action theories. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pp. 1441-1446, Nagoya, Japan, 1997.

[27] C. Baral and L. Tuan. Effect of knowledge representation on model based planning: experiments using logic programming encodings. In A. Provetti and S.C. Tran, editors, *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning, AAAI Spring 2001 Symposium Series*, pp. 110–115, Stanford University, California, March 2001.

[28] C. Baral, T. Son, and L. Tuan. A transition function based characterization of actions with delayed and continuous effects. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR-02)*, pp. 291–302, 2002.

[29] M. Barbacci, D. Siewiorek, R. gordon, R. Howbrigg, and S. Zuckerman. An Architectural research facility — ISP Descriptions, simulation, data collection. In *Proceedings of the AFIPS National Computer Conference*, 1977.

[30] M.R. Barbacci and T. Uehara. Computer Hardware Description Languages: The Bridge Between Software and Hardware. In *IEEE Computer*, pp. 6–8, February 1985.

[31] M. Barry and R. Watson. Reasoning about actions for spacecraft redundancy management. In *Proceedings of the 1999 IEEE Aerospace Conference*, vol. 5, pp. 101–112, 1999.

[32] G.M. Baudet, M. Cutler, M. Davio, A.M. Peskin, and F.J. Ramming. The Relationship between HDLs and Programming Languages. In *VLSI and Software*

Engineering Workshop, pp. 64–69, Port Chester, NY, June 1982.

[33] G. Brewka. *Nonmonotonic reasoning: logical foundations of commonsense.* Cambridge University Press, 1991.

[34] P.W. Case, H.H. Graff, and M. Kloomok. The Recording, Checking, and Printing of Logic Diagrams. In *Proceedings of the Eastern Joint Computer Conference*, pp. 108–118, 1958.

[35] G.R. Case and J.D. Stauffer. SALOGS-IV: A Program to Perform Logic Simulation and Fault Diagnosis. In *Proceedings of the 15th Design Automation Conference*, pp. 392–397, ACM/IEEE, June 1978.

[36] S.G. Chapel and P.R. Menon. Functional Simulation in the LAMP System. *Journal of Design Automation and Fault Tolerant Computing*, pp. 203–215, May 1977.

[37] H.Y. Chang, G.W. Smith, and R.B. Walford. LAMP: System Description. *The Bell System Technical Journal*, 53(8), pp. 1431–1449, October 1974.

[38] P. Cholewinski, W. Marek, and M. Truszczyński. Default Reasoning System DeReS. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR-96)*, pp. 518–528, Morgan Kauffman, 1996.

[39] Y. Chu. An ALGOL-like Computer Design Language. *Communications of the ACM*, pp. 607–615, October 1965.

[40] Y. Chu, D.L. Dietmeyer, F. Hill, and D. Siewiorek. Introducting Computer Hardware Description Languages. *IEEE Computer*, 7(12), pp. 27–44, December 1974.

[41] S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The dlv system: Model generator and application frontends. In *Proceedings of the Twelfth Workshop on Logic Programming*, pp. 128–137, 1997.

[42] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logics and Databases*, pp. 293–322, Plenum Press, New York, 1978.

[43] P. Clark and B. Porter. *KM – the knowledge machine: Reference manual.* Technical report, AI Lab, University of Texas at Austin, 1998.

[44] A. Colmerauer. *Les Systèmes-Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur.* Technical Report 43, Department of Computer Science, University of Montreal, Quebec, Canada, 1970.

[45] A. Colmerauer, H. Kanoui, P. Roussel, and R. Pasero. *Un Système de Communication Homme-Machine en Français.* Technical Report, Groupe de Recherche en Intelligence Artificielle, Université d'Aix-Marseille II, Luminy, France, 1973.

[46] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3), pp. 201-215, July 1960.

[47] D.L. Dietmeyer. Introducing DDL. *IEEE Computer*, 7(12), pp. 34–38, December 1974.

[48] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. In *Lecture Notes in Artificial Intelligence - Recent Advances in AI Planning, Proceedings of the Fourth European Conference on Planning (ECP-97)*, vol. 1348, pp. 169–181, 1997.

[49] P. Doherty. *Notes on PMON circumscription.* Technical Report, LiTH-IDA-R-94-43, Computer Science Department, Linköping University, Linköping, Sweden, 1983.

[50] P. Doherty. Reasoning about Action and Change using Occlusion. In *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI-94)*, pp. 401–405, Amsterdam, The Netherlands, August 1994.

[51] P. Doherty and J. Kvarnstrm. Tackling the Qualification Problem using Fluent Dependency Constraints: Preliminary Report. In *Proceedings of the Fifth International Workshop on the Temporal Representation and Reasoning (TIME-98)*, pp. 97-104, Sanibel Island, Florida, 1998.

[52] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Databases. *ACM Transactions on Database Systems*, 22(3), pp. 364–418, September 1997.

[53] T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The DLV System. In J. Minker, editor, *Workshop on Logic-Based*

Artificial Intelligence, Computer Science Department, College Park, Washington, D.C., Maryland, June 1999.

[54] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge–State Planning, II: The DLV-K System. *Artificial Intelligence*, 144(1-2), pp. 157–211, 2003.

[55] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System dlv: Progress Report, Comparisons and Benchmarks. In A.G. Cohn, L. Schubert, and S.C. Shapiro, editors, *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR-98)*, pp. 406–417, Morgan Kaufmann, 1998.

[56] T. Eiter, J. Lu, and V.S. Subrahmanian. *A first-order representation of stable models*. Technical Report, IFIG, Universität Giessen, 1998. (Also in *The European Journal on Artificial Intelligence (AI Communications)*, 11(1), pp. 53–73, IOS Press, 1998.)

[57] C. Elkan. Reasoning about Action in First-Order Logic. In *Proceedings of the Ninth Biennial Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI-92)*, Morgan Kaufmann, Vancouver, Canada, May 1992.

[58] E. Erdem. *Theory and applications of answer set programming.* Ph.D. Dissertation, Department of Computer Sciences, The University of Texas at Austin, Texas, 2002.

[59] E. Erdem and V. Lifschitz. Transitive Closure, Answer Sets and Predicate Completion. In A. Provetti and S.C. Tran, editors, *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning, AAAI Spring 2001 Symposium Series*, pp. 60–65, Stanford University, CA, 2001.

[60] E. Erdem, V. Lifschitz, L. Nakhleh, and D. Ringe. Reconstructing the evolutionary history of Indo-European languages using answer set programming. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL'03)*, pp. 160–176, 2003.

[61] E. Erdem, V. Lifschitz, and M. Wong. Wire routing and satisfiability planning. In *Proceedings of the First International Conference on Computational Logic (CL 2000)*, pp. 822–836, London, U.K., 2000.

[62] K. Eshghi. *Diagnoses As Stable Models.* Technical Report, Hewlett Packard Laboratories, 1990.

[63] D. Etherington, R. Mercer, and R. Reiter. On the Adequacy of Circumscription for Closed-World Reasoning. *Computational Intelligence*, vol. 1, pp. 11–15, 1985.

[64] W. Faber, N. Leone, and G. Pfeifer. Optimizing the Computation of Heuristics for Answer Set Programming Systems. In T. Eiter, W. Faber, and M. Truszczyński, editors, *Lecture Notes in Artificial Intelligence - Proceedings of*

the 6th International Conference in Logic Programming and Nonmonotonic Reasoning (LPNMR-01), Vienna, Austria, vol. 2173, pp. 288–301, Springer Verlag, September 2001.

[65] R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2, pp. 189–208, 1971.

[66] J.J. Finger. *Exploiting Constraints in Design Synthesis.* PhD Thesis, Department of Computer Science, Stanford University, California, 1987.

[67] R.A. Finkel, V.W. Marek, N. Moore, and M. Truszczyński. Computing Stable Models in Parallel. In A. Provetti and S.C. Tran, editors, *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning, AAAI Spring 2001 Symposium Series*, pp. 72–76, Stanford University, California, March 2001.

[68] A. Finzi, F. Pirri, and R. Reiter. Open World Planning in the Situation Calculus. In *Proccedings of the 17th National Conference of Artificial Intelligence (AAAI-00)*, pp. 754–760, 2000.

[69] J.R. Galloway. *Diagnosing Dynamic Systems in A-Prolog.* Master's Thesis, Computer Science Department, The University of Texas at El Paso, Texas, 2000.

[70] M. Gelfond. Autoepistemic logic and formalization of commonsense reasoning, In M Reinfrank, J. de Kleer, M. Ginsberg, and E. Sandewall, editors, *Lecture*

Notes in Artificial Intelligence - Non-Monotonic Reasoning: Second International Workshop, vol. 346, pp. 176–186, Springer-Verlag, 1989.

[71] M. Gelfond. Representing Knowledge in A-Prolog, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski*, volume 2408, Part II, pp. 413–451, Springer-Verlag, Berlin, 2002.

[72] M. Gelfond and A. Gabaldon. From Functional Specifications to Logic Programs. In J. Maluszynski, editor, *Proceedings of the International Logic Programming Symposium (ILPS-97)*, pp. 355–369, Port Jefferson, Long Island, N.Y., October 1997.

[73] M. Gelfond and N. Leone. Logic Programming and Knowledge Representation - An A-Prolog perspective. In *Artificial Intelligence*, 138(1-2), pp. 3–38, June 2002.

[74] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. In *Proceedings of the Fifth International Conference on Logic Programming*, pp. 1070–1080, 1988.

[75] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3/4), pp. 365–386, 1991.

[76] M. Gelfond and V. Lifchitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17(2-4), pp. 301–322, 1993.

[77] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 3(16), 1998.

[78] M. Gelfond, V. Lifschitz, H. Przymusińska, and M. Truszczyński. Disjunctive defaults. In J. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference*, pp. 230–237, Morgan Kaufmann, 1991.

[79] M. Gelfond, V. Lifschitz, and A. Rabinov. What are the limitations of the situation calculus? In *Working Notes of the AAAI Spring Symposium on the Logical Formalizations of Commonsense*, pp. 59–69, Stanford University, AAAI Press, Menlo Park, California, 1991.

[80] M. Gelfond and T. Son. Reasoning with Prioritized Defaults. In J. Dix, L.M. Pereira, and T. Przymusinski, editors, *Lecture Notes in Artificial Intelligence - Selected Papers from the Workshop on Logic Programming and Knowledge Representation 1997*, vol. 1471, pp 164–224, 1998.

[81] M. Gelfond, and R. Watson. On methodology for representing knowledge in dynamic domains. In *Proc of the 1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems*, pp. 57–66, 1999.

[82] M. Gelfond and R. Watson. Diagnostics with answer sets: Dealing with unobservable fluents. In *Proceedings of the Third International Workshop on Cognitive Robotics (CogRob-02)*. To appear, 2002.

[83] M.R. Genesereth and M.L. Ginsberg. Logic programming. *Communications of the ACM*, 28(9), pp. 933–941, September 1985.

[84] Sumit Ghosh. *From Hardware Description Languages: Concepts and Principles.* In IEEE Press Series on Microeoeletronic Systems, New York, NY, 2000.

[85] P.C. Gilmore. A Proof Method for Quantification Theory. *IBM Journal Research and Development*, vol. 4, pp. 28–35, 1960.

[86] M.L. Ginsberg and D.E. Smith. Possible Worlds and the Qualification Problem. In *Proceedings of the Sixth National conference on Artificial Intelligence (AAAI-87)*, pp. 212–217, 1987.

[87] M.L. Ginsberg and D.E. Smith. Reasoning about Action I: A Possible Worlds Approach. *Artificial Intelligence*, 35(2), pp. 165–195, 1988.

[88] M.L. Ginsberg and D.E. Smith. Reasoning About Action II: The Qualification Problem. *Artificial Intelligence*, 35(3), pp. 311–342, 1988.

[89] G. Gottlob, S. Marcus, A. Nerode, G. Salzer, and V.S. Subrahmanian. A non-ground realization of the stable and well-founded semantics. *Theoretical Computer Science*, 166(1-2), pp. 221–262, 1996.

[90] C.C. Green. Theorem Proving by Resolution as a Basis for Question-Answering Systems. In B. Meltzer D. Michie, editors, *Machine Intelligence*, vol. 4, pp. 183–205, Edinburgh University Press, Edinburgh, U.K., 1969.

[91] C.C. Green. Application of theorem-proving to problem solving. In D.E. Walker and L.M. Norton, editors, *Proceedings of the First International Joint Conference on Artificial Intelligence (IJCAI-69)*, pp. 219–239, Washigton, D.C., 1969.

[92] J. Gustafsson and P. Doherty. Embracing Occlusion in Specifying the Indirect Effects of Actions. In L. Aiello, J. Doyle, and S. Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR-96)*, Morgan Kauffman, 1996.

[93] S. Hanks and D. McDermott. Nonmonotonic Logic and Temporal Projection. *Artificial Intelligence*, 33(3), pp. 379–412, 1987.

[94] P. Hayes. Computation and Deduction. In *Proceedings of the Second Symposium on Mathematical Foundations of Computer Science*, pp. 105–118, Czechoslovak Academy of Sciences, Czechoslovakia, 1973.

[95] M. Heidt. *Developing an inference engine for ASET-Prolog.* Master's Thesis, Computer Science Department, The University of Texas at El Paso, Texas, 2001.

[96] D. Hill. *Adlib Users Manual.* Technical Report 177, Computer Systems Lab., Stanford University, CA, 1979.

[97] D. Hill. *Language and Environment for Multi-level Simulation.* Techical Report 185, Computer Systems Lab., Stanford University, California, 1980.

[98] F.J. Hill and G.R. Peterson. *Digital Systems: hardware Organization and Design*, Chapter Introduction, pp. 5–6, John Wiley and Sons, New York, 2 edition, 1978.

[99] Y. Huang, H. Kautz, and B. Selman. Control Knowledge in Planning: Benefits and Tradeoffs. In *Proccedings of the 16th National Conference of Artificial Intelligence (AAAI-99) and 11th Conference on Innovative Applications of Artificial Intelligence*, pp. 511–517, AAAI Press/The MIT Press, 1999.

[100] Y.C. Huang, B. Selman, and H.A. Kautz. Learning Declarative Control Rules for Constraint-Based Planning. In P. Langley, editor, *Proceedings of the Seventeenth International Conference on Machine Learning (ICML-00)*, Stanford University, Stanford, California, Morgan Kaufmann, pp. 415–422, 2000.

[101] The Institute of Electrical and Electronic Engineers. *IEEE Standard VHDL Language Reference Manaual.* ANSI/IEEE Std 1076–1993, IEEE, Institute of Electrical and Electronic Engineers, Inc., New York, NY, April 14 1994.

[102] L. Karlsson, J. Gustafsson, and P. Doherty. Delayed Effects of Actions. In *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, Brighton, August 1998.

[103] G. Kartha and V. Lifschitz. Actions with indirect effects: Preliminary report. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR-94)*, pp. 341–350, 1994.

[104] H. Kautz and B. Selman. The Role of Domain-Specific Axioms in the Planning as Satisfiability Framework. In *Proceedings of AIPS-98*, pp. 181–189, 1998.

[105] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill Computer Science Series, 1978.

[106] J. Kvarnstrom and P. Doherty. Tackling the Qualification Problem using Dependency Constraints. *Computational Intelligence*, 16(2), pp.169–209, 2000.

[107] R.A. Kowalski. The predicate calculus as a programming language. In *Proceedings of the International Symposium and Summer School on Mathematical Foundations of Computer Science*, Jablonna, Polland, August 1972.

[108] R.A. Kowalski. *Predicate logic as a programming language.* DCL Memo 70, School of Artificial Intelligence, University of Edinburgh, U.K., November 1973. (Also in Proceedings of Information Processing (IFIP-74), pp. 569–574, Stockholm, North-Holland, Amsterdan, 1974.

[109] R.A. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7), pp. 424–436, July 1979.

[110] R.A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1), pp. 38–43, January 1988.

[111] N. Leone, R. Rosati, and F. Scarcello. Enhancing Answer Set Planning. In A. Cimatti, H. Geffner, E. Giunchiglia, and J. Rintanen, editors, *IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information*, pp. 33–42, August 2001.

[112] H. Levesque and R. Scherl. The Frame Problem and Knowledge Producing Actions. In *Proceedings of the 10th National Conference of Artificial Intelligence (AAAI 93*, pp. 689–695, 1993.

[113] H. Levesque and R. Scherl. Knowledge Producing Actions. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR94)*, pp. 1139–1146, 1994.

[114] V. Lifschitz. Computing circumscription. In *Proceedings of the Nineth International Joint Conference on Artificial Intelligence (IJCA1-85)*, Morgan-Kaufmann, San Mateo, California, 1985.

[115] V. Lifschitz. Formal theories of action. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI-87)*, pp. 966–972, Milan, Italy, 1987.

[116] V. Lifschitz. On the declarative semantics of logic programs with negation. *Readings in nonmonotonic reasoning*, Morgan Kaufmann Publishers Inc., San Francisco, California, 1987.

[117] V. Lifschitz. Circumscription. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *The Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 3: Nonmonotonic Reasoning and Uncertain Reasoning*, pp. 297–352, Oxford University Press, 1994.

[118] V. Lifschitz. Foundations of logic programming. In G. Brewka, editor, *Principles of Knowledge Representation*, pp. 69–128, CSLI Publications, 1996.

[119] V. Lifschitz. Two components of an action language. In *Proceedings of Common Sense 96*, 1996.

[120] V. Lifschitz. Action languages, Answer Sets, and Planning. In *The Logic Programming Paradigm: a 25-Year Perspective*, pp. 357–373, Spring-Verlag, 1999.

[121] V. Lifschitz. Answer set programming and plan generation. In *Artificial Intelligence*, 138(1-2), pp. 39–54, June 2002.

[122] V. Lifschitz and H. Turner. Splitting a logic program. In P. van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming*, pp. 23–37, 1994.

[123] V. Lifschitz and H. Turner. Representing transition systems by logic programs. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*, pp. 92–106, 1999.

[124] F. Lin. Embracing causality in specifying the indirect effects of actions. *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pp. 1985-1991, Montreal, Canada, 1995.

[125] F. Lin. Embracing causality in specifying the indeterminate effects of actions. *Proceedings of the 13th National Conference of Artificial Intelligence (AAAI-96*, pp. 670–676, 1996.

[126] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag Symbolic Computation Series, second edition, 1987.

[127] R. MacGregor and R. Bates. *The LOOM knowledge representation language.* Technical Report ISI-RS-87-188, ISI, California, 1987.

[128] V. Marek and V.S. Subrahmanian. The relationship between logic program semantics and non-monotonic reasoning. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pp. 600–617, 1989.

[129] V. Marek and M. Truszczyński. Stable semantics for logic programs and default theories. In *Proceedings of the North American Conference on Logic Programming*, pp. 243–256, MIT Press, 1989.

[130] V. Marek and M. Truszczyński. *Nonmonotonic Logic.* Springer, 1993.

[131] V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pp. 375–398. Springer-Verlag, 1999.

[132] N. McCain and H. Turner. A causal theory of ramifications and qualifications. *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pp. 1978–1984, Montreal, Canada, 1995.

[133] N. McCain and H. Turner. A causal theory of action and change. In *Proceedings of the 14th National Conference of Artificial Intelligence (AAAI-97)*, pp. 460–465, 1997.

[134] J. McCarthy. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes*, pp. 75–91, London, U.K., 1959. Her Majesty Stationery Office. (Also in M. Minsky, editor, Semantic Information processing, pp. 403–418. MIT, Cambridge, 1960.)

[135] J. McCarthy. Epistemological problems of artificial intelligence. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77)*, pp. 1038–1044, Cambridge, Massachusetts, 1977.

[136] J. McCarthy. Circumscription – a form of non-monotonic reasoning. *Artificial Intelligence*, vol. 13(1-2), pp. 27–39, 1980.

[137] J. McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 26(3), pp. 89–116, 1986.

[138] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence*, vol. 4, pp. 463–502, Edinburgh University Press, Edinburgh, 1969.

[139] E.J. McCluskey. *Introduction to the Theory of Switching Circuits.* McGraw-Hill, 1965.

[140] D. McDermott. Nonmonotonic logic II: nonmonotonic modal theories. *Journal of the ACM*, 29(1), pp. 33–57, 1982.

[141] D. McDermott and J. Doyle. Nonmonotonic Logic I. *Artificial Intelligence*, 13(1-2), pp. 41–72, 1980.

[142] V. Mellarkod. *Optimizing The Computation Of Stable Models Using Merged Rules*, Master's Thesis, Department of Computer Science, Texas Tech University, Texas, May 2002.

[143] M. Mendler. Timing analysis of combinational circuits in intuitionistic propositional logic. *Formal Methods in System Design*, 2000. (A short preliminary version was presented at *TABLEAUX'96*, Lecture Notes in Artifical Intelligence, vol. 1071, pp. 261–277, Springer, 1996.)

[144] M. Mendler. Characterising Timing Analyses in Intuitionistic Modal Logic. *Logic Journal of the IGPL*, 2000.

[145] S.A. McIlraith. A Closed-Form Solution to the Ramification Problem (Sometimes). In *Proceedings of the IJCAI'97 Workshop on Nonmonotonic Reasoning Action and Change*, pp. 103–126, Nagoya, Japan, August 1997.

[146] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Series in Electrical and Computer Engineering, 1994.

[147] J. Minker. An overview of nonmonotonic reasoning and logic programming. *Journal of Logic Programming*, 17(2-4), pp. 95–126, 1993.

[148] M. Minsky. A Framework for Representing Knowledge. In P. Winston, editor, *The Psych. of computer vision*, pp. 211–277, McGraw-Hill, NY, 1975. Reprinted in: R. Brachman and H. Levesque, editors, *Readings in Knowledge Representation,* pp. 245–262, 1985.

[149] R.C. Moore. Possible-world semantics for autoepistemic logic. In R. Reiter, editor, *Proceedings of the Workshop on Nonmonotonic Reasoning*, pp. 344–354, 1984. Reprinted in: M. Ginsberg, editor, *Readings on nonmonotonic reasoning*, pp. 137-142, Morgan Kaufmann, 1990.

[150] R.C. Moore. Semantical Considerations on Nonmonotonic Logic. *Artificial Intelligence*, 25(1), pp. 75–94, 1985.

[151] The *National Aeronautics and Space Administration (NASA)* web site located at http://www.nasa.org, 2003.

[152] I. Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. In *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pp. 72–79, 1998.

[153] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of Joint International Conference and Symposium on Logic Programming*, pp. 289–303, MIT Press, 1996.

[154] I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the Fourth International Conference on Logic Programming and Non-Monotonic Reasoning*, pp. 420–429, 1997.

[155] I. Niemelä and P. Simons. Extending the Smodels System with Cardinality and Weight Constraints. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pp. 491–521, Kluwer Academic Publishers, 2000.

[156] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: a system for answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, Breckenridge, Colorado, USA, April 2000.

[157] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-Prolog decision support system for the Space Shuttle. In A. Provetti and S.C.

Tran, editors, *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning, AAAI Spring 2001 Symposium Series*, pp. 139–145, Stanford University, California, March 2001.

[158] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-Prolog decision support system for the Space Shuttle. In *Lecture Notes in Computer Science - Proceedings of Practical Aspects of Declarative Languages (PADL-01)*, vol. 1990, pp. 169–183, 2001.

[159] Open Verilog International. *Verilog HDL Language Reference Manual (LRM).*

[160] S. Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis,* SunSoft Press, Prentice Hall, 1996.

[161] E. Pednault. ADL: exploring the middle ground between STRIPS and the situation calculus. In R. Brachman, H. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR-89)*, pp. 324–332, 1989.

[162] J. Pinto and R. Reiter. Reasoning about Time in the Situation Calculus. *Annals of Mathematics and Artificial Intelligence*, 14(2-4), pp. 251–268, September, 1995.

[163] D. Poole, A. Mackworth, and R. Goebel. *Computational Intelligence - a logical approach.* Oxford University Press, 1998.

[164] E. Pontelli, M. Balduccini, and F. Bermudez. Non-monotonic Reasoning on Beowulf Platforms. In V. Dahl and P. Wadler, editors, *Lecture Notes in Artificial Intelligence - Proceedings of Practical Aspects of Declarative Languages (PADL-03)*, vol. 2562, pp. 37–57, January 2003.

[165] E. Pontelli and O. El-Khatib. Exploiting Vertical Parallelism from Answer Set Programs. In A. Provetti and S.C. Tran, editors, *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning, AAAI Spring 2001 Symposium Series*, pp. 174–180, Stanford University, California, March 2001.

[166] D. Prawitz. An Improved Proof Procedure. *Theoria*, vol. 26, pp. 102–139, 1960.

[167] T. Przymusinski. The Well-Founded Semantics Coincides With The Three-Valued Stable Semantics, *Fundamenta Informaticae*, vol. 13, pp. 445–464, 1990.

[168] R. Reiter. On Closed World Data Bases. In H. Gallaire and J. Minker, editors, *Logic and Databases,* pp. 55-76, Plenum, 1978.

[169] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2), pp. 81–132, 1980.

[170] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1), pp. 57–95, 1987.

[171] R. Reiter. The Frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John Mccarthy*, pp. 359–380, Academic Press, 1991.

[172] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In L.C. Aiello, J. Doyle, and S.C. Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR-96)*, pp. 2–13. Morgan Kaufmann, 1996.

[173] J.A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1), pp. 23–41, January 1965.

[174] J.A. Robinson. Logic and logic programming. *Communications of the ACM*, 35(3), pp. 40–65, March 1992.

[175] C.H. Roth, Jr. *Fundamentals of Logic Design*, West Publishing Company, 1992.

[176] C.H. Roth, Jr. *Digital Systems Design Using VHDL.* PWS Publishing Company, Boston, MA, 1998.

[177] P. Roussel. *PROLOG: manuel de Reference et d'Utilization.* Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, 1975.

[178] K. Sagonas, T. Swift, and D.S. Warren. XSB as an Efficient Deductive Database Engine. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 442–453, 1994.

[179] E. Sandewall. *Systematic comparison of approaches to ramification using restricted minimization of change.* IDA Techical Report, LiTH-IDA-R-95-15, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 1995.

[180] M. Shanahan. *Solving the frame problem: a mathematical investigation of the common sense law of inertia*, MIT Press, Cambridge, MA, 1997.

[181] C.E. Shannon. A Symbolic Analysis of Relay and Switching Circuits. *Transactions of AIEE*, vol. 57, pp.713–723, 1938.

[182] C.E. Shannon. The Synthesis of Two-terminal Switching Circuits. *Bell System Tech. Journal*, vol. 28, pp. 59–98, 1949.

[183] P. Simons. *Extending and Implementing the Stable Model Semantics.* Doctoral dissertation. Research Report 58, Helsinki University of Technology, Helsinki, Finland, April 2000.

[184] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of Practical Aspects of Declarative Languages (PADL-99)*, pp. 305–319, Springer-Verlag, 1999.

[185] T. Son, C. Baral, and S. McIlraith. Planning with domain-dependent knowledge of different kinds – an answer set programming approach. In T. Eiter, W. Faber, and M. Truszczyński, editors, *Lecture Notes in Artificial Intelligence - Proceedings of the 6th International Conference in Logic Programming and Non-monotonic Reasoning (LPNMR-01)*, Vienna, Austria, vol. 2173, pp. 226–239, Springer Verlag, September 2001.

[186] E. Sternheim, R. Singh, and Y. Trivedi. *Digital Design with Verilog HDL*, Automata Publishing Company, Cupertino, CA, 1990.

[187] S.Y.H. Su. A Survey of Computer Hardware Description Languages in the U.S.A., *IEEE Computer*, Dec 1974.

[188] V.S. Subrahmanian and C. Zaniolo. Relating Stable Models and AI Planning Domains. In *Proceedings of the Twelfth International Conference on Logic Programming*, Tokyo, Japan, pp. 233–247, June 1995.

[189] T. Syrjänen. *Lparse 1.0.11 User's Manual.* Available at http://www.tcs.hut.fi/Software/smodels, 2003.

[190] S.A. Szygenda and A.A. Lekkos. Integrated Techniques for Functional and Gate Level Digital Logic Simulation. In *Proceedings of the 10th Design Automation Conference*, pp. 159–172, 1973.

[191] S.A. Szygenda and E.W. Thompson. Digital Logic Simulation in a Time-Based Table-Driven Environment, Part I: Design Verification. *IEEE Computer*, 8(3), pp. 24–40, March 1975.

[192] M. Thielscher. Representing actions in equational logic programming. *Proceedings of the International Conference of Logic Programming*, 1994.

[193] M. Thielscher. *Ramification and Causality.* Technical Report TR-96-003, International Computer Science Institute (ICSI), Berkeley, CA, January 1996.

[194] H. Turner. Representing actions in logic programs and default theories: A situation calculus approach. *Journal of Logic Programming*, 31(1-3), pp. 245–298, 1997.

[195] The *United Space Alliance (USA)* web site located at http://www.unitedspacealliance.com, 2003.

[196] M. van Emden and R. Kowaslski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4), pp. 733-742, 1976. (Also: DCL Memo 73, School of Artificial Intelligence, University of Edinburgh, U.K., February 1974.)

[197] A. van Gelder, K.A. Ross, and J.S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3), pp. 620–650, 1991.

[198] W.M. vanCleemput. A Hierarchical Language for the Structural Description of Digital Systems. In *Proceedings of the 14th Design Automation Conference*, pp. 378–385, ACM/IEEE, New Orleans, June 1977.

[199] W.M. vanCleemput. Computer Hardware Languages and their Applications. In *Proceedings of the 16th Design Automation Conference*, pp. 554–560, ACM/IEEE San Diego, California, June 1979.

[200] W.M. vanCleemput and H. Ofek. Design Automation for Digital Systems. *IEEE Computer*, pp. 114–122, October 1984.

[201] J.F. Wakerly. *Digital Design Principles and Practices*, Prentice Hall, 1994.

[202] R. Watson. *Action Languages and Domain Modeling.* Ph.D. Dissertation, Computer Science Department, The University of Texas at El Paso, Texas, 1999.

[203] R. Watson. An application of action theory to the space shuttle. In G. Gupta, editor, *Lecture Notes in Computer Science – Proceedings of Practical Aspects of Declarative Languages (PADL-99)*, vol. 1551, pp. 290–304, 1999.

[204] J.C. Weber. On the representation of concurrent actions in the situation calculus. In *Proceedings of the Eighth Biennial Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI-90)*, pp. 28–32, University of Ottawa, Ottawa, Canada, July 1990.

CURRICULUM VITAE

Monica de Lima Nogueira was born on January 3, 1961 in Manaus, Amazonas, Brazil, the first among four daughters of Ivens José de Lima and Maria do Carmo Marques de Lima. She is married to Antonio Geraldo Queiroz Nogueira and together they have three children; the twins, Heloisa and Marcus Vinicus, and a younger daughter, Daniela.

Before coming to the United States, Monica received her M.Sc. in Computer Science from the University of Campinas, São Paulo, Brazil, in 1989. Prior to that, she received a B.S. in Electronic Engineering from the University of Technology of Amazonia, Manaus, Brazil, in 1982, and a B.S. in Electrical Engineering from the Federal University of Amazonas, Manaus, Brazil, in 1983. She taught Computer Science classes as a lecturer at the University of Piracicaba, University of Campinas, and the Federal University of Amazonas.

Monica began her doctoral program at the University of Texas at El Paso (UTEP) in 1996. She worked as a teaching assistant and as a research assistant in the area of artificial intelligence at the Computer Science Department at UTEP for several years, where she also lectured Artificial Intelligence in the fall of 2000. In the summer of 2000, she worked as a research assistant at Texas Tech. University. In 2001, she was named a National Science Foundation Scholar.

Her research interests include reasoning about actions and change, nonmonotonic reasoning, knowledge representation, answer set programming, robotics, and digital design. In 1996, she was part of the UTEP-robotic team which won the third place in the Robot Competition of the Thirteenth National Conference on Artificial Intelligence. In 2003, she was choosen as the outstanding graduate in computer science at UTEP. She is a member of the Upsilon Pi Epsilon honors society, the American Association of Artificial Intelligence, and the Association for Computing Machinery.

Permanent address: 6240 Brisa Del Mar Dr.

El Paso, TX 79912

This dissertation was typeset by Monica Nogueira using LaTeX 2_ε.