Pandora: Non-Deterministic Parallel Logic Programming

by

Reem Bahgat

February 1991

A thesis submitted to the University of London for the degree of Doctor of Philosophy (PhD)
Wa-Ma-Tawfsiki-Illa-Billah
To My Parents
I consider myself a privileged student. I had two remarkable supervisors: Keith Clark and Steve Gregory. I am grateful to both of them. Keith Clark with his vision and vast knowledge, as well as his continuous challenge to the results, was a major driving force to achieve the contributions of the thesis. Working and brainstorming with Steve Gregory was a motivating and profitable experience. I hope that our partnership as well as our friendship will last for years to come.

My first meeting with David Warren and Rong Yang in 1987 had a great influence on my work. We discussed the advantages and disadvantages of the "exclusive relation" in P-Prolog, based on which the basic Andorra model was later on developed.

I am also grateful to Seif Haridi and David Warren for inviting me to several PEPMA workshops in Sweden and Bristol which gave me the opportunity to discuss the work with members of their groups.

I had long discussions with Jim Crammond concerning the design of the Pandora abstract machine; thank you Jim.

I must also thank a number of people who read earlier drafts of the thesis and supplied extremely informative comments. They are (in alphabetical order): Keith Clark, Andrew Davison, Steve Gregory, Bob Kemp, and Evan Tick. Special thanks to Andrew Davison for his time and patience in reading even the final draft of the thesis.

I believe that the Department of Computing at Imperial College and the logic programming group in the department were the right choice. It was a growing experience to spend the last four years at this place.

My research has been supported by a scholarship from University of London as well as an O.R.S. award, to whom I am most grateful.

Last but not least, I would like to express my gratitude first to my parents and then to my friends who tolerated my ups and downs in the last four years.
Welcome to the Legend of

PANDORA

All the secrets were revealed except One

Hope.....

It was calling, calling...
Deep, deep in the box,
I reached far down to find it,
I worked hard to shape it,
and now to you I present it....
Abstract

Most of the logic programming languages that have been seriously implemented and used to date fall into one of two categories: (a) variants of Prolog, and (b) the committed-choice logic programming languages, such as Parlog. The basic distinction is that Prolog features don't-know non-determinism, the ability to search through multiple solutions to a relation; the committed-choice languages provide stream and-parallelism, whereby conjoined goals are evaluated concurrently and may communicate incrementally by bindings to shared variables, but are based on committed-choice non-determinism. Earlier attempts to combine both categories into one language have revealed new levels of complexity and seemed infeasible to implement.

In this thesis, we propose a new parallel logic programming language called Pandora (Parlog and Andorra), whose operational semantics is a generalization of Warren’s Andorra model. Pandora extends Parlog with a deadlock handling mechanism and a simple non-deterministic fork primitive that can be implemented with reasonable efficiency.

In addition to subsuming the applications in which logic programming is used at present, Pandora provides a programming paradigm of (don't-know) non-deterministic concurrent, communicating processes, which opens up interesting application areas that cannot conveniently be expressed in existing logic programming languages. We describe the use of Pandora for constraint programming, solving resource allocation problems, and distributed discrete event simulation.

Moreover, Pandora provides a meta-level deadlock handler relation by which the user can explicitly program the behaviour on deadlock. One way in which the deadlock handler relation can be used is in combination with the non-deterministic fork, to implement (application-dependent) heuristic search. More generally, the deadlock handler relation can be utilized to manipulate suspended goals at the time of deadlock in a flexible manner. Redundant goals can be removed, new goals can be added, and a group of goals can be replaced by a simpler group.
A prototype implementation of the language was developed on top of Parlog, and a multi-processor abstract machine was designed. The design gained from the existing techniques that were developed and used for the implementation of existing logic programming languages, and also solved the new issues arising from combining stream and-parallelism and don't-know non-determinism.
# Table of Contents

## 1. Introduction
- 1.1. Evolution of Programming Languages 8
- 1.2. Non-Determinism in Logic Programming 9
- 1.3. Parallel Logic Programming 10
- 1.4. Context of Research 13
- 1.5. Contributions 14
- 1.6. Preview of Contents 15

## 2. Logic Programming
- 2.1. Syntax of Logic Programs 18
- 2.2. Semantics of Logic Programs 19
  - 2.2.1. Unification 20
  - 2.2.2. SLD Resolution 21
- 2.3. Non-Determinism 22
  - 2.3.1. Or-Nondeterminism 22
  - 2.3.2. And-Nondeterminism 24
    - 2.3.2.1. "Generate and Test" Evaluation 25
    - 2.3.2.2. "Incremental Generate and Test" Evaluation 26
    - 2.3.2.3. Restricted And-Parallelism 28
    - 2.3.2.4. Stream And-Parallelism 28
- 2.4. Committed-Choice Logic Programming Languages 29
- 2.5. Semantics of Committed-Choice Programs 31
  - 2.5.1. Standard Form of Parlog Clauses 33
  - 2.5.2. Parlog Goal Resolution 34
- 2.6. Don't-Care Non-Determinism in Committed-Choice Programs 37
- 2.7. Stream And-Parallelism in Committed-Choice Programs 38
- 2.9. Combining Stream And-Parallelism and Don't-Know Non-Determinism 42
  - 2.9.1. Parlog's Set Constructor Primitives 43
7.2. PANDA's Main Strategy 145
7.3. The PANDA Compiler 147
7.4. The PANDA Run-Time System 149
  7.4.1. The And-Parallel Module 150
    7.4.1.1. Reducing Goals in the And-Parallel Phase 150
    7.4.1.2. Controlling the Termination of an And-Parallel Phase 152
    7.4.1.3. Evaluating Sequential Conjunctions 153
  7.4.2. The Deadlock-Handling Module 154
    7.4.2.1. Implementing the Deadlock Breaking Capabilities of Pandora 155
    7.4.2.2. A Sequential Implementation of Or-Nondeterminism 157
    7.4.2.3. A Parallel Implementation of Or-Nondeterminism 159

8. The Pandora Abstract Machine 161
  8.1. An Overview of the Warren Abstract Machine (WAM) 161
    8.1.1. The WAM Execution Model for Prolog Programs 162
    8.1.2. The WAM Abstract Machine 163
      8.1.2.1. Environments in WAM 165
      8.1.2.2. Choice Points in WAM 167
  8.2. An Overview of Jim Crammond's Abstract Machine (JAM) 168
    8.2.1. The JAM Execution Model for Parlog Programs 168
    8.2.2. The JAM Abstract Machine 170
      8.2.2.1. The Run Queue 171
      8.2.2.2. The Process Stack 172
      8.2.2.3. The Argument Stack 173
      8.2.2.4. The Process Structure 173
      8.2.2.5. Goal Arguments in JAM 174
      8.2.2.6. Environments in JAM 175
      8.2.2.7. The Variable Table 176
      8.2.2.8. Suspending on Variable(s) 177
      8.2.2.9. Suspending on Children 178
      8.2.2.10. Scheduling 179
  8.3. The Pandora Abstract Machine 181
    8.3.1. The PAM Execution Model for Pandora Programs 181
    8.3.2. Comparison with the WAM and the JAM Models 183
    8.3.3. The PAM Abstract Machine 184
      8.3.3.1. The Process Structure 185
8.3.3.2. The Choice-point Stack 186
8.3.3.3. The Trail 187
8.3.3.4. The Deadlock List 188
8.3.3.5. The Trailing Mechanism 189
  8.3.3.5.1. Trailing a Process Structure 190
  8.3.3.5.2. Trailing a Variable 191
  8.3.3.5.3. Trailing a Hanger 192
8.3.3.6. Collecting Suspended Processes in the Deadlock Phase 193
8.3.3.7. The Pandora Deadlock Handler 193
8.3.3.8. Scheduling 195
8.3.3.9. Failure and Backtracking 196
8.3. Summary 197

9. Conclusions 200
  9.1. Summary 200
  9.2. Related Research 202
  9.3. Future Research 203

References 205
Chapter 1

Introduction

1.1 Evolution of Programming Languages

Programming languages are continuously evolving, beginning with assemblers, then Fortran and Algol, and languages like Pascal, C and Ada which are termed *imperative* or *conventional* languages. Such languages are oriented towards the underlying 'von-Neumann' machine on which they are implemented and have inherited many of its characteristics, notably the concept of destructive assignment. Programs written in such languages have an *operational (procedural) semantics*, where the correctness of each program statement can only be determined with reference to its effect on the run-time state on a (real or abstract) machine.

A new style of programming languages, called *declarative languages*, emerged with the introduction of LISP (Winston and Horn, 1984). These languages are distinguished by being based on an abstract formalism. LISP, for example, was the first and is still the most widespread *functional language* based on the mathematical theory of functions known as the lambda calculus (Church, 1941).

A program in a declarative programming language can be read as a formal description of its own specification independent of any machine. Therefore, declarative languages have a *declarative* as well as an operational *semantics*.

*Logic programming* has emerged as another form of declarative language, based on first order predicate logic. This formalism was originally devised for the purpose of studying human reasoning, therefore logic programming languages are declared to be particularly human oriented.
1. Introduction

Logic programming has developed from the work of Kowalski and Colmerauer in the early 1970's. Kowalski (Kowalski, 1974) provided a procedural interpretation for the Horn clause subset of first order predicate logic while Colmerauer and others at Marseille (Roussel, 1975) designed and implemented the first logic programming language, Prolog (PROgramming in LOGic). The main drawback with the Marseille interpreter for Prolog was that it was inefficient and so it attracted only a limited interest in the AI community. It was not until the development of the Prolog-10 compiler (Warren, 1977) that Prolog received wider attention.

The next important step in the implementation of logic programming languages came in 1983 when Warren produced an abstract Prolog instruction set (Warren, 1983) which became known as the Warren Abstract Machine (WAM). The Warren Abstract Machine has had an enormous impact on Prolog implementations in the following years as well as many other logic programming language implementations.

Prolog today is quite a mature language with many commercial systems available and a wide variety of applications developed in the language.

1.2 Non-Determinism in Logic Programming

Most logic programming languages use a goal driven evaluation strategy based on a resolution inference system for Horn clauses, now referred to as SLD. SLD attempts to reduce a conjunction of goals to the program into an empty conjunction by a series of resolution steps in which unification plays the main role (Robinson, 1965; Robinson, 1979; Kowalski, 1974).

As will be explained in Chapter 2, two sources of non-determinism are realized in a resolution step:

1. and-nondeterminism: more than one goal may exist in the current conjunction of goals. Hence, any one of them can be selected for reduction.

2. or-nondeterminism: several clauses may be available for reducing the selected goal and any one of them can be chosen.

Or-nondeterminism causes non-determinism in the solutions computed; successful evaluation paths may produce distinct solutions to the problem. It is not in general
possible to know which path will lead to a solution. In this situation, or-nondeterminism is referred to as *don't-know non-determinism*. If every path is as good as the others and the programmer does not care how or which solution is obtained, then a committed choice to one of these paths is possible. In the latter situation, or-nondeterminism is referred to as *don't-care non-determinism* (also called *committed-choice non-determinism*).

Don't-know non-determinism provides a powerful tool for searching through multiple solutions. Therefore, it makes logic programming languages suitable for applications like knowledge representation, natural language parsing, and expert systems.

And-nondeterminism, on the other hand, affects the efficiency of finding a solution to the problem; different sequencing of a goal reduction result in different problem-solving behaviours. Prolog adopts a fixed *computation rule* by which the leftmost goal is always selected for reduction. This sequential evaluation strategy allows for an efficient stack-based implementation of the language on sequential machines. However, an evaluation may inefficiently visit several unsuccessful search branches before reaching a solution to the problem. Several logic programming languages have been developed as variants of Prolog, in which the sequential computation rule is relaxed. These are explained in Chapter 2.

### 1.3 Parallel Logic Programming

A major influence in favour of declarative languages since the mid 1980's is the increasing attraction to parallel processing and the availability of commercial multiprocessors. Exploiting parallelism in a single application in a conventional language is much harder than in declarative languages. While the former are inherently sequential and include many characteristics of the von-Neumann type of architecture, the latter are based on declarative formalism and lack side-effects (at least in principle).

Additionally, it was announced in 1981 that the main objective of the Japanese Fifth Generation Project is to develop highly parallel architectures for very fast execution of logic programs (Kawanobe, 1984).

There are two major kinds of parallelism that can be identified and exploited in logic programs. These are based on the kinds of non-determinism previously explained:
1. and-parallelism: the concurrent reduction of several goals in a conjunction.

2. or-parallelism: the concurrent exploration of several evaluation paths in the search tree.

In Chapter 2, and-parallelism is classified into several types. The most interesting one is referred to as stream and-parallelism: the concurrent evaluation of goals in a conjunction which share variables, with the values of the shared variables communicated incrementally among the goals. The special significance of stream and-parallelism is that it provides a useful programming paradigm, namely that of parallel communicating processes.

The problem in implementing stream and-parallelism is that concurrently executing goals may generate conflicting bindings to shared variables. One of the goals must then be forced to undo its bindings and generate alternative ones. Other goals that are sharing these variables must also backtrack to their state of computation before the conflicting bindings were made. This behaviour does not seem to coexist well with the idea of stream and-parallelism, where shared variable bindings act as "messages" between processes. This is why a new family of logic programming languages, called committed-choice logic programming languages, has been developed. These languages are based on committed-choice non-determinism instead of don't-know non-determinism. The use of committed-choice non-determinism, together with a suspension mechanism that delays binding a goal variable until after commitment, ensures that all variable bindings are committed bindings: that is, they will not be undone. Members of this family are the Relational Language (Clark and Gregory, 1981), Parlog (Clark and Gregory, 1986), Concurrent Prolog (CP) (Shapiro, 1983), and Guarded Horn Clauses (GHC) (Ueda, 1985).

Committed-choice non-determinism makes the committed-choice languages less suitable than Prolog for applications which require searching through multiple solutions, but it has had the substantial benefit of allowing very simple and efficient implementations of stream and-parallelism (Crammond, 1988; Foster, 1988). Another major contribution of these languages has been to make logic programming applicable to a wide range of new applications: those which can naturally be expressed as systems of concurrent, communicating processes. Examples of these applications are systems programming (Foster, 1988), discrete event simulation (Broda and Gregory, 1984), and specification, verification and simulation of
communication protocols (Gregory et al, 1985).

Earlier attempts have been made to combine don't-know non-determinism and stream and-parallelism into one language or system (e.g. CP[↓, l, &;] (Saraswat, 1987a] and P-Prolog (Yang, 1987)). This is attractive for two reasons:

(a) more parallelism can be exploited in the kinds of application currently written in Prolog;

(b) combining don't-know non-determinism with a concurrent programming style may open up yet more interesting application areas to logic programming.

However, each of these attempts has one or more of the following drawbacks:

(a) only a restricted form of search is supported.

(b) any number of goals can be subject to an or-parallel execution at the same time, and copies of the entire computation is then required for each branch of the search tree. This significantly affects the efficiency of the implementation.

(c) don't-know non-determinism and stream and-parallelism are provided by interacting, but separate, systems. Hence, several features are duplicated and the result is a non-unified system.

In 1988, David Warren proposed the **basic Andorra model** (Yang, 1988; Yang, 1989): a new execution model which transparently extracts stream and-parallelism from Prolog programs. In this model, execution alternates between two phases: the and-parallel phase and the non-deterministic phase. During the **and-parallel phase**, all deterministic goals are evaluated concurrently. A goal is **deterministic** if it has at most one clause that is unifiable with it. This is checked by simple run-time tests, intended to be deduced by a compile-time analysis. When no deterministic goals remain, a **non-deterministic phase** is started in which a choice point is created for the leftmost goal. For each search branch, a new and-parallel phase is begun, running all deterministic goals until they are exhausted.

The basic Andorra model executes search programs in a coroutining manner which can significantly reduce the search space in many cases, resulting in a more intelligent problem-solving behaviour than either a sequential evaluation, as in Prolog, or a fully eager parallel evaluation as in CP[↓, l, &;].

An additional bonus of this novel form of "lazy non-determinism" is to make an implementation of an integrated system combining stream and-parallelism and
1. Introduction

don't-know non-determinism feasible.

However, the basic Andorra model does not fully subsume Prolog. An Andorra evaluation of a query might not terminate while the Prolog evaluation would fail. Moreover, a mechanism for sequencing the evaluation of a series of actions should be added to the basic Andorra model in order to run Prolog programs with side effects.

When compared with committed-choice languages, the basic Andorra model can run a deterministic Prolog program with behaviour similar to that of committed-choice languages. However, a non-deterministic goal cannot be executed in the and-parallel phase by committing to one of its candidate clauses, as in committed-choice languages. Also, it cannot be forced to suspend, on an explicit binding condition, if it is the leftmost goal in the non-deterministic phase. These considerations suggest that the Prolog syntax is insufficient if it is desired to support applications that are currently expressible in committed-choice languages.

1.4 Context of Research

This thesis proposes a new parallel logic programming language: Pandora (Parlog and Andorra), combining stream and-parallelism and "lazy don't-know non-determinism".

As a language, Pandora extends Parlog with a deadlock handling mechanism and a don't-know non-deterministic fork primitive. Thus, it provides stream and-parallelism, committed-choice non-determinism, and don't-know non-determinism. The Pandora computational model is related to Warren's Andorra model. In Pandora, a non-deterministic choice is made after all deterministic computation, as well as committed-choice non-deterministic computation take place. Therefore, the computation alternates between two phases: the and-parallel phase when goals are evaluated concurrently except for the don't-know non-deterministic ones which suspend, and the deadlock phase in which an "arbitrary" non-deterministic goal can be reduced. A new and-parallel phase is then begun for each search branch of the non-deterministic choice.

Pandora inherits from Parlog its synchronization mechanism and committed-choice non-determinism, while it inherits from the basic Andorra model the advantages
obtained by the lazy don't-know non-deterministic behaviour, including the reduction of search space and the feasibility of implementation. Moreover, Pandora provides a meta-level deadlock handler relation by which the programmer can explicitly program the behaviour on deadlock in a manner that is suitable to the particular application. One way in which the deadlock handler relation can be used is in combination with the non-deterministic fork, to program a heuristic search. This is a way to reduce the search space by intelligently selecting a non-deterministic goal to execute, when several are possible. More generally, the deadlock handler relation can be utilized to manipulate suspended goals at the time of deadlock in a flexible manner. Redundant goals can be removed, new goals can be added, and a group of goals can be replaced by a simpler group.

In addition to the standard applications for logic programming, this research opens up interesting application areas that cannot conveniently be expressed in existing logic programming languages; some of these applications are explored in this thesis.

1.5 Contributions

The main contributions of the thesis can be summarized by the following points:

1. the development of a practical parallel logic programming language, called Pandora, which combines stream and-parallelism, committed choice non-determinism, and lazy don't-know non-determinism.

2. the definition of the Pandora computational model which provides an intelligent problem-solving behaviour.

3. the development of several programming algorithms and techniques using the novel problem-solving behaviour of the Pandora language computational model.

4. the exploration of new interesting application areas, such as resource allocation problems, constraint-based reasoning, and distributed discrete event simulation.

5. the definition of a meta-level deadlock handler relation by which the user can program the behaviour on deadlock, and the utilization of such powerful tool in several applications including the development of constraint logic programming systems.

6. the development of a prototype implementation of the language on which potential applications are explored and design decisions are tested.

7. the design of a multi-processor abstract machine for Pandora which can be
implemented with reasonable efficiency. The implementation benefits from the techniques that were developed and used for the implementation of existing logic programming languages, and also solves new issues arising from combining stream and-parallelism and don't-know non-determinism.

1.6 Preview of Contents

This thesis is logically divided into 4 parts. Part 1 (Chapters 2 to 5) covers the motivation, design and applications of the basic Pandora language.

Chapter 2 begins with an overview of established concepts of logic programming. In particular, it introduces don't-know non-determinism, committed-choice nondeterminism and stream and-parallelism, with their potential uses in logic programming. This is followed by a survey of previous attempts for combining don't-know non-determinism with stream and-parallelism. The most recent attempt considered is the basic Andorra model.

Chapter 3 introduces the basic features of Pandora: a language and a computational model. Several compile-time analyses that are required to ensure a correct operational semantics of Pandora programs are also explained. Then, Pandora is compared with both the Parlog language and the Andorra model. The chapter ends with an example of a Pandora program and explains its behaviour.

Chapters 4 and 5 further illustrate the use of Pandora by means of substantial programming examples. Several programming techniques are developed to: (1) adopt constraint-based reasoning in Pandora programs and use it in solving resource allocation problems, (2) break the deadlock of a distributed discrete event simulation. Two Pandora programs are given to illustrate these techniques. Namely, an automated generation of naval flying programmes, and a simulation of a cyclic computer system.

Part 2 (Chapter 6) illustrates additional features of the language.

Chapter 6 introduces the Pandora deadlock handler relation by which suspended goals in the deadlock phase can be examined and manipulated in order to break deadlock in a manner that is related to the particular application. Various programming examples are given to illustrate the use of this relation.
Part 3 (Chapters 7 and 8) covers the language implementation.

Chapter 7 presents a prototype system for Pandora that runs on top of the Parlog SPM system. The system comprises a compiler from Pandora programs to Parlog procedures and an interpreter for running queries to the compiled programs.

Chapter 8 introduces PAM: the Pandora Abstract Machine, and its process-oriented execution model.

The final part of the thesis consists of Chapter 9, which summarizes the past and present research on Pandora, and future plans, together with a survey of some of the related research.
Chapter 2

Logic Programming

This chapter is primarily an overview of established concepts of logic programming. However, we shall assume that the reader has some familiarity with these concepts and hence we shall not provide a complete introduction to the subject; there are many such introductions available including (Kowalski, 1979; Hogger, 1984; Lloyd, 1987) to which the interested reader is referred.

The chapter begins by summarizing the syntax and semantics of (pure) Horn clause programs, and defines the various types of non-determinism in the evaluation of logic programs. This is followed by a discussion of how these types of non-determinism are controlled in various logic programming languages starting with Prolog, which provides don't-know non-determinism but adopts a sequential evaluation strategy of goals in a conjunction, and ending with committed-choice logic programming languages, which are based on committed-choice non-determinism in order to provide an efficient stream and-parallel programming paradigm. The syntax and semantics of guarded Horn clauses in committed-choice programs are also defined.

While don't-know non-determinism provides a powerful tool for programming search problems, stream and-parallelism provides a powerful and efficient tool for programming a system of parallel communicating processes. The final section surveys some of the schemes that have been proposed to combine stream and-parallelism with don't-know non-determinism, and lists some of their drawbacks. This establishes a basis for our thesis which proposes Pandora: a new parallel logic programming language combining don't-know non-determinism and stream and-parallelism in an integrated manner.
2.1 Syntax of Logic Programs

A logic program is a sequence of universally quantified Horn clauses, each of the form

\[ H \text{ if } B_1 \text{ and } B_2 \text{ and } \ldots \text{ and } B_n \]

where \( H, B_1, \ldots, B_n \) are atomic formulae. \( H \) is termed the head of the clause and "\( B_1 \text{ and } \ldots \text{ and } B_n \)" is its body.

Using the Edinburgh syntax (Clocksin and Mellish, 1981), an atomic formula is of the form \( r(T_1, \ldots, T_k) \) \( k \geq 0 \). It represents a relation (predicate) \( r/k \) where \( r \) is the relation name and \( k \) is its arity. \( T_1, \ldots, T_k \), the arguments of the relation, are terms.

An atomic formula which does not appear as the clause head is also called an atomic goal (or a goal for simplicity). A conjunction of atomic goals "\( G_1 \text{ and } G_2 \text{ and } \ldots \text{ and } G_n \)" \( n \geq 0 \) is called a goal statement.

A term is either a logical variable, a constant, or a structured term, \( f(T_1, \ldots, T_m) \), which represents a function whose name is \( f \), its arity is \( m \), and its arguments \( T_1, \ldots, T_m \) are terms. A list is a structured term that is oftenly used in logic programming. For convenience, a special syntax is used for lists. A list with head \( X \) and tail \( Xs \) is written as \([X|Xs]\) while the empty list is denoted by \('[]'\).

A logical variable is an unquoted alphanumeric identifier beginning with either an upper-case letter or an underscore. For example, \( X, _X, Y_{12}, \_Y_{12} \) are all variables. A logical variable is initially an unbound variable that is instantiated when bound to a non-variable term \( T \). Once instantiated, a logical variable cannot be bound to a different term, i.e. it is a single assignment variable. A term which contains no unbound variables is known as a ground term.

A constant is either a number or a structured term with no arguments and is syntactically differentiated from a variable name by beginning with a lower-case letter, a number or any character string enclosed in quotes. Examples of constants are \( 213, \_x, 'no more solutions' \).
2. Logic Programming

Program 2.1 is an example of a logic program which consists of four Horn clauses defining the perm and delete relations.

```
perm([], []).
perm([H|T], [X|Xs]) if
    delete(X, [H|T], L1) and
    perm(L1, Xs).
% c11.

delete(X, [X|Xs], Xs).
% c12.

delete(X, [Y|Z], [Y|Z1]) if
    delete(X, Z, Z1).
% c13.

delete(X, [X|Xs], Xs).
% c14.
```

Program 2.1. Permutation of a list.

2.2 Semantics of Logic Programs

Horn clauses in a logic program have a clear declarative reading. A clause is a universally quantified implication. If \( X_1, \ldots, X_k \) are the variables in the clause, then the clause is read:

"for all \( X_1, \ldots, X_k \): \( H \) is true if \( B_1 \) and \( B_2 \) and ... and \( B_n \) are true ".

A program is then considered as a set of statements about the relations in the heads of its clauses. For instance, the relations in Program 2.1 have the following logical reading:

" \( \text{perm}(X, Y) \): \( Y \) is a permutation of the list \( X \) (clauses 1 and 2).

\( \text{delete}(X, Y, Z) \): \( Z \) is the list \( Y \) without one occurrence of the element \( X \).

\( X \) can be either the first element in \( Y \) (clause 3), or any other element in \( Y \) (clause 4) ".

Kowalski (Kowalski, 1979) introduced the procedural interpretation of logic programs in which a clause is read:

" to prove \( H \), prove \( B_1 \) and \( B_2 \) and ... and \( B_n \)".

The head of a clause is a procedure entry point, a goal is a procedure call (a procedure invocation), and a procedure is a set of clauses with the same head predicate. Instances of a relation can be non-deterministically computed using the different entry points of the relation's procedure.
2. Logic Programming

Most logic programming languages use a goal driven evaluation strategy based on a resolution inference system (Robinson, 1965; Robinson, 1979) for Horn clauses. The evaluation strategy is referred to as the SLD (Selective Linear resolution for Definite clauses), in which unification plays a main role. In this section we give a brief introduction to these concepts. For a complete treatment the reader should consult (Robinson, 1965; Kowalski, 1974; Robinson, 1979).

2.2.1 Unification

Unification is the process of finding a substitution, called a unifier, that makes two or more atomic formulae syntactically identical.

A substitution \( s = (X_1/T_1, ..., X_k/T_k) \) is a set of variable/term pairs in which \( X_1, ..., X_k \) are distinct variables and no variable \( X_i \) appears in any term \( T_j \) (1 ≤ i ≤ k, 1 ≤ j ≤ k). If \( T \) is a term and \( s \) is a substitution, then \([T]s\) is a substitution instance of \( T \). \([T]s\) is the term obtained by replacing \( X_1, ..., X_k \) in \( T \) by \( T_1, ..., T_k \), respectively.

Two terms are syntactically identical if they are both the same variable, the same constant, or are both structured terms with the same function name and syntactically identical arguments.

For example, the substitution:

\[
sl = (A/X, Y/2, Z/[1], U/[2|Z1])
\]

is a unifier for the pair of atomic formulae:

\[
\text{delete}(A, [2,1], U), \text{delete}(X, [Y|Z], [Y|Z1])
\]

When applied to each atomic formula, it produces:

\[
\text{delete}(X, [2,1], [2|Z1])
\]

A unifier \( s \) is said to be a most general unifier if it is not an instance of any other unifier. More formally, \( s \) is a most general unifier if every other unifying substitution \( s' \) is such that \( s' = s \cdot s'' \) for some substitution \( s'' \). \( s \cdot s'' \) is the composition of the two substitutions \( s \) and \( s'' \). Let \( s \) and \( s'' \) be defined as:

\[
s = (X_1/T_1, ..., X_n/T_n), \quad s'' = (Y_1/T'_1, ..., Y_k/T'_k)
\]

Then, \( s \cdot s'' \) is the substitution:
2. Logic Programming

\[ s'1 \lor s'2 \]  
where \( s'1 = (X_1/[T_1]s'', \ldots, X_n/[T_n]s'') \)
and \( s'2 \) is \( s'' \) with any bindings for the variables \( X_1, \ldots, X_n \) deleted.

For example, suppose \( s_2 \) and \( s_3 \) are the substitutions:

\[ s_2 = \{ A/X, U/[2|Z_1] \} \]
\[ s_3 = \{ X/1, U/[] \} \]

Then, \( s_2 \cdot s_3 \) is the substitution \( \{ A/1, U/[2|Z_1] \} \), and \( \text{delete}(A, [2,1], U) \cdot s_2 \cdot s_3 \) is the substitution instance "\( \text{delete}(1, [2,1], [2|Z_1]) \)".

\( s_1 \), above, is a most general unifier for the \text{delete} formulae; any other unifier must have the same bindings for \( Y \) and \( Z \) but can bind \( U \) to any term of the form \( [2|T_1] \) if it includes the binding \( Z_1/T_1 \), and can bind \( A \) (or \( X \)) to any term \( T_2 \) if it includes the binding \( X/T_2 \) (or \( A/T_2 \)).

Unification may result in either \textit{success} or \textit{failure}. Unification of two terms succeeds when a most general unifier for the two terms exists. If there is no substitution that makes the two terms syntactically identical, unification fails. For example, the unification of the terms \( \text{delete}(A, [2,1], U) \) and \( \text{delete}(A, [1,2], Z) \) fails since the lists \([2,1]\) and \([1,2]\) are not syntactically identical.

2.2.2 SLD Resolution

\textit{SLD} attempts to reduce an initial goal statement (a \textit{query}) to an empty goal statement by a series of resolution steps, forming a \textit{computation}. As a by-product, a substitution for the variables in the query may be produced.

At each resolution step, the state of computation can be represented by \( <G;\theta> \) where \( G \) is the current goal statement (initially the query) and \( \theta \) is the current substitution (initially empty). A resolution step consists of the following actions:

1. select a goal \( G_j \) from \( G \) (the current goal statement).
2. select a clause \( C_j \) whose head \( H_j \) has the same relation name and arity as \( G_i \).
3. unify \( H_j \) and \( [G_i]\theta \) with a most general unifier \( \alpha \), after renaming the variables in \( C_j \) to be different from the variables in \( G \).
4. produce the new state of computation \( <G';\theta> \) where \( G' \) is \( G \) after replacing...
2. Logic Programming

$G_1$ with the body of the clause $C_j$, and $\theta'$ is the composition of $\theta$ (the current substitution) and $\alpha (\theta \cdot \alpha)$.

A computation is successful when an empty goal statement is finally derived. If a resolution step fails (in any of the actions 2-4), the resulting goal statement is unsolvable and the computation is unsuccessful. The substitution in the terminal state of a successful computation, restricted to the variables in the initial goal statement, is called an answer substitution (also called a solution).

2.3 Non-Determinism

Two sources of non-determinism are apparent in the above definition of a resolution step. In 1, any goal can be selected from the current goal statement, and any clause for that goal can be selected in 2. These are called and-nondeterminism and or-nondeterminism, respectively.

2.3.1 Or-Nondeterminism

Or-nondeterminism causes non-determinism in the solutions computed. If there are several clauses whose heads have the same relation name and arity as a goal in the current goal statement, several evaluation paths can be followed, and each path may either fail (forming an unsuccessful computation) or succeed with a solution that may be different from the solutions produced by other successful computations. Some of the evaluation paths may be infinite. Figure 2.1 illustrates a search tree of all the computations determined by Program 2.1 with the query delete(X, [2,1],Z).

The substitution produced by each resolution step is directly applied to the resulting goal statement in order to achieve an easy reading of the diagram. The tree contains two successful computations and an unsuccessful one. The empty goal statement is represented by "☐".

Or-nondeterminism can be further classified (Kowalski, 1979) into don't-know non-determinism and don't-care non-determinism. Several clauses in a logic program may have heads that are unifiable with a particular goal. It is not in general possible to know which clause must be evaluated to compute a solution. This underspecification is referred to as don't-know non-determinism. If each clause is as good as the others in reducing a goal and the programmer "does not care" how or which solution is obtained, then a "committed-choice" of one clause can be made,
ignoring the other unifiable clauses. Such behaviour is referred to as don’t-care non-determinism (also called committed-choice non-determinism).

Consider for example the query \texttt{delete(X, [2,1], Z)} to Program 2.1 which has two possible solutions: \{X/2, Z/[1]\} and \{X/1, Z/[2]\} as shown in Figure 2.1. If the programmer does not care which solution is produced, then any of clauses 3 or 4 can be used to reduce the initial goal. If the query is "\texttt{delete(X, [2,1], Z)} and \texttt{X = 1}" where '=' is the equality relation, then there is only one possible solution, that is \{X/1, Z/[2]\}. While both clauses 3 and 4 are unifiable with the goal \texttt{delete(X, [2,1], Z)}, it is not directly clear which one of them will eventually lead to a successful computation. Therefore, committing to one clause may ignorantly commit to clause 3 (Br1 in Figure 2.2) which will lead to failure.

Don't-know non-determinism provides a powerful tool for searching through several branches of the search tree until a solution is found or, even better, until all solutions are collected. Hence, it makes logic programming languages suitable for applications like data base querying, natural language parsing, and expert systems since it liberates the user from explicitly programming the tree search. However, logic programming systems apply different \textit{search rules} to determine the order in which clauses are chosen to reduce a goal.

Sequential Prolog systems apply the clauses in the procedure of a goal in their textual order, and backtrack in case of failure. The search rule applies the top-most clause for the left-most goal. If the resolution step fails, \textit{backtracking} causes the most recent resolution step which has alternative clauses to be repeated using the next clause in the textual order. Only if no alternative clause exists in any previous step does resolution fail. As a result, the system traverses the search tree in a left-
most depth-first manner.

For example, \( \{ \text{Br1}\} \) of the search tree in Figure 2.2 will be the first evaluation path to be visited by a sequential Prolog implementation. Then, the system backtracks and visits \( \{ \text{Br2, Br2.1}\} \). The evaluation terminates successfully and produces the solution \( \{ X/1, Z/[2]\} \).

Several systems (Warren, 1987) have been developed which implement don't-know nondeterminism by \textit{or-parallelism}, that is, visiting several evaluation paths simultaneously. The principal problem associated with an or-parallel implementation is maintaining alternative values for variables generated by different clause evaluations. In the previous example, the evaluation path \( \{ \text{Br1}\} \) binds \( X \) to 2 and \( Z \) to \([1]\) while the evaluation path \( \{ \text{Br2, Br2.1}\} \) produces the bindings \( \{ X/1, Z/[2]\} \). A number of schemes for maintaining this information have been proposed (Crammond, 1985; Warren, 1987).

\section*{2.3.2 And-Nondeterminism}

Logic programming systems apply different \textit{computation rules} to determine the order in which goals in a conjunction are reduced, hence controlling and-nondeterminism. As previously explained in (Kowalski, 1974; Clark, 1979) and is shown below, the order in which goals are reduced affects the efficiency of searching for a successful evaluation to a query.
2.3.2.1 "Generate and Test" Evaluation

Prolog has a fixed computation rule which always selects the left-most goal for reduction. As a result, Prolog systems adopt a generate and test evaluation strategy in which a value is assigned to each variable in the problem before testing whether these values satisfy the problem's conditions. If the assignment is infeasible, i.e. it does not satisfy one or more tests, it is rejected and the computation retracts the value given to the most recently assigned variable (i.e. the most recent choice) and assigns an alternative value for it. It then tests the new assignment. This process is repeated until either a solution is reached or all the possible assignments proved to be infeasible.

For example, Program 2.2 defines the problem of sorting a list of numbers in ascending order. The relations in the program have the following logical reading:

"sort(X, Z): Z is the sorted version of list X.
ord(Z): Z is a list of partially ordered elements by the relation '<'.
perm(X, Z) was previously defined in Program 2.1".

Program 2.2. Naive sorting of a list of numbers in ascending order.

A query sort([2,1,3], Z) is immediately reduced to perm([2,1,3], Z) and ord(Z). Using the left-most computation rule as in Prolog, Z will be completely constructed by the perm goal before being tested by the ord goal to check whether it is ordered. One unsuccessful evaluation path for the query is presented in Figure 2.3.
sort([2, 1, 3], Z)
c11 | perm([2, 1, 3], Z) and ord(Z)
c12 | Z = [X/Z]
delete(X, [2, 1, 3], Z1) and perm(Z1, Z2) and ord([X/Z])
c13 | X = 2
     | Z1 = [1, 3]
perm([1, 3], Z2) and ord([2/Z2])
c12 | Z = [2, X1/Z4]
delete(X1, [1, 3], Z3) and perm(Z3, Z4) and ord([2, X1/Z4])
c13 | X1 = 1
     | Z3 = [3]
perm([3], Z4) and ord([2, 1/Z4])
c12 | Z = [2, X2/Z6]
delete(X2, [1, 3], Z5) and perm(Z5, Z6) and ord([2, 1, X2/Z6])
c13 | X2 = 3
     | Z5 = []
perm([1], Z6) and ord([2, 1, 3/Z6])
c11 | Z6 = []
ord([2, 1, 3])
c14 | 2 ≤ 1 and ord([1, 3])

Fig 2.3. An unsuccessful leftmost computation for the query sort([2, 1, 3], Z).

2.3.2.2 "Incremental Generate and Test" Evaluation

Several logic programming languages have been developed to apply a coroutining computation rule, whereby the sequential "generate and test" evaluation is relaxed. Examples of these systems are IC-Prolog (Clark and McCabe, 1979), Prolog-II (Colmerauer, 1982) and MU-Prolog (Naish, 1985).

By default, goals are reduced from left to right. However, specific notations can be used in the program to nominate a goal as a "consumer" of a particular value; the evaluation of the consumer is delayed until the value it is to consume is computed. If the goals in the query as well as in the body of a clause are ordered so that the
consumer of a value is invoked before its producer, then the evaluation proceeds in a coroutining manner. Consumer goals suspend until their arguments are sufficiently instantiated while (don't-know non-deterministic) producers can be evaluated, assigning values to variables. As soon as a variable is instantiated, its consumer is eagerly evaluated, testing whether the value that is assigned to it satisfies the problem's constraints. If a test does not hold, then the computation retracts the value given to the most recently assigned variable and assigns an alternative value for it.

The incremental generation of values and the eager evaluation of its consumers increase the possibility of detecting failure immediately after producing the values responsible for it. This results in a more intelligent problem-solving behaviour than a sequential execution as in Prolog.

Consider for example the previous program. If ord/1 is nominated as a consumer of its argument and the body of clause 1 is re-arranged so that the ord goal is invoked before perm, then a coroutining evaluation will initiate the ord test goal as soon as the first two elements of the permutation have been determined. This has the advantage that failure can be detected as soon as possible before generating the whole permutation while earlier elements are not ordered. Figure 2.4 below illustrates the effectiveness of eagerly executing the ordering test to reject in one step all permutations which have the first two elements 2 and 1 respectively.

**Fig 2.4.** An unsuccessful coroutined computation for the query `sort([2,1,3], Z).`
2.3.2.3 Restricted And-Parallelism

Coroutining allows the reduction of goals to be interleaved, thus relaxing the sequential "generate and test" evaluation. This can be further relaxed to support and-parallelism. That is, the parallel execution of two or more goals in a goal statement or, in other words, the parallel evaluation of several resolution steps. The simplest form of and-parallelism is known as independent or restricted and-parallelism in which independent goals, i.e. goals that do not share variables, are evaluated in parallel. Algorithms to determine at compile time and/or at runtime that goals in a conjunction are independent have been described in (Conery and Kibler, 1981; DeGroot, 1984; Hermenegildo, 1986; Conery87).

For instance, suppose the current state of computation is:

\[ <G_1 \text{ and } G_2; \theta> \]

where \( G_1 \) and \( G_2 \) have no shared variables. Hence, \( \theta \) can be split into:

\[ \theta = \theta_1 \lor \theta_2 \]

where \( \theta_1 \) is the substitution of the variables in \( G_1 \), and \( \theta_2 \) is the substitution of the variables in \( G_2 \). A restricted and-parallel evaluation may then evaluate \( <G_1; \theta_1> \) and \( <G_2; \theta_2> \) concurrently. If both resolutions succeed and result in the substitutions \( \alpha_1 \) and \( \alpha_2 \) respectively, then the composition \( \alpha_1 \lor \alpha_2 \) is equivalent to \( \alpha_1 \cdot \alpha_2 \) since \( \alpha_1 \) and \( \alpha_2 \) have no shared variables.

Detecting independent goals enables some intelligent backtracking to be performed should a goal fail. For example, suppose goals \( G_1, G_2 \) and \( G_3 \) are executing in parallel and \( G_2 \) fails. As they are all independent, \( G_1 \) and \( G_3 \) cannot possibly find an alternative solution to satisfy \( G_2 \). Therefore, the whole conjunction can immediately fail without considering alternative clauses for reducing \( G_1 \) or \( G_3 \), hence avoiding visiting several failing branches in the search tree.

2.3.2.4 Stream And-Parallelism

An interesting form of and-parallelism in logic programming is stream and-parallelism, whereby goals in a conjunction are evaluated concurrently and may
communicate incrementally by bindings to shared variables. As will be shown later in this chapter, the special significance of stream and-parallelism is that it provides a useful programming paradigm: that of parallel communicating processes.

The problem in implementing stream and-parallelism is that concurrently executing goals may generate conflicting bindings to shared variables. For example, suppose that the current state of computation is:

\[<G_1 \text{ and } G_2; \theta>\]

where \(G_1\) and \(G_2\) are only sharing the unbound variable \(X\). \(\theta\) can be split into:

\[\theta = \theta_1 \lor \theta_2\]

where \(\theta_1\) is the substitution of the variables in \(G_1\), and \(\theta_2\) is the substitution of the variables in \(G_2\). \(X\) is not bound in \(\theta_1\) nor \(\theta_2\) since it is an unbound variable. If the resolutions of \(<G_1; \theta_1>\) and \(<G_2; \theta_2>\) result in the substitutions \(\alpha_1\) and \(\alpha_2\) respectively, whereby \(\alpha_1\) and \(\alpha_2\) have conflicting bindings for the shared variable \(X\), then the composition \(\alpha_1 \cdot \alpha_2\) will not exist. Both evaluations should then backtrack to their intermediate states of computation before the conflicting bindings were made and one of them should take an alternative evaluation path, producing a different binding to \(X\). Realizing which intermediate states of computation to backtrack to, and synchronizing the backtracking process of each of the concurrent evaluations can be complex, specially in a parallel implementation whereby the concurrent evaluations take place on distributed machines. Therefore, an efficient parallel implementation of stream and-parallelism requires that each goal computes no more than one solution.

### 2.4 Committed-Choice Logic Programming Languages

Several concurrent logic programming languages have been developed to provide stream and-parallelism. The most widely used of these languages are called committed-choice languages because they are based on committed-choice nondeterminism. That is, a goal is reduced by committing to only one clause in its procedure and hence computes no more than one solution. Members of this family are the Relational Language (Clark and Gregory, 1981), Concurrent Prolog (CP) (Shapiro, 1983), Guarded Horn Clauses (GHC) (Ueda, 1985), and Parlog (Clark...
and Gregory, 1986), which are described and compared in (Takeuchi and Furukawa, 1986; Shapiro, 1989).

A committed-choice program is a logic program that is augmented with specific control annotations which affect the program's procedural interpretation. A clause in a committed-choice program is a universally quantified guarded Horn clause. A guarded Horn clause is a Horn clause optionally augmented with a commit operator which syntactically separates the right hand side of the clause into two conjunctions of goals, termed the guard and body of the clause:

\[ \text{H if} \langle \text{guard} \rangle \text{ commit} \langle \text{body} \rangle \]

For instance, Program 2.3 is a Parlog program defining the relations qsort/3 and partition/4. qsort/3 is defined by two guarded Horn clauses while partition/4 is defined by three guarded Horn clauses. The implication operator is denoted by '->' and the commit operator by ':'. Each procedure is preceded by a mode declaration (mode qsort(? , ?, ?) and mode partition(? , ?, ? , ?)); it is a specific Parlog notation to state whether each argument is input (?) or output ('^'). In this example, the first and third arguments of qsort are input while the second argument is output. Similarly, the first two arguments of partition are input while the third and fourth arguments are output. The role of these notations is explained in the following section.

\begin{verbatim}
mode qsort(? , ?, ?).
qsort([], R , R). % cl1.
qsort([X|Xs], RO, R) <- partition(Xs, X , L1, L2),
qsort(L1, RO, [X|R1]),
qsort(L2, R1, R). % cl2.

mode partition(? , ? , ? , ?).
partition([], _, [I, []]). % cl3.
partition([XI|Xs], A, [XI|Lower], Upper) <- X <= A :
partition(Xs, A, Lower, Upper). % cl4.
partition([XI|Xs], A, Lower, [XI|Upper]) <- X > A :
partition(Xs, A, Lower, Upper). % cl5.
\end{verbatim}

\textbf{Program 2.3.} The Parlog quick sort program.
2.5 Semantics of Committed-Choice Programs

Ignoring the control language, a committed-choice program is a set of logical axioms which have a declarative reading, similar to Horn clause programs. If \( X_1, \ldots, X_k \) are the variables in the guarded Horn clause, then the clause is read:

"for all \( X_1, \ldots, X_k: \text{H is true if } <\text{guard}> \text{ and } <\text{body}> \text{ are true}".

A committed-choice relation such as \( \text{partition}(X, A, \text{Lower}, \text{Upper}) \) in Program 2.3 can thus be logically read:

"If \( X \) is a list of numbers and \( A \) is a number, then \( \text{Lower} \) is the list of all the elements in \( X \) that are less than or equal to \( A \) and \( \text{Upper} \) is the list of all the elements in \( X \) that are greater than \( A \)".

Procedurally, the evaluation of a committed-choice program is based on a specialized form of resolution, called input resolution. In each resolution step, several goals can be reduced in parallel. A new state of computation \( \langle G; \theta' \rangle \) is then produced from the current one \( \langle G; \theta \rangle \) by replacing each reduced goal in \( G \) with the conjunction of goals to which it is reduced, and \( \theta' \) is the composition of \( \theta \) with all the substitutions resulting from the reduction of the goals.

An atomic goal is reduced by applying a goal resolution step, which consists of three phases:

1. the clause selection phase, in which the evaluation searches for a clause or clauses that can reduce the goal.
2. the commit phase, in which the evaluation commits to only one of the clauses resulting from the previous phase.
3. the reduction phase, in which the goal is reduced with the selected clause.

Searching for a clause with which to solve a goal must not bind any variable in the goal; such bindings must only be made when committing to a clause. Committed-choice systems apply various mechanisms to impose such a restriction. These mechanisms differ in their programming capabilities as well as the complexity of their implementation. The operational semantics of Parlog, for example, delays the unification of output arguments in the head of a clause with the corresponding arguments in the goal until commitment. Moreover, input matching and test unification are applied on input arguments. These features provide a reasonably rich programming capabilities and efficient implementations that are comparable in speed to commercial Prolog systems (Crammond, 1988; Foster and Taylor, 1988).
2. Logic Programming

Input matching (also called matching, or one-way unification) is a restricted form of the "full" unification that was described in Section 2.2.1. While input matching is denoted in Parlog by the primitive "<=", full unification is denoted by the primitive "=". "H <= G" is the process of finding the most general matching substitution s by which G becomes an instance of H. In other words, G = [H]s. As a result, s will not include bindings to any variable in G.

While the unification of two terms results in either success or failure, input matching results in one of the following:

1. success: if there is s such that G = [H]s.
2. failure: if no s exists such that [G]s = [H]s.
3. suspension: if for every s, such that [G]s = [H]s, s includes bindings for variables in G.

For example, the goal qsort([], R1, R2) succeeds in matching with the head of clause 1 in Program 2.3 but fails to match with the head of clause 2 ([] * [X|Xs]). An attempt to match the goal qsort(X1, R1, R2) with the head of clause 1 or clause 2 will suspend since it can only proceed by binding X1 to [] in clause 1 or to [X|Xs] in clause 2.

Another restricted form of unification that is employed by Parlog is test unification. This is denoted by the Parlog primitive "==". Test unification is the process of unifying two terms without binding variable(s) in any of the terms.

The test unification G == H may result in one of the followings:

1. success: if G and H are syntactically identical.
2. failure: if G and H cannot be unified, i.e. no s exists such that [G]s = [H]s.
3. suspension: if the test can only proceed by binding variable(s) in H and/or G. That is, for every substitution s, such that [H]s = [G]s, s includes bindings for variables in H or G.

For example, R1 == R2 suspends until the variables R1 and R2 are bound to the same unbound variable or to any identical terms, then succeeds. If R1 and R2 are bound to non-unifiable terms, the test unification fails.

In addition to the various types of unification, Parlog imposes a restriction on the goals that can appear in the guard of a clause. These goals must not bind any
variables in the input arguments of the goal; output arguments can be bound by the
guard goals since output unification (i.e. unification of output arguments) is delayed
until commitment.

In order to further clarify the procedural semantics of a Parlog program, we first
introduce the standard form of Parlog clauses in which the various types of
unification are made explicit as calls to primitives. This form was previously
described in (Gregory, 1987).

2.5.1 Standard Form of Parlog Clauses

A clause in a Parlog program is of the form:

\[ r(T_1, ..., T_k) \leftarrow G : B. \]

where \( G \) and \( B \) are conjunctions of goals. In order to transform the clause to a
standard form, the head arguments are replaced by distinct variables. Then a call to
the primitive '\( = \)\)' is added to the guard for each input argument, and a call to the
full unification Parlog primitive '\( = \)\)' is added to the body for each output argument.
If a variable \( Q \) occurs more than once in the input argument terms, new variables
\( Q_1, ..., Q_j \) are introduced in place of the repeated occurrences and calls to the '\( = = \)\'
primitive \( Q = = Q_1, ..., Q = = Q_j \) are added to the guard to test that every occurrence of
the variable is given the same value.

For example, Program 2.4 illustrates the standard form of the clauses in Program
2.3. Several optimizations can be applied to remove unnecessary calls in the
illustrated standard form of clauses. These are irrelevant to our discussion but are
explained in (Gregory, 1987).

\[
\begin{align*}
\text{qsort}(P_1, P_2, P_3) & \leftarrow \\
\text{[\text{[]} \leftarrow P_1, R \leftarrow P_3; \\
R = P_2. \quad \% \text{ cl1}. \\
\text{qsort}(P_1, P_2, P_3) & \leftarrow \\
\text{[X|Xs] \leftarrow P_1, R \leftarrow P_3; \\
R_0 = P_2, \\
partition(Xs, X, L1, L2), \\
\text{qsort}(L1, R_0, [X|R_1]), \\
\text{qsort}(L2, R_1, R). \quad \% \text{ cl2}. 
\end{align*}
\]
2. Logic Programming

\[
\text{partition}(P_1, P_2, P_3, P_4) \leftarrow \\
\{\} \leftarrow P_1: \\
P_3 = \{\}, P_4 = \{\}. \quad \% \text{cl13.}
\]

\[
\text{partition}(P_1, P_2, P_3, P_4) \leftarrow \\
\{X|Xs\} \leftarrow P_1, A \leftarrow P_2, \\
X \leq P_2: \\
P_3 = \{X|\text{Lower}\}, P_4 = \text{Upper}, \\
\text{partition}(Xs, A, \text{Lower}, \text{Upper}). \quad \% \text{cl14}
\]

\[
\text{partition}(P_1, P_2, P_3, P_4) \leftarrow \\
\{X|Xs\} \leftarrow P_1, A \leftarrow P_2, \\
X > A: \\
P_3 = \text{Lower}, P_4 = \{X|\text{Upper}\}, \\
\text{partition}(Xs, A, \text{Lower}, \text{Upper}). \quad \% \text{cl15}
\]

\text{Program 2.4. The standard form of the clauses in Program 2.3.}

2.5.2 Parlog Goal Resolution

The evaluation of a Parlog program is a variant of the resolution principle that was defined in Section 2.2.2. The selection of goals from the current goal statement proceeds in parallel. The selection of a clause with which to reduce a goal is a committed choice. The unification between a goal and a clause head is restricted, in that input matching and test unification are used for input arguments and the output unification takes place after commitment.

The evaluation of an atomic Parlog goal \(A\), when the current state of computation is \(<G;\theta>\), consists of the following phases:

A) \textit{Clause Selection Phase}

During the clause selection phase, clauses are tested in parallel for candidacy. For each clause \(C_j\):

\[H_j \leftarrow G_j : B_j\]

where \(H_j\) has the same relation name and arity as \(A\), the following actions take place:

1. transform \(C_j\) into a standard form and rename the variables to be different from the variables in the current goal statement. Suppose the resulting clause is:
2. Logic Programming

\[ C_j : H_j' \leftarrow G_j : B_j' \]

where \( \{A_1, ..., A_k\} \) are the input head arguments in \( H_j' \), \( \{A_k+1, ..., A_n\} \) are the output head arguments in \( H_j' \), \( G_j' \) is the conjunction

\( \langle \text{input matching and test unification goals} \rangle, G_j \)

and \( B_j' \) is the conjunction

\( \langle \text{output unification goals} \rangle, B_j \)

2. unify \([A]0\) and \( H_j' \) with a most general unifier \( \alpha_1 \). Since the arguments of \( H_j' \) are all new and distinct variables, \( \alpha_1 \) can only bind variables in \( H_j' \) and not in \([A]0\).

3. evaluate the goals in the guard of \( C_j' \) after applying the substitution \( \alpha_1 \) (i.e. \([G_j']\alpha_1\)), resulting in a substitution \( \alpha_2 \). This includes the evaluation of the input matching and test unification goals as well as the original guard goals in \( G_j \). As in the previous step, \( \alpha_2 \) can only bind variables that are local to \( C_j' \).

A Parlog clause is called a candidate clause when the input unification (i.e. the unification of the input arguments) and the guard evaluation succeed. That is when the evaluation in step 3 above succeeds. If the input unification or the guard evaluation fails, the clause is a non-candidate clause. Moreover, input unification, and similarly the guard evaluation, may suspend if it cannot proceed without binding variables in the input arguments of the goal \([A]0\). If this is the case and the clause is not a non-candidate, the clause is said to be suspended.

B) Commit Phase

A Parlog clause can be used to reduce a goal if it is a candidate clause. In the commit phase, the evaluation commits to an arbitrary candidate clause; other candidate clauses are dismissed. If there are no candidate clauses for the goal and at least one clause is suspended, the commit phase suspends until all the clauses become non-candidates, at which case the commit phase fails, or at least one clause becomes a candidate and the evaluation commits to one of them.

C) Reduction Phase

The reduction phase is begun after a clause is selected in the previous phase. In the reduction phase, output unification is performed for the clause and the goal reduces to its body. The new state of computation \( \langle G', \theta' \rangle \) results from replacing \( A \) in \( G \) by \( B_j \), and \( \theta' \) is the composition \( \theta \cdot \alpha_1 \cdot \alpha_2 \cdot \alpha_3 \), where \( \alpha_3 \) is the substitution resulting from \( \langle \text{output unification goals} \rangle \alpha_1 \cdot \alpha_2 \).
Since the evaluation commits to one clause, it is also committed to the effects of the corresponding output unification. There is no backtracking on the choice of the clause, so the bindings made to variables in output arguments are never retracted.

Resolution of a Parlog goal *succeeds* if the above three phases terminate successfully. A goal resolution *fails* if the commit phase fails or if $\theta \cdot \alpha_1 \cdot \alpha_2 \cdot \alpha_3$ fails. Moreover, the resolution of a goal *suspends* when the commit phase suspends.

A Parlog computation may be *successful* or *unsuccessful*. A computation may also *deadlock* if every attempt to reduce a goal is suspended.

For example, an attempt to reduce the `qsort` goal in the query "`qsort(X1, S, [])`, `X1 = [2,3,1]`" to Program 2.4 suspends until `X1` is sufficiently instantiated. The goal `X1 = [2,3,1]` is reduced by unifying `X1` to `[2,3,1]`. `qsort([2,3,1], S, [])` can be then reduced using its only candidate clause: clause 2. An evaluation of the query is shown in Figure 2.5. `qs/3` stands for `qsort/3` and `p/4` stands for `partition/4`. The reduced goals in each step are underlined while the substitution produced is shown in *italic* form between the steps and is applied to the resulting goal statement in the following step.

```
qs(X1, S, []), X1 = [2,3,1]
| X1 = [2,3,1]
qs([2,3,1], S, [])

cl2  |
 p([3,1, 2, L1, L2], qs(L1, S, [2|S1]), qs(L2, S1, []))

cl5  | L2 = [3|L2']
 p([1, 2, L1, L2'], qs(L1, S, [2|S1]), qs([3|L2'], S1, []))

cl4  | L1 = [1|L1']

cl2  |
 p([1, 2, L1', L2''], qs([1|L1'], S, [2|S1]), p(L2', 3, L1'', L2''), qs(L1'', S1, [3|S2]), qs(L2'', S2, []))

cl3  | L1' = [], L2' = []

cl2  |
 p([1, 1, L1'', L2''], qs(L1'', S, [1|S3]), qs(L2'', S3, [2|S1]), p([1, 3, L1'', L2''], qs(L1'', S1, [3|S2]), qs(L2'', S2, []))

cl3  | L1'' = [], L2'' = [], L1'' = [], L2'' = []
qs([1], S, [1|S3]), qs([1], S3, [2|S1]), qs([1], S1, [3|S2]), qs([1], S2, [])
```
2.6 Don't-Care Non-Determinism in Committed-Choice Programs

Input unification, together with the committed choice of one candidate clause when several are available, provide a powerful tool for programming a "don't-care" situation in which the user does not care which solution is produced by a query if several solutions are available. A standard example for a don't-care situation is to merge two lists of elements into one list in an arbitrary order. The Parlog program for the merge relation is defined in Program 2.5. A call to merge suspends until at least one of its first two arguments is instantiated. If the first two arguments of merge are instantiated to non-empty lists, both clauses 1 and 2 will be candidates for the goal. The Parlog goal will commit to any one of them non-deterministically. Similarly when the first two arguments are instantiated to empty lists, the goal can commit either to clause 3 or clause 4.

```
merge([H | T], Y , [H | Rest]) <-
   merge( T , Y , Rest).  % c11.
merge(X, [H | T], [H | Rest]) <-
   merge(X , T, Rest).  % c12.
merge(X, [], X ).  % c14.
```

Program 2.5. A Parlog merge of two lists.

One drawback of the committed choice is that an evaluation may commit to a clause which results in failure or deadlock while other clauses can lead to successful solutions. A programming style, termed the sufficient guards property, has been defined (Gregory, 1987) to avoid failure in a committed-choice evaluation when solutions exist. The criterion is that the input matching and the guard of each clause of a relation should be sufficient to ensure that, if the clause is a candidate for a call, then either:
1. a solution to the call can be computed using that clause, or
2. no solution can be found using any other clause.

For example, the clauses in the procedure of the \textit{partition/4} relation in Program 2.3 have sufficient guards; the \textit{partition([2,4], 3, Lower, Upper)} goal would succeed producing the solution \{\texttt{Lower/[2], Upper/[4]}\}.

The above criterion does not avoid the situation when an evaluation deadlocks and no solution is computed. For example, consider the query \textit{partition(X, 3, [2], [4])}. This should succeed binding \texttt{X} to [2,4] or [4,2]. However, the evaluation results in deadlock rather than success. Deadlock is considered a semantics error in committed-choice programs.

\section*{2.7 Stream And-Parallelism in Committed-Choice Programs}

The suspension of a goal on input unification, together with the stream and-parallel evaluation of goals in a conjunction, provide a useful programming paradigm; that of parallel communicating processes. A conjunction of goals is regarded as a system of concurrent processes, whose communication pattern is specified by the logical variables shared between the goals. Processes communicate by partially instantiating shared variables and synchronize by waiting for variables to be instantiated. The initial network of processes is represented by the query to the program.

The possible behaviours of a process are specified by guarded Horn clauses. The head and guard of a clause specify the conditions under which the clause can be used to change the state of the process, while the body specifies the new process's state. A process can halt (empty body), change state (one goal in the body), or split into several concurrent processes (more than one goal in the body). Once a process changes its state, its previous state cannot be restored and hence other processes can trust its behaviour.

The process interpretation of concurrent logic programs has made logic programming applicable to a wide range of new applications: those which can naturally be expressed as systems of concurrent, communicating processes. For example, Foster (Foster, 1988) illustrated how to use Parlog in systems
programming, and Davison (1989) developed a higher-level language (Polka) on top of Parlog combining the concepts of concurrent logic programming and object oriented programming. Other applications such as specification, verification and simulation of communication protocols (Gregory et al, 1985), as well as discrete event simulation (Broda and Gregory, 1984; Davison, 1988) have also been developed. In Chapter 5, the discrete event simulation of systems of communicating processes is explained.

2.8 Committed-Choice Programming for Search Problems

The parallel search for a candidate clause, together with the committed choice, provides a limited form of or-parallelism; this is termed committed or-parallelism (Gregory, 1987). The guards of alternative clauses are evaluated concurrently until one of them successfully terminates and the evaluation commits to that clause; the evaluation of sibling guards (guards of other clauses in the procedure) is then aborted. Committed or-parallelism is an or-parallel search for at most one solution, in contrast to full or-parallelism, where all solutions are computed concurrently.

As explained in Section 2.3.1, don't-know non-determinism liberates the user from explicitly programming search. If the evaluation fails when using one clause, other clauses may produce solutions. The committed choice, on the other hand, requires an explicit programming of search. A standard programming technique is to simulate an or-parallel search by an and-parallel evaluation of a conjunction of goals, each is examining a search path. A list of all solutions is then produced.

Based on this technique, several compilation methods have been introduced for automatically compiling search type programs, that are written in pure Horn clauses, into deterministic programs in a committed-choice language. Ueda (Ueda, 1986; Ueda, 1987), for example, proposed a continuation-based method for compiling a Horn clause search program into a GHC (Ueda, 1985) program which returns a list of all the solutions of the original program. Different search paths are examined by independent and-parallel goals, while the conjoined goals in the query of the original program are evaluated sequentially from left to right by passing a "continuation" around.

For instance, the delete relation in Program 2.6 (previously defined in Section 2.1) can be transformed by the continuation-based method into the GHC program
illustrated in Program 2.7. The GHC program below, can be read as a Parlog program after replacing "-:" by "<:-", and "|" by "|". Additionally, all arguments should be declared as input.

Search corresponding to the two clauses of delete is performed by the conjunction of goals d1 and d2. Their arguments are as follows:

1. the second argument of the original program (the input).
2. the continuation.
3. the head of the list of solutions.
4. the tail of the list of solutions.

```
delete(X, [X|Xs], Xs).                   % c11.
delete(X, [Y|Z], [Y|Z1]) :-
    delete(X, Z, Z1).                  % c12.
```

**Program 2.6.** A Prolog delete program.

A call to the transformed program would be:

```
?- d(List, '10', Solutions, [])
```

which returns the result equivalent to the Prolog goal:

```
?- bagof((X, Xs), delete(X, List, Xs), Solutions)
```

That is, Solutions would be instantiated to the list of all terms, each of the form (X, Xs). For example, if List = [2, 1], then Solutions would be bound to [(2, [1]), (1, [2])] (see Figure 2.1 for the search tree of the query).

Because of the call to the GHC primitive otherwise, the second clause in each of d1 and d2's procedures will be tried for candidacy only when the first clause is non-candidate. That is, when the unification of the input argument with the first argument in the head of the first clause fails. For instance, when List is non-empty, d1 is reduced using its second clause without producing solutions (SO = SI).

```
d(List, Cont, SO, S2) :- true I 
    d1(List, Cont, SO, S1),
    d2(List, Cont, S1, S2).
```

```
d1([X|Xs], Cont, SO, S1):- true I contd(Cont, X, Xs, SO, S1).
```

- 40 -
Program 2.7. A continuation-based transformation for the `delete` program.

Tamaki (1986) proposed another method, called the *stream-based* method, in which dependent conjoined goals (i.e. goals which are sharing variables) in the query of the original program are evaluated sequentially from left to right while the independent ones can be evaluated in parallel, extracting more and-parallelism than the above method. However, this method usually entails more run-time overhead than the continuation-based method.

Both the continuation-based and the stream-based methods are only applicable to transforming a restricted subset of Horn clause programs, in which the arguments of every goal appearing in the program can be classified into input arguments and output arguments. When a goal is to be reduced, its input arguments must have been instantiated to ground terms and when the goal succeeds it must instantiate its output arguments to ground terms. In the `delete` relation above, the second argument is classified as input while the first and last arguments are classified as output.

Another programming technique for solving search problems in committed-choice languages was introduced in (Okumura and Matsumoto, 1987). It is called *layered stream programming*. Suppose the solution for a search problem can be represented by a list. A Prolog program first determines the head of the list and then generates its tail. Since there can be more than one possible tail for the same head, a sequential Prolog implementation selects the tails one by one using a backtracking mechanism while an or-parallel implementation may produce them simultaneously. Using the layered stream method, the solution can be expressed by a pair, \(H^*T_s\), consisting of a head \(H\) and a stream \(T_s\) of all possible tails. When there is more than one possible head, the set of all solutions is represented by

\[
[H_1^*T_{s1}, H_2^*T_{s2}, ..., H_n^*T_{sn}]
\]

This type of structure is called a *layered stream* because an element in a layered stream may also include other layered streams. Hence, each \(T_{si}(i < n)\) in the
above representation can be a layered stream. For instance, the set of permutations of \( \{1, 2, 3\} \) can be expressed as:

\[
\begin{cases}
1 \times [2 \times [3 \times \text{begin}], 3 \times [2 \times \text{begin}]], \\
2 \times [1 \times [3 \times \text{begin}], 3 \times [1 \times \text{begin}]], \\
3 \times [1 \times [2 \times \text{begin}], 2 \times [1 \times \text{begin}]]
\end{cases}
\]

where "\text{begin}" marks the deepest end of each solution.

The layered stream method provides a large degree of parallelism. All the processes corresponding to possible elements in the problem domain become active almost simultaneously, and a process generates the head of a layered stream regardless of the value of the tail. However, if there is no tail compatible with this head (i.e. there is no solution starting with this head), then having produced the head and transmitted it to other processes is a wasted computation. This represents a trade off between supporting a large degree of parallelism and the overhead caused by unnecessary generation of the partial data.

The layered stream method is a different, but more complicated, programming style from the one logic programmers normally use when programming search problems in committed-choice languages. Moreover, even in the simple examples described in (Okumura and Matsumoto, 1987), the layered stream method does not present a uniform programming style since the generated set of processes in some problems represents the configuration of the problem and in other problems represents the configuration of the solution.

Finally, only few search problems have been successfully programmed using this new method; specifying the class of problems for which the method could be effective has been recently investigated (Tick, 1991).

### 2.9 Combining Stream And-Parallelism and Don't-Know Non-Determinism

Several attempts have been recently made to integrate the search benefits of don't-know non-determinism with stream and-parallelism. This is attractive for two reasons:

1. more parallelism can be exploited in the kinds of application currently
2. Logic Programming

written in Prolog-like languages.

(b) combining don't-know non-determinism with a concurrent programming style may open up interesting new application areas to logic programming.

These attempts are summarized below together with their limitations.

2.9.1 Parlog's Set Constructor Primitives

In (Clark and Gregory, 1986), two set constructor primitives for Parlog were introduced to enrich Parlog with search capabilities. The call:

```
set(List?, Term?, Conj?)
```

accepts a term Term and a conjunction Conj of Prolog calls as input arguments. It then generates a list List of copies of substitution instances of Term, one for each successful evaluation of Conj. List is incrementally generated as the different solutions to the conjunction are found.

For instance, a call `set(List, (X, Xs), delete(X, [2, 1], Xs))`, where `delete` is as defined in Program 2.1, would incrementally bind List to `[(2, 1), (1, 2)]`.

The subset primitive:

```
subset(List?, Term?, Conj?)
```

accepts a term Term, a Prolog conjunction Conj, and a list List of variables. The evaluation of subset is a lazy constructor of the solutions to the Prolog query. It is not allowed to run ahead of the input stream of variables. When a variable is supplied on List, subset binds the variable to a copy of the next value of Term. If there are no more solutions, it binds the variable to the constant "end". Therefore,

```
subset([X1, X2, X3], (X, Xs), delete(X, [2, 1], Xs))
```

would result in the substitution:

```
[X1/(2, 1), X2/(1, 2), X3/end]
```

The Parlog set constructors only provide a restricted communication between Parlog and Prolog. During the evaluation of a call to a set constructor, the only communication between the Parlog and the Prolog computation is List; the arguments of the Prolog goals in Conj should be sufficiently instantiated before the
execution of the set constructor and thus cannot be incrementally supplied by the Parlog computation.

2.9.2 Parlog and Prolog United

A new Parlog primitive, prolog-call(Call) was introduced in (Clark and Gregory, 1987) to invoke a Prolog goal Call from a Parlog program while allowing the arguments of Call to be incrementally instantiated during the evaluation. Several conjunction operators were also introduced to combine Parlog and Prolog goals in a parallel conjunction in the body of the clauses in a Prolog program. These operators allow different types of communication between the Parlog and the Prolog goals. For instance, the operator "::" in

```
Prolog-conjunction :: Parlog-conjunction
```

allows Prolog-conjunction and Parlog-conjunction to be either independent or to share variables that can only be instantiated deterministically. The "<>" operator allows a Prolog goal in Prolog-conjunction to eagerly make a non-deterministic binding to a shared variable with Parlog-conjunction. If a Prolog goal fails or if Parlog-conjunction fails, the backtracking within Prolog-conjunction might undo this binding. In this case, Parlog-conjunction should be 'rolled back' to its state before the non-deterministic binding was made.

The "<<" operator provides a lazy non-deterministic communication between Parlog and Prolog. If Prolog-conjunction attempts to make a non-deterministic binding to a shared variable with Parlog-conjunction, it is suspended until Parlog-conjunction terminates or suspends. Then, the state of Parlog-conjunction is saved and the Prolog computation continues. By delaying the non-deterministic binding, the evaluation of the Parlog goals may change this binding into a deterministic one and hence eliminates the need to interrupt the Parlog computation and to save its state, which is the case when using the "<>" operator.

We believe that introducing several conjunction operators for different purposes makes the resulting language difficult to learn and hence to use. It is also not clear if these operators can be implemented efficiently. Moreover, combining Parlog and Prolog goals by several operators or primitives results in a non-integrated language with duplicate features that are existing in both languages.
2.9.3 A Parallel Logic Programming System Based on Precise Type and Mode Declarations

It has been argued (Somogyi, 1989) that the intelligent backtracking algorithms which support restricted and-parallel execution of logic programs can be extended to handle stream and-parallelism if the system knows at compile time the producer of every variable. Such information will also increase the precision (intelligence) of the algorithms.

A system of precise mode and type declarations for logic programs was proposed in (Somogyi, 1989), whereby each variable in a goal statement can have only one producer; other goals sharing the same variable are denoted as consumers of the variable's binding. An intelligent backtracking algorithm was then defined (Somogyi, 1989) which evaluates conjoined goals concurrently. It starts the evaluation of a consumer as soon as the value it is to consume is produced. When the computation fails, the system backtracks to the relevant producer and restarts the evaluation of all the affected goals.

The resulting logic programming system combines stream and-parallelism and don't-know non-determinism in an elegant manner. However, not only the mode of use of each predicate's argument should be defined as in Parlog, but also the mode of use of all sub-terms in a structured term argument. Moreover, in the computational model of the system, goal frames (i.e. processes) of the terminated goals are kept in the system in case they are required to recover from future failure, while in committed-choice systems a process can be replaced by other processes as soon as it commits to a clause since it cannot produce alternative bindings for its output arguments. As an optimization, the backtracking algorithm has been modified to be able to remove a goal provided that the compiler can verify whether the corresponding relation is stable, i.e. it provides at most one binding for an output variable.

2.9.4 Parallel NU-Prolog

NU-Prolog (Thom and Zobel, 1986) is a sequential implementation of Prolog which allows coroutining. Therefore, NU-Prolog programs may exhibit don't-know non-determinism, implemented by backtracking. In order to incorporate
stream and parallelism, the language was extended with optional syntactic annotations (Naish, 1988) which specify the input and output arguments of a relation and declare the relation as being deterministic (i.e. at most one clause can match with the goal) or don't-care non-deterministic. For example, the partition relation in Program 2.8 has a lazyDet declaration which specifies that the relation is deterministic with its first two arguments being input and the last two arguments being output. Program 2.8 also includes an eagerDet declaration for the merge relation which specifies that the relation is don't-care non-deterministic with its first two arguments being input and its third argument being output.

lazyDet partition(I, I, O, O).
partition([], _ [], []).
partition(Xs, A, X.Lower, Upper):- X ≤ A, I,
    partition(Xs, A, Lower, Upper).
partition(Xs, A, Lower, X.Upper):- X > A, I,
    partition(Xs, A, Lower, Upper).

eagerDet merge(I, I, O).
merge(H.T, Y, H.Rest):- !, merge(T, Y, Rest).
merge(X, H.T, H.Rest):- !, merge(X, T, Rest).
merge([], Y, Y):- !.
merge([X], Y, X):- !.

Program 2.8. The partition and merge relations in Parallel NU-Prolog.

The extended language is called Parallel NU-Prolog in which deterministic and don't-care non-deterministic goals can run concurrently, similar to committed-choice languages. However, Parallel NU-Prolog has a stricter delaying mechanism; if an input variable in the head of a clause occurs in one of its guard calls, it is forced to be ground. A guard of a clause is the conjunction of goals before a cut "!" operator.

Moreover, a don't-know non-deterministic goal can run in parallel with don't-care non-deterministic ones only if both types of goals are not sharing variables. Otherwise, only one type of goals can run at a time. This imposes restrictions on the communication between the don't-know non-deterministic goals and the don't-care non-deterministic ones.
2.9.5 CP[$\downarrow$, $\|$, $\&$, $;$]

CP[$\downarrow$, $\|$, $\&$, $;$] (Saraswat, 1987a) is a parallel logic programming language which combines don't-know non-determinism and stream and-parallelism by allowing two kinds of commit operators to be used in guarded Horn clauses: don't-know commit ('$\&$') and don't-care commit ('$|$'). Other annotations in the language, such as '$\downarrow$' and '$;$' are irrelevant to our discussion.

On commitment, the don't-care commit forces the abandonment of alternative clauses for a goal, as in committed-choice languages, while the don't-know commit splits the computation into two disjunctive copies: one reflecting the bindings produced by the chosen clause, and the other reflecting the remaining alternative clauses for the goal. Computation succeeds if any one of the or-disjuncts succeeds.

Copying an entire computation is a very costly operation. Moreover, it may lead to wasteful duplication of effort if there is a goal which is unaffected by the bindings being published. This goal will have the same sequence of reductions in both of these branches. Thus, the language provides a construct, called the block ('[...']') which specifies the extent to which copying, and the propagation of bindings, is done at $\&$-commit time. When the computation terminates in two conjoined blocks, the two search branches from both blocks are merged, and their substitutions are composed.

CP[$\downarrow$, $\|$, $\&$, $;$] includes a variety of control constructs which supports a wide range of applications (Saraswat, 1987b). However, it is unlikely to provide an efficient implementation for a language with all these control constructs. Moreover, we believe that it is difficult for a user to learn all the language constructs in order to be able to utilize them properly.

2.9.6 P-Prolog

P-Prolog (Yang, 1986) introduced a new control concept in logic programming languages, namely the exclusive relation of guarded Horn clauses, in order to synchronize the communication among concurrently executing goals.

As in Parlog, a clause can be used to reduce a goal if it is a candidate clause. A
clause is said to be a candidate clause for a goal if it succeeds in unifying with the goal and successfully executes its guard. Using syntactic annotations, it can be pointed out in the program that a set of clauses should be exclusive in reducing a goal. That is, the goal can be reduced when it has at most one candidate clause from this set of clauses; otherwise, the goal is suspended.

A procedure in P-Prolog comprises subsets of guarded Horn clauses. The exclusive relation is defined among the different subsets while clauses within the same subset can be non-exclusive. In order to reduce a goal, all clauses are checked in parallel for candidacy and one of the following situations may result:

(a) there is more than one candidate clause for the goal in different subsets. The goal is then suspended.

(b) there is one or more candidate clause(s) for the goal in only one subset. The goal is then reduced by all its candidate clauses, possibly producing alternative solutions.

(c) all clauses are non-candidates and the goal fails.

P-Prolog combines stream and-parallelism with don’t-know non-determinism in an elegant and integrated manner. However, the language has the following drawbacks:

(a) any number of goals can be subject to an or-parallel execution at the same time, and copies of the entire computation is then required for each branch of the search tree. In her thesis (Yang, 1986), Yang described an abstract computational model for P-Prolog which reduces the amount of copying. However, having several don’t-know non-deterministic goals at a time leads to an amount of copying that significantly affects the efficiency of the implementation.

(b) when there is more than one candidate clause for a goal within the same subset of clauses, then the goal is subject to an or-parallel execution. A Parlog goal with more than one candidate clause, such as `merge([1, 1, Z])` for Program 2.5, will commit to an arbitrary one of its candidate clauses. In P-Prolog, such a goal remains suspended since it has more than one candidate clause that should be exclusive.

A don’t-care situation can be implemented in P-Prolog by a deterministic program in which the guards should be sufficient to ensure that at most one clause at a time can unify with the goal. Another solution is to use the P-Prolog primitive other for sequential exclusive checking. In order to reduce a goal for Program 2.9, clauses 1 and 2 are first tried for candidacy. If only one clause is a candidate, the goal is
reduced using that clause. If both clauses are non-candidates or both are candidates, clauses 3 and 4 are tried for candidacy. For example, merge([], [], Z) is reduced using clause 2 without trying clauses 3 and 4 for candidacy. merge(X, [], Z) will not suspend because both clauses 1 and 2 are candidates; it is rather reduced using clause 4 since clause 3 is a non-candidate.

merge([H|T], Y, [H|Rest]) :-
   merge(T, Y, Rest). % c11.
merge([], Y, Y). % c12.
merge(X, [H|T], [H|Rest]) :- other:
   merge(T, X, Rest). % c13.
merge(X, [], X). % c14.

Program 2.9. A P-Prolog merge of two lists.

2.9.7 The Basic Andorra Model

The basic Andorra computational model (Yang, 1988) was proposed by Warren to transparently extract stream and-parallelism (as well as or-parallelism) from Prolog programs. We shall refer to this model as Pure Andorra. The model combines the idea of suspending goals until they are deterministic, similar to the exclusive check in P-Prolog, with a novel "deadlock breaking" policy to realize don't-know non-determinism.

A goal is deterministic if there is at most one unifiable clause with it. This is checked by simple run-time tests, intended to be deduced by a compile-time analysis for the heads of the clauses as well as any primitive test goals in their bodies.

In Pure Andorra, execution alternates between two phases. In the and-parallel phase, all deterministic goals are evaluated concurrently. When no deterministic goals remain, a non-deterministic phase is started in which the leftmost goal is selected and a choice point created for it. For each search branch, a new and-parallel phase is begun, running all deterministic goals until they are exhausted.

Executing goals concurrently during the and-parallel phase is feasible because each reduced goal computes no more than one solution as in committed-choice programs. However, unlike committed-choice programs, an evaluation cannot result in failure or deadlock while solutions exist. An evaluation fails only if there
is no solution specified according to the declarative semantics of the program and, when the and-parallel phase terminates with non-deterministic goals being suspended, a choice point is created for the leftmost goal searching for possible solutions. For instance, it was explained in Section 2.6 how the Parlog query \( \text{partition}(X, 3, [2], [4]) \) for Program 2.3 will deadlock. The same relation is defined in Prolog in Program 2.10. Using an Andorra system for the Prolog Program, the \text{partition} goal will first suspend during the and-parallel phase. Then, it will be non-deterministically reduced in the non-deterministic phase using clauses 4 and 5. For each search branch, an and-parallel phase will begin in which goals are reduced in parallel. The Andorra evaluation of the query is illustrated in Figure 2.6, where \( p/4 \) represents \( \text{partition}/4 \).

![Diagram of Andorra evaluation](image)

**Fig 2.6. The Andorra evaluation of the goal \( \text{partition}(X, 3, [2], [4]) \).**

### 2.9.7.1 The Basic Andorra Model for Executing Prolog Programs

Pure Andorra runs non-deterministic programs in a coroutining manner whereby non-deterministic goals are executed after deterministic ones. This "lazy nondeterminism" can significantly reduce the search space in many cases (Clark and Gregory, 1987) resulting in a more intelligent problem-solving behaviour than either a sequential execution as in Prolog, or a fully parallel evaluation as in CP[\( \downarrow \), \( \land \), \&], or P-Prolog. Consider for example the following query:

```prolog
nondet_goal1, nondet_goal2, det_goal
```

where "," is the Prolog conjunctive operator, \( \text{nondet_goal} \) \((i = 1, 2)\) is a non-deterministic goal with several solutions, and \( \text{det_goal} \) is a deterministic goal with a single solution. The Prolog execution model will repeatedly execute \( \text{det_goal} \) on
each successful branch of "nondet_goal1, nondet_goal2". One should observe that a textual reordering of the Prolog goals cannot in general get the same effect because determinacy of goals is a dynamic property rather than a static one.

An eager parallel evaluation may also execute det_goal on each successful branch of "nondet_goal1, nondet_goal2", depending on the order in which the conjoined goals are reduced which can be different each time the query is evaluated. However, Pure Andorra will perform the computation of det_goal only once, before reducing the non-deterministic goals.

An extra bonus of the lazy non-determinism is that it makes it more likely that a failure is encountered immediately after the choice that caused it, while an eager evaluation may make several choices (by non-deterministic goals) before a failing goal is encountered; a chronological backtracking scheme will then backtrack to the most recent choice which may well not be the one responsible for the failure.

However, Pure Andorra does not fully subsume Prolog. Prolog programs may include goals to relations with side effects, such as write/1 or var/1. In order to achieve the same result as Prolog, these goals should not run in parallel with each other or with other goals in the conjunction; the evaluation of the goals to the right of one with side effects should be delayed until the latter terminates successfully. Consider, for example, the following conjunction of goals:

\[ \text{write}(X), \text{g}(X) \]

if \( X \) is unbound before the evaluation of the conjunction, then a Prolog evaluation would output an unbound variable before evaluating \( \text{g}(X) \). An Andorra evaluation of the same conjunction might first evaluate \( \text{g}(X) \) which might instantiate \( X \). \( \text{write}(X) \) would then output the binding of \( X \).

Another situation in which the Andorra evaluation does not achieve the same result as Prolog is illustrated in the following program.

\[
\text{c}(X) \leftarrow \text{compute1}(X). \\
\text{c}(X) \leftarrow \text{compute2}(X). \\
\text{compute1}(b). \\
\text{compute2}(c). 
\]
2. Logic Programming

\[ p(a) :- p(a). \]

An Andorra evaluation of the query \( c(X), p(X) \) would lead to non-termination while the Prolog evaluation would fail. Using an Andorra implementation, \( c(X) \) will suspend since it is non-deterministic while \( p(X) \) will be deterministically reduced to \( p(a) \) after unifying \( X \) with 'a'. \( p(a) \) is also deterministic and its evaluation will not terminate. A Prolog computation first evaluates \( c(X) \), binding \( X \) to either 'b' or 'c', then attempts to reduce \( p(b) \) or \( p(c) \) which both fail.

### 2.9.7.2 The Basic Andorra Model for Executing Committed-Choice Programs

Pure Andorra can run a deterministic Prolog program with behaviour similar to that of a committed-choice language. For instance, Program 2.10 defines the quick sort Prolog program which was previously defined as a Parlog program in Section 2.4. The evaluation of the query

\[ \text{qsort}(X_1, S, [], X_1 = [2,3,1]) \]

using the Andorra model is completely deterministic and behaves exactly the same as the Parlog evaluation that was illustrated in Figure 2.5.

```prolog
qsort([], R, R). % c11.
qsort([X|Xs], R0, R) :-
    partition(Xs, X, L1, L2),
    qsort(L1, R0, [X|R1]),
    qsort(L2, R1, R). % c12.

partition([], _, [], []). % c13.
partition([X|Xs], A, [X|Lower], Uppr) :-
    X \leq A, % c14.
    partition(Xs, A, Lower, Uppr).
partition([X|Xs], A, Lower, [X|Upper]) :-
    X > A, % c15.
    partition(Xs, A, Lower, Upper).

Program 2.10. The Prolog quick sort program.
```
However, Pure Andorra does not fully subsume committed-choice languages for the following reasons:

1. **Committed Choice**
   As in P-Prolog, Pure Andorra does not directly support don't-care non-determinism since the only synchronization mechanism for the communication among conjoined goals is the exclusive relation. In a committed-choice language, don't-care non-deterministic goals reduce by making a committed choice while in Andorra they will be delayed until the non-deterministic phase and then will be evaluated using don't-know non-determinism, which is not what is required.

2. **Forcing Suspension**
   In committed-choice languages, a goal is supposed to suspend until its arguments are sufficiently instantiated, e.g. `qsort(X, R0, R)` should suspend until `X` is bound. Pure Andorra may evaluate such a goal in the non-deterministic phase, and "guess" the value of `X` instead of waiting for it, as is desired. Similarly, primitive goals like `X > A` should suspend until its arguments are bound.

These considerations suggest that the Prolog syntax in Pure Andorra is insufficient if it is desired to support applications that are currently expressible in committed-choice languages.
Chapter 3

Pandora: The Language

In the previous chapter, it has been explained why committed-choice logic programming languages are not suitable for applications that require a don't-know non-deterministic search through multiple solutions. Several attempts for combining stream and-parallelism with don't-know non-determinism have also been described. The last attempt discussed was the basic Andorra model which can run most of the Prolog don't-know non-deterministic applications in an efficient coroutining manner, as well as running deterministic programs in a behaviour similar to committed-choice languages. However, the basic Andorra model subsumes neither Prolog nor committed-choice languages for reasons that were listed in Chapter 2.

In this chapter, we propose Pandora: a non-deterministic parallel logic programming language which extends Parlog with a powerful deadlock handling mechanism and a simple non-deterministic fork. Pandora is not intended to subsume Prolog but rather to extend committed-choice languages with search capabilities. The language has an operational semantics that is derived from the basic Andorra model. It is hence named Pandora (Parlog + Andorra).

3.1 Syntax of Pandora Programs

There are two kinds of relation in a Pandora program: don't-care relations and don't-know relations, corresponding to the kinds of or-nondeterminism in logic programming. A Pandora atomic goal is a call to either a don't-care or a don't-know relation.
A *don't-care relation* is defined by a procedure which comprises a sequence of universally quantified guarded Horn clauses, each of the form:

\[ \text{Head} \leftarrow \text{Guard} : \text{Body} \]

Head is the head of the clause, Guard and Body are conjunction of goals forming the guard and body of the clause, respectively. '\(\leftarrow\)' denotes the implication operator, ',' is the parallel conjunction operator, and ':' is the commit operator. If Guard is an empty conjunction, the commit operator is omitted. If Guard is non-empty, then an empty conjunction in the body is denoted by the goal true. The implication operator is omitted when both Guard and Body are empty conjunctions.

Pandora is a flat language. That is, a goal in the guard can only be a call to a Pandora primitive relation, such as the inequality primitives "\(</2" and ">/2" in Program 3.1, while a body goal is any Pandora goal. For a full list of the Pandora primitive relations, the reader may refer to the user-manual for the Parlog SPM system (Foster et al, 1986), since Pandora inherited all the primitives of Parlog except for the meta-level calls call/3 and not/1.

Each procedure is preceded by a mode declaration denoting its input and output arguments similar to a Parlog mode declaration. As an example, Program 3.1 defines the \texttt{o_merge/3} don't-care relation.

\begin{verbatim}
mode o_merge(? , ? , ^).
o_merge([X|Xs], [Y|Ys], [X|Zs]) <- X <= Y:
o_merge(Xs, [Y|Ys], Zs).  % cl1.
o_merge([X|Xs], [Y|Ys], [Y|Zs]) <- X > Y:
o_merge([X|Xs], Ys, Zs).  % cl2.
o_merge([], Y, Y).        % cl3.
o_merge(X, [], X).        % cl4.
\end{verbatim}

**Program 3.1.** The \texttt{o_merge/3} don't-care relation.

A *don't-know relation* is defined by a procedure which comprises a sequence of universally quantified guarded Horn clauses. The syntax of these clauses is similar to the clauses in a don't-care relation procedure, except for the implication operator which is denoted by(':', as in a Prolog procedure) instead of '\(\leftarrow\)'. Moreover, the

\footnote{While mode declarations are optional for Parlog procedures, they are obligatory for the don't-care relations in Pandora since they are the only means of differentiating between a don't-care and a don't-know relation with empty conjunctions in the guard and body.}
procedure defining a don't-know relation is not preceded by any declaration. Program 3.2 defines the in_between/4 don't-know relation.

\[\text{in\_between}(X, Y, Z, \text{[]}): X > Z: \text{true}. \quad \% \text{cl1.}\]
\[\text{in\_between}(X, Y, Z, [Y]): X \leq Y, Y \leq Z: \text{true}. \quad \% \text{cl2.}\]

Program 3.2. The in_between/4 don't-know relation.

### 3.2 Informal Operational Semantics of Pandora Programs

As with any logic program, a Pandora program is a set of logical axioms which have a declarative reading. The ordered merge relation in Program 3.1, for example, can be logically read:

"\text{o\_merge}(X, Y, Z): if X and Y are two lists of numbers, then Z is the list of all the numbers in X and Y in ascending order ".

Similarly, the in_between relation in Program 3.2 can be logically read:

"\text{in\_between}(X, Y, Z, \text{List}): If X is greater than Z, then List is empty. If X is less than or equal to Z and Y is a number between X and Z inclusive, then List is the unary list of the number Y".

Procedurally, an evaluation of a Pandora program alternates between two phases: the and-parallel phase and the deadlock phase, which respectively correspond to the and-parallel phase and the non-deterministic phase in the basic Andorra model.

In the and-parallel phase, all deterministic goals for don't-know relations are evaluated in parallel. Moreover, goals for don't-care relations are also evaluated concurrently, using input resolution as in a committed-choice evaluation. The evaluation of an atomic goal for a don't-care relation as well as an atomic goal for a don't-know relation is explained later in the section.

The and-parallel phase results in one of the following cases:

1. an empty conjunction of goals is finally derived, in which case the computation is successful.
2. an attempt to reduce one of the goals fails, in which case the computation is unsuccessful.
3. every goal in the final conjunction is suspended. A *deadlock phase* is then begun.

Pandora provides a meta-level deadlock handler relation by which the user can explicitly program the behaviour on deadlock; this will be introduced in Chapter 6. In this chapter as well as Chapters 4 and 5, we assume that the deadlock handler relation is not defined and hence does not affect the behaviour of the deadlock phase.

When the computation deadlocks, an arbitrary goal for a don't-know relation is selected and a choice point created for it. For each search branch, a new and-parallel phase is begun, running all deterministic don't-know relation goals as well as reducible (non-deterministic) goals for don't-care relations until they are exhausted.

The computation is *successful* if an empty conjunction of goals is finally derived. If the reduction of a don't-care relation goal or a don't-know relation goal fails, the computation is *unsuccessful*. Moreover, a computation may be *deadlocked* if a deadlock phase is begun with no goal for a don't-know relation is suspended.

### 3.2.1 Evaluating a Goal for a Don't-Care Relation

As in Parlog, a clause in a don't-care relation procedure is said to be a *candidate clause* for a goal if and only if the input unification and the guard evaluation succeed. If the input unification and/or the guard evaluation fails, the clause is a *non-candidate clause*. Moreover, the clause is said to be *suspended* if neither the input unification nor the evaluation of the guard has failed and at least one of them is suspended. For instance, clauses 1 and 2 in Program 3.1 are non-candidate clauses for o_merge([], Y, Z), while clause 3 is a candidate for the goal and clause 4 is suspended.

The evaluation of a don't-care relation goal starts with an or-parallel search for a candidate clause, and one of the following cases results:

(a) *all the clauses in the goal's procedure are non-candidates*:

the goal fails.
(b) *more than one clause is a candidate*:
the goal is reduced with an arbitrary candidate clause. That is, the goal is
replaced in the current conjunction of goals by the goals in the body of the
selected clause and the output arguments in the head of the clause are unified
with the corresponding arguments of the goal.

(c) *there are no candidate clauses but there is at least one suspended clause*:
the goal suspends until it is eventually reduced with a clause which becomes
a candidate, or fails if all the suspended clauses become non-candidates.

For instance, a call `o_merge(X, Y, Z)` to Program 3.1 suspends until `X` or `Y` is
instantiated. Once `X` is bound to `['']`, clause 3 becomes a candidate and the call is
reduced. Similarly, when `Y` is bound to `['']`, clause 4 becomes a candidate and the
goal is reduced. If both `X` and `Y` are bound to `['']`, clauses 3 and 4 become
candidates and the evaluation commits to an arbitrary one of them. If `X` and `Y` are
bound to non-empty lists with the first element in each list being a numerical value,
then either clause 1 or clause 2 becomes a candidate, depending on whether `X ≤ Y` or
`X > Y` succeeds and the call `o_merge` is reduced.

A goal for a don't-care relation goal is not affected by the change of phases during
the computation; a suspended goal is re-evaluated only when a variable on which it
is suspended gets bound.

### 3.2.2 Evaluating a Goal for a Don't-Know Relation

A clause in a don't-know relation procedure is said to be a *non-candidate clause* for
a goal if and only if its head is not unifiable with the goal and/or it has a false
(unsatisfiable) guard. For instance, clause 1 in Program 3.2 is a non-candidate for
`in_between(3, 4, 5, List)` because 3 is not greater than 5.

Following the Andorra model, we wish to execute in and-parallel all the don't-care
relation goals that are deterministic. Hence, the following definition.

**Definition**
A goal for a don't-know relation `P` is *deterministic* if and only if at least `K-1`
clauses are non-candidates, where `K` is the number of clauses in `P`'s procedure.
Otherwise, the goal is *non-deterministic*. 

- 58 -
Examples:

1. \( p(X) :- b. \)
   
   A goal for the relation \( p/1 \) is always deterministic since there is only one clause in the procedure of \( p/1 \).

2. \( p(X) :- b_1. \)
   \[ p(X) :- b_2. \]
   
   A goal for the relation \( p/1 \) is always non-deterministic.

3. \( p(1) :- b_1. \)
   \[ p(X) :- b_2. \]
   
   A goal \( p(Y) \) is deterministic if \( Y \) is instantiated to a non-variable term that is not equal to 1. If \( Y \) is an unbound variable or bound to 1, the goal is non-deterministic.

4. \( p(0). \)
   \[ p(X) :- X > 0 : b. \]
   
   \( p(Y) \) is non-deterministic if \( Y \) is an unbound variable. Otherwise, the goal is deterministic.

5. \( \text{perm([], []]).} \)
   
   \[ \text{perm}(L, [X|Xs]) :- \]
   \[ \quad \text{delete}(X, L, L1), \]
   \[ \quad \text{perm}(L1, Xs). \]
   
   A goal \( \text{perm}(X, Y) \) is deterministic if at least one of the following conditions is satisfied:
   
   a) \( Y \) is instantiated to a non-variable term.
   
   b) \( X \) is instantiated to a non-variable term \( T \) and \( T \neq [] \).

The evaluation of a don't-know relation goal starts with testing whether the goal is deterministic, and one of the following cases results:

(a) all clauses in the goal's procedure are non-candidates:
the goal fails.

(b) **all clauses except one, say \( Q \) are non-candidates:**
the goal is reduced with \( C_j \).

(c) **the goal is non-deterministic:**
it suspends until it becomes deterministic, in which case either case (a) or case (b) will be satisfied, or the deadlock phase is begun.

If the computation deadlocks, an arbitrary goal for a don't-know relation is selected and a choice point created for it. For each search branch, a new and-parallel phase is begun after reducing the goal with one of the clauses in its procedure.

The role of the guards in a don't-know relation procedure should be further clarified. A clause with a false guard is non-candidate for a goal and is not used to reduce the goal in the and-parallel phase. However, a goal may be deterministically reduced with a clause whose guard includes suspended clauses. For instance, the goal \( \text{in_between}(3, Y, 5, L) \) is deterministic since the first clause in Program 3.2 is non-candidate. However, the calls in the guard of the second clause, namely \( 3 \leq Y \) and \( Y \leq 5 \) are suspended waiting for \( Y \) to be instantiated. Therefore, when reducing a don't-know relation goal with a clause, the goal is unified with the head of the clause and is then replaced in the current conjunction of goals by a parallel conjunction of the goals in the guard and body of the clause. For example when reducing \( \text{in_between}(3, Y, 5, L) \) with the second clause, \( L \) is unified with \( [Y] \) and the goal is replaced in the current conjunction by:

\[ 3 \leq Y, Y \leq 5, \text{true} \]

### 3.3 Sequential Conjunction

In addition to the parallel conjunction operator `,' Pandora goals may be conjoined with the sequential conjunction operator `&'. Both operators have the declarative reading 'and' but they have different operational semantics.

Suppose that the state of computation is:

\[ <G_1, G_2; \emptyset> \]

Then, \( G_1 \) and \( G_2 \) are evaluated concurrently, as previously explained. If the state of computation is:
<G1 & G2; \theta>

then <G1; \theta> is first evaluated. For each successful computation Q of <G1; \theta>, the evaluation of <G2; \theta \cdot \alpha> is commenced, where \alpha is the substitution produced from the computation Q.

The ',' operator is more tightly binding than '&'. For example:

G1, G2 & G3, G4 & G5

indicates that G1 and G2 are first evaluated concurrently until they terminate successfully. Then, the evaluation of G3 and G4 is begun. G5 is evaluated when all the other goals in the conjunction have succeeded. In order to override the operators' precedences, parentheses can be used, such as in the following conjunction:

G1, (G2 & G3)

which specifies that the evaluation of G2 should terminate successfully before G3 is evaluated, while G1 is evaluated in parallel with the evaluation of (G2 & G3).

Suppose that the computation deadlocks when the current conjunction of goals is:

G1, G2 & G3, G4

G1 and G2 are suspended goals while the evaluation of G3 and G4 did not commence because of the sequential conjunctions. In order to complete the evaluation of "G1, G2", a suspended don't-know relation goal (G1 or G2 or one of their descendants) is selected for a non-deterministic reduction. Only when "G1, G2" succeeds that the conjunction "G3, G4" is evaluated.

The sequential conjunction operator in a concurrent logic programming language provides an explicit means by which the user can order the evaluation of a series of actions in an easy and readable manner. The most common situation which requires such facility is the evaluation of calls with side effects in some specified order. For example, Program 3.3 defines the don't-care relation write_list/1 whose declarative reading is:

"write_list(L): L is a list whose elements are printed in order".

The write/1 and nl primitives are used as in Prolog and Parlog; they are executed immediately with no suspension. The use of the '&' operator ensures that the items
on the list are printed in the correct order.

mode write_list(?).
write_list([X|List]) <-
  write(X) & nl &
  write_list(List).
write_list([]).

Program 3.3. Program to write the elements of a list in order.

Another principal use of sequential conjunctions is to avoid *speculative computation*. That is, a computation which does not contribute to the final solution of the problem and thus should be avoided. Program 3.4 illustrates an example of such situation. The `flying_pan/3` relation is the top-level relation in a Fly-Pan Pandora system which assigns a set of available aircraft to a set of scheduled flights after taking into account several constraints. The program is explained in Chapter 4. `produce_assignment/3` may non-deterministically assign aircraft to flights, while `extract_programme/2` accepts these assignments and generates the final form of the programme that is presented to the user.

Suppose that the conjunction in Program 3.4 is a parallel conjunction. As soon as an aircraft is (deterministically/non-deterministically) assigned to a flight, it will be processed by `extract_programme`. If later in the computation `produce_assignment` retracts an infeasible assignment, `extract_programme` should also retract its partially produced programme. Alternatively, a sequential conjunction of these goals delays the evaluation of `extract_programme` until a complete solution is found.

mode flying_pan(? , ?, ^).
flying_pan(Input_flights, Input_aircraft, Flying_programme) <-
  produce_assignment((Input_flights, Input_aircraft, By_flight)&
  extract_programme(By_flight, Flying_programme).

Program 3.4. Avoiding speculative and-parallel computation in Fly-Pan.

### 3.4 Sequential Clause Search

In a Pandora procedure, the '.' following the final clause acts as the procedure terminator, while the one following the other clauses in the procedure is the parallel
search operator. In addition, Pandora provides a sequential search operator ';$'
which is less tightly binding than '.;'.

The sequential clause search affects the declarative reading of Pandora relations: ';$'
is not simply read as 'and'. A procedure of the form:

\[
\begin{align*}
  r(X1, \ldots, Xk) & \leftarrow G1: B1; \\
  r(X1, \ldots, Xk) & \leftarrow G2: B2; \\
  r(X1, \ldots, Xk) & \leftarrow B3.
\end{align*}
\]

has the same declarative reading as:

\[
\begin{align*}
  r(X1, \ldots, Xk) & \leftarrow G1: B1. \\
  r(X1, \ldots, Xk) & \leftarrow \text{not}(G1) \& G2: B2. \\
  r(X1, \ldots, Xk) & \leftarrow \text{not}(G1) \& \text{not}(G2): B3.
\end{align*}
\]

That is, to each guard must be added the negation of the guards of the clauses
preceding the last ';$' operator.

Procedurally, the clauses in a ';$'-separated group in a procedure of a don't-care
relation are searched in parallel to find candidate clause(s), as previously explained
in Section 3.2.1. A ';$' in a don't-care relation procedure specifies that the clauses
following the ';$' should not be checked for candidacy until all the clauses preceding
the ';$' are non-candidates, i.e. their input unification and/or guard evaluation have
failed.

Suppose, for example, that the following sequence of clauses constitutes a
procedure for a don't-care relation. When evaluating a call to this procedure, only
Clause1 is tried at first. If this clause is suspended, the goal is suspended. If
Clause1 is a candidate, it is selected. Only if Clause1 is a non-candidate does the
search proceed beyond the ';$'. In this event, Clause2 and Clause3 are tried in
parallel for candidacy. If at least one of these clauses is a candidate, the goal is
reduced. If none of Clause2 and Clause3 is a candidate while one of them is
suspended, the goal is suspended. Only if these two clauses are both non-
candidates that Clause4 is tried.
A ';' in a don't-know relation procedure specifies that the clauses following the ';' should not be checked for non-candidacy until all the clauses preceding the ';' are non-candidate clauses.

For example, when evaluating a don't-know relation goal for which the sequence of four clauses above constitutes a procedure, Clause1 is first checked. If Clause1 is a non-candidate, the search proceeds beyond the ';'. Otherwise, the goal is reduced with Clause1. In the former case, Clause2 and Clause3 are checked in parallel. If only one of these two clauses is a non-candidate, the goal is reduced with the other clause. If both clauses are non-candidates, Clause4 is tried. Otherwise, the goal is suspended since it is non-deterministic.

With the introduction of ';', the definition of determinism should be refined as follows.

**Definition:**
Suppose a procedure P for a don't-know relation is defined as follows:

\[
\begin{align*}
&G_1; \\
&G_2; \\
&\ldots \\
&G_n.
\end{align*}
\]

where \( G_i (1 \leq i \leq n) \) is a group of clauses separated by ';'. Then, a goal for P is *deterministic* if and only if one of the following conditions holds:

1. exactly \( K-1 \) clauses are non-candidates, where \( K \) is the number of clauses in \( G_j (1 \leq j \leq n) \) and all the clauses in the preceding groups \( G_i (\forall i < j) \) are non-candidates.
2. all clauses in \( G_1, \ldots, G_n \) are non-candidates.

There are two major advantages of providing a sequential clause search in Pandora. Namely: increasing readability and avoiding speculative computation. Programs that use sequential search operator are often easier to write, to read, and to understand than those obtaining sequencing by adding the negative of the guard goals in the
clauses preceding ".". For example, Program 3.5 defines the don't-know relation word3/3 in a word dictionary data base. The "." that is following the clause before the last specifies that if word3(X, Y, Z, E) cannot unify with any 3-letter word in the data base, then the last clause is tried, which sends an error message and terminates successfully. If the "." has not been used, all the possible combinations of bindings for the three letters in the word should have been negated in the guard of the last clause. In addition to an unreadable program, this would result in a speculative evaluation of the negated guard calls when reducing a word3 goal with the last clause in its procedure.

\begin{verbatim}
word3(a, c, t, _).
word3(a, I, I, _).
...
word3(b, i, g, _);
word3(_, _, E) :- send_err(E).
\end{verbatim}

Program 3.5. The don't-know relation word3/3 in a common word dictionary.

3.5 Soundness and Completeness of the Language

Because the Pandora inference system is based on resolution as well as input resolution, it is sound. That is, any solution that a program computes is a solution according to the declarative semantics of the program. However, as in committed-choice languages, an evaluation of a Pandora program with don't-care relations is incomplete; it may not produce all the solutions to the query. Moreover, it may result in failure or deadlock while declaratively it should produce solutions.

The "sufficient guards" property that was defined in Chapter 2 (Section 2.6) for committed-choice programs can be also used when defining don't-care relations in order to avoid failure of a Pandora evaluation when solutions exist. However, this criterion does not avoid the situation when an evaluation deadlocks and no solution is computed. Deadlock is considered a semantic error in a committed-choice program and is reported to the user as a run-time error. In Pandora, deadlock may not necessarily be a semantic error. It may rather cause the computation to fork in a non-deterministic manner. If a search branch fails or deadlocks with no suspended goals for don't-know relations, other search branches may succeed producing solution(s) to the problem. Only when all the branches fail or deadlock with all
suspended goals being for don't-care relations does the Pandora program fail or deadlock, respectively.

3.6 Determinacy Analysis

A straightforward algorithm for checking whether a don't-know relation goal is deterministic is by unifying it with the head of each clause in the first '.'-separated group in its procedure and executing the guards of the clauses. If all clauses except one are non-candidates, commit to that clause. If all clauses are non-candidates, try the next '.'-separated group. Otherwise, suspend the goal until a variable appearing in it is instantiated and then repeat the same process. When checking the last '.'-separated group, if all clauses are non-candidates, fail the goal. We call this approach the dynamic determinacy analysis.

Since the variables in the goal may be shared with other goals in the current conjunction, these variables should not get bound before the goal commits to only one clause. Suppose that the arguments of the goal are "A1, ..., Ak". Hence when testing a clause Cj for non-candidacy, instead of unifying the goal with the head of the clause, a fresh copy "Xj1, ..., Xjk" of the goal's arguments can be unified with the arguments in the head of Cj instead of the real arguments of the goal, and the guard goals are evaluated. Only when all clauses are non-candidates except one, say Ci, that "X11, ..., Xik" are unified with "A1, ..., Ak". If unification succeeds, the goal is reduced using Ci.

Given that the guard goals are only primitive goals, together with the above definition of determinism, it is possible in most cases to test whether a goal is deterministic by inspecting its arguments, i.e. without binding any variables in the goal. Several compile-time determinacy analysis algorithms (Costa et al, 1990a) have been implemented for the Andorra model which analyse at compile time the heads of the clauses as well as the primitive goals in their bodies, and produce a sequence of simple run-time tests to test the determinism of the goal and either execute the goal if it is realized to be deterministic, or suspend it on variable(s) otherwise.

For example, a determinacy analysis algorithm has been developed for Andorra-I (Costa et al, 1990b): a prototype implementation of the Andorra model, by which Program 3.6 was analysed and the decision graph in Figure 3.1 was produced.
Program 3.6. The don't-know relation cell/3.

Figure 3.1. The decision graph resulting from the determinacy analysis of cell/3.

The basic idea of the algorithm is to generate at compile-time a decision graph, formed by an 'or' of decision trees corresponding to each argument of the goal. This is represented in Figure 3.1 by dotted lines. For each tree, if all the tips correspond to committing to a clause or failing, then the tree is satisfactory. Otherwise, the algorithm will try to expand the tree by looking at other arguments until either the tree is satisfactory or no more arguments are available.

A combination of arguments is only generated if it provides new information, i.e. if some of the tips produced by the combination correspond to committing to a clause or failing, and also if it was not tried when expanding some other argument. For instance in Figure 3.1, the decision tree for the first argument is expanded with testing that the second and third arguments are not unifiable. It should be noted that
the second and third arguments are unifiable if they are unbound or if they are bound to identical terms. In both cases, the decision tree corresponding to the first argument leads to exit 5 in Figure 3.1.

The nodes of a decision graph correspond to simple tests for the bindings of the goal's arguments and the exits correspond to either verifying that the goal should fail, or commit to a clause, or wait. The latter case is when there is no way of making the goal deterministic, or when some other alternative argument can be used to verify the determinacy of the goal.

Using the same algorithms, the heads of the clauses and the guard goals in a Pandora don't-know relation procedure can be analysed. For example, a compile-time determinacy analysis algorithm for flat Pandora don't-know relations has been recently developed by Korsloot and Tick (1990). It is based on the committed-choice compilation techniques given by Kliger and Shapiro (1990). The algorithm has produced a similar decision graph for the cell/3 don't-know relation as the one in Figure 3.1.

3.7 Completeness of the Determinacy Analysis

It should be noted that the satisfiability of a clause's guard in a don't-know procedure is a logical property: a guard may be true or false even when, operationally, it would include a suspended goal. This is illustrated by Program 3.2 in which clause 2 is non-candidate for in_between(5, Y, 3) although Y is uninstantiated. Two approaches have been described for detecting deterministic don't-know relation goals: the dynamic determinacy analysis and the compile-time determinacy analysis. Both approaches should be enhanced with an intelligent analysis of the guard goals in order to determine the non-satisfiability of a guard even if it includes suspended calls. In Program 3.2, for example, the analysis should know about the transitivity of the '<=' relation.

It has been claimed that the compile-time determinacy analysis approach is in most cases more efficient than the dynamic approach since it produces simple run-time tests and does not require copying of the goal's arguments each time a clause is checked for non-candidacy. However, a compile-time analysis can produce a large number of run-time tests which are to be executed each time a goal for a don't-
know relation is evaluated. This problem is related to sophisticated indexing for traditional Prolog systems (van Roy et al, 1987), and to compiling concurrent programming languages into decision trees (Kliger and Shapiro, 1988).

In order to produce efficient code, most of the compile-time determinacy analysis algorithms are not complete. That is, the decision graph which they produce from analyzing a relation may not detect all the deterministic goals for that relation; uncommon deterministic goals may remain suspended until the deadlock phase. For example, the decision graph produced by the Andorra-I determinacy analysis algorithm for Program 3.7 below cannot detect the following deterministic goals for the relation:

\[ p(X, X), \quad p(g(X), X) \]

Therefore, these goals remain suspended until non-deterministically executed in the deadlock phases, instead of immediately failing in the and-parallel phase.

\begin{verbatim}
 p(g(a), b) :- p1. \quad \% cl1.
p(g(c), b) :- p2. \quad \% cl2.
\end{verbatim}

\textbf{Program 3.7.} The don't-know relation \( p/2 \).

### 3.8 Safety Analysis

The determinacy analysis algorithm ensures that the code produced to check the determinacy of a don't-know relation goal in the and-parallel phase does not bind any variables in the goal. Similarly, during the search for a candidate clause with which to solve a goal for a don't-care relation, a guard must not be allowed to bind any variables in the goal; such bindings must only be made after commitment.

The operational semantics of don't-care relations specifies that output unification is performed after commitment to a candidate clause. At this point, the output arguments in the head of the selected clause are unified with the corresponding arguments of the goal. Variables in the head argument terms may have been instantiated by input unification or by the guard evaluation but these bindings will not affect the output arguments of the goal until the output unification is performed.

Moreover, every guard must be \textit{safe}; that is it does not bind variables in input arguments of a goal. A guard may only bind variables appearing in the guard, body, and in the output arguments of the clause head.
Gregory (Gregory, 1987) defined a potentially unsafe guard in a Parlog clause, i.e. a guard whose execution might bind input arguments of the goal. He then designed a compile-time analysis algorithm which rejects a clause with a potentially unsafe guard. The algorithm considers deep guards as well as flat ones. That is, it allows for a user-defined relation to be called from the guard of a clause. Having only primitive calls in the guard of a clause, can greatly simplify the definition and consequently the algorithm. The following is our version of the definition applied on flat-guarded clauses in a don't-care relation.

**Definition:**

Suppose $U$ is a variable in an input argument position in the head of a clause and $G$ is the guard of the clause. Then, $G$ is potentially unsafe for the variable $U$ if at least one of the following conditions is satisfied:

1. $U$ occurs in an output argument position of a call in $G$ (e.g. $U$ is $Y + 3$).
2. $U$ occurs in at least one of the arguments of a "=" call in $G$ (e.g. $U=Y$ or $Y=U$).
3. $U$ occurs in the first argument of a "<=$ call in $G$ (e.g. $U <= Y$).
4. $U$ occurs in the second argument of a call "<=$ in $G$, and there is a variable $V$ in the first argument of that call for which $G$ is potentially unsafe, after excluding the "<=$ call under consideration (e.g. $[V|Y]<=[U|Z], V=3$).

A guard call may directly bind an input argument of the goal if one of the first three conditions above is satisfied. The fourth condition detects the case when an input argument is connected to another variable which may get bound by a guard call. Consider for example the conjunction of guard calls:

$$[V|Y]<=[U|Z], V=3$$

$[V|Y]<=[U|Z]$ produces the bindings $\{V/U, Y/Z\}$. Then, $V=3$ binds $V$ and hence $U$ to 3.

The following is an algorithm which only accepts a guard that is not potentially unsafe according to the above definition. `safe_guard` should be first invoked with `InputVars`: the set of all variables in input argument positions in the head of the clause, and `GuardCalls`: the set of all calls in the guard of the clause.
safe_guard(InputVars, GuardCalls)

for each $U \in \text{InputVars}$ do
{
  $U \neq$ an output argument of $P$ ($P$ in GuardCalls)
  $U \neq$ an argument of '=' in GuardCalls
  $U \neq$ the first argument of '<=' in GuardCalls

  if ($U \neq$ the second argument of $P$ in GuardCalls) and
    ($P$ is '<=') then
  {
    NewVars = Variables in first argument of $P$
    NewGuardCalls = GuardCalls - $P$
    safe_guard(NewVars, NewGuardCalls)
  }
}

3.9 Completeness of the Safety Analysis

A guard that is accepted by the algorithm in the previous section is guaranteed to be safe. However, the opposite is not true. That is, a guard can be safe although it is rejected by the algorithm as being potentially unsafe. An example of such a guard is in Program 3.8. Since $Z$ is a local variable to the clause and is not instantiated by any other goal in the guard, $X$ will not be bound by the goal. However, this represents a hypothetical situation that does not occur in real programs.

```
mode r(?).
  r(X) <- X = Z: true.
```

Program 3.8. A don't-care relation with a potentially unsafe guard in its clause.

3.10 A Comparison with the Basic Andorra Model

Pandora is a language that provides explicit control of concurrency and non-determinism. Therefore, Pandora extends the basic Andorra model in the following ways:
1. Don't-Care Non-Determinism
   Not only are deterministic goals for don't-know relations reduced in the and-parallel
phase, but also (don't-care) non-deterministic goals for don't-care relations. A non-deterministic goal for a don't-care relation is reduced by committing to an arbitrary one of its candidate clauses. For example, the goal \texttt{o_merge([], [], Z)} for Program 3.1 can be reduced by either clause 3 or clause 4, binding \texttt{Z} to \texttt{[]}.

In order to achieve a similar behaviour using the basic Andorra model for Prolog programs or using don't-know relations in Pandora, the don't-care situation should be explicitly programmed by a deterministic procedure. For example Program 3.9 below behaves similar to the don't-care relation in Program 3.1. The guards of clauses 3 and 4 are enriched by the test primitives \texttt{Y \ensuremath{\ne} \texttt{[]} and \texttt{X \ensuremath{\ne} \texttt{[]}} respectively, and clause 5 is added to handle the case when the goal's first two arguments are empty lists.

\begin{verbatim}
\texttt{o_merge([X|Xs], [Y|Ys], [X|Zs]) :- X \leq Y:}
\texttt{\hspace{1cm} o_merge(Xs, [Y|Ys], Zs). % cl1.}
\texttt{o_merge([X|Xs], [Y|Ys], [Y|Zs]) :- X > Y:}
\texttt{\hspace{1cm} o_merge([X|Xs], Ys, ZS). % cl2.}
\texttt{o_merge([], Y, Y) :- Y \ensuremath{\ne} \texttt{[]}: true. % cl3.}
\texttt{o_merge(X, [], X) :- X \ensuremath{\ne} \texttt{[]}: true. % cl4.}
\texttt{o_merge([], [], []). % cl5.}
\end{verbatim}

**Program 3.9.** The ordered merge don't-know relation.

2. **Forcing Suspension**

One of the major drawbacks of the basic Andorra model is having the exclusive relation as the sole synchronization mechanism for the communication among conjoined goals. In Pandora, a consumer goal can be defined by a don't-care relation for which the goal is forced to suspend on input unification or a guard evaluation until the value it is to consume is produced. For instance, a goal \texttt{o_merge(X, Y, Z)} for Program 3.1 suspends until \texttt{X} and/or \texttt{Y} is bound to a non-variable term. If the computation deadlocks, \texttt{o_merge} cannot non-deterministically guess the value of its input arguments since it is defined by a don't-care relation, which is the required behaviour.

Forcing suspension can also be used as a mechanism for controlling don't-know non-determinism in the computation. Consider the following program, where \texttt{p2/n} is a don't-know relation:
mode p(?, ..., \_).
p(X, ..., Y) <- constraints on X:
p2(X, ..., Y).

p2/n is introduced to the computation only when the constraints on X are satisfied. If X is not sufficiently bound to evaluate the constraints, p/n remains suspended during the deadlock phase and cannot be evaluated non-deterministically. A practical example of such a situation is illustrated when defining the time_binary_merge/3 relation in Chapter 5 (Section 5.5).

3. *Don't-Know Non-Determinism*

Pandora is not intended to subsume Prolog, but rather to extend the role of Parlog with the minimum of complication. Therefore, the textual order of goals in a Pandora conjunction is irrelevant and, when the computation deadlocks, an arbitrary don't-know relation goal can be selected for a non-deterministic reduction.

4. *Powerful Deadlock Breaking Mechanism*

In addition to creating a choice point for an arbitrary don't-know relation goal, Pandora provides other means for breaking the deadlock of the computation. These are introduced in Chapter 6.

5. *Sequencing*

Unlike the basic Andorra model, Pandora provides explicit means by which the user can order the evaluation of a series of goals as well as sequence the clause search. The advantages of supporting these features in the language were previously mentioned.

### 3.11 A Comparison with Parlog

Don't-care relations in Pandora have the same syntax and semantics as relations in Parlog. Therefore, a Pandora query with only goals for don't-care relations would behave similar to a Parlog query. However, Pandora extends Parlog with the following:

1. *Don't-Know Non-Determinism*

When the computation deadlocks, a goal for a don't-know relation is selected and a choice point created for it. The computation then forks into several search branches.
and a new and-parallel phase is begun for each search branch.

2. Powerful Deadlock Breaking Mechanism
As mentioned in the previous section, Pandora provides several ways for breaking the deadlock of the computation. These are introduced in Chapter 6.

3.12 Pandora Programming for Search Problems
As in Pure Andorra and coroutining languages, Pandora adopts an "incremental generate and test" evaluation strategy of search problems. Don't-know non-deterministic generators can be defined by don't-know relations, while consumers can be defined by don't-care relations. A goal to a non-deterministic generator lazily assigns values to variables. As soon as a value is generated, its consumers are eagerly evaluated, testing the validity of the produced value. A consumer may (don't-care) non-deterministically accept any value that satisfies certain constraints. If all the executable tests are satisfied, the computation deadlocks and a new variable is assigned a value which, as a consequence, allows other consumers to be evaluated.

The lazy (don't-know) non-determinism makes it more likely that failure is encountered immediately after the value that caused it is produced. As a consequence, several failing branches in the search tree may not be traversed.

A Pandora evaluation goes beyond a coroutining evaluation for a search problem; several conjoined Pandora goals may be evaluated in parallel, and these goals may incrementally communicate by bindings to shared variables as long as these bindings are not retractable.

Pandora also goes beyond Pure Andorra by allowing a (don't-care) non-deterministic evaluation of consumer goals, and by forcing a consumer to suspend until the value it is to consume is produced. In an Andorra evaluation, a consumer may guess the value it is to consume if the goal is non-deterministically reduced in the non-deterministic phase.

Below is a Pandora program for the n-queens problem. The program should be sufficiently simple to illustrate the efficiency gained by the Pandora evaluation strategy.
3.12.1 The N-Queens Problem

The n-queens problem is the problem of placing n queens on an n*n board whilst taking into account the following restrictions:

- each column must have exactly one queen.
- each row must have exactly one queen.
- each diagonal can have at most one queen.

A solution to the problem is a list \([Q_1, Q_2, \ldots, Q_n]\), where \(Q_j\) is the row number of the square in column \(j\) in which a queen is placed.

Program 3.10 solves the n-queens problem in Pandora. The top-level query to the program is the goal \(n\_queens(N, Queens)\). It accepts \(N\): the size of the board, and spawns three concurrently executing processes: \(generate\_list/3\), \(perm/2\), and \(safe/1\).

\(generate\_list/3\) is a don’t-care relation whose goal suspends until \(N\) is bound and then generates \(List: [1, 2, \ldots, N]\). \(perm/2\) incrementally accepts \(List\) as it is being produced by \(generate\_list/3\) and generates \(Queens\): a permutation of \(List\). Having \(Queens\) as a permutation of \([1, 2, \ldots, N]\) ensures that no two queens will be placed on the same column or the same row.

\(perm/2\) generates \(Queens\) by spawning \(N\) \(delete/3\) goals, each to select a position for a queen on the board. \(delete/3\) is the only don’t-know relation in the program. When the computation deadlocks, a \(delete/3\) goal non-deterministically selects a position for a queen on the board. If the computation fails because a queen is placed in a position that is not safe with other queens on the board, \(delete/3\) selects an alternative position for the queen.

\(safe/1\) is defined by a don’t-care relation. A \(safe\) goal cannot guess the positions of the queens on the board. It rather suspends until its argument is either instantiated to \(['']\) or partially instantiated to a non-empty list, then spawns a \(no\_attack/3\) goal for each queen in the solution to check that no other queen is placed on the same diagonal. As soon as two queens are placed on the same diagonal, \(no\_attack\) fails and the current partial instantiation of \(Queens\) is rejected.

- 75 -
mode n_queens(? , ^).
n_queens(N, Queens) :-
generate_list(1, N, List),
perm(List, Queens),
safe(Queens).

mode perm(? , ^).
perm([], []).  
perm(List, [Q|Qs]) :- List =\= [],
delete(Q, List, List1),
perm(List1, Qs).

dele te(X, [X|Ys], Ys).
dele te(X, [Y|Ys], [Y|Zs]) :-
delete(X, Ys, Zs).

mode safe(?).
safe([]).
safe([Q|Qs]) <- no_attack(Q, Qs, 1),
safe(Qs).

mode no_attack(? , ?, ?).
no_attack(_, [], _).
no_attack(U, [V|X], N) :-
no_diagonal(U, V, N),
N1 is N + 1,
no_attack(U, X, N1).

mode no_diagonal(? , ?, ?).
no_diagonal(U, V, N) <- U =\< V:
Temp is V - U,
Temp =\= N.
no_diagonal(U, V, N) <- U > V:
Temp is U - V,
Temp =\= N.

Program 3.10. An incremental generate and test algorithm for the n-queens problem.
Table 3.1 presents statistics on the number of backtracking exhibited by our program (with the heading Pandora) to find the first solution to the problem, compared with a sequential Prolog evaluation for the same problem.

<table>
<thead>
<tr>
<th>Number of queens</th>
<th>Sequential</th>
<th>Pandora</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>186</td>
<td>46</td>
</tr>
<tr>
<td>7</td>
<td>186</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>2842</td>
<td>223</td>
</tr>
<tr>
<td>9</td>
<td>7685</td>
<td>95</td>
</tr>
<tr>
<td>10</td>
<td>58986</td>
<td>276</td>
</tr>
</tbody>
</table>

Table 3.1. Number of backtracking until finding a solution to the n-queens problem.
Chapter 4

Constraint-Based Reasoning in Pandora for Solving Resource Allocation Problems

In this chapter and the following one, several programming techniques are developed based on the novel problem-solving behaviour of Pandora. These techniques are then used in programming, and efficiently running, interesting application areas in Pandora.

This chapter proposes Pandora for solving resource allocation problems. Most of the resource allocation problems in Operations Research are computationally complex and are best viewed as search problems which can be naturally stated by "incremental generate and test" Pandora programs, including optimization functions and domain-dependent heuristics. However, in the Pandora search strategy, constraints are used in a passive manner to test the feasibility of an assignment after being made. In order to increase the efficiency, a special programming technique to adopt constraint-based reasoning in a Pandora program is introduced.

4.1 The Resource Allocation Problem

The resource allocation problem is the problem of assigning limited resources to several activities whilst taking into account a number of factors and restrictions. Decisions are to be made on assigning which resources to which activities; these decisions are represented by decision variables. Each decision variable has a set of possible values that can be assigned to it. Restrictions on these values are called
A "toy" example of a resource allocation problem is the n-queens problem, that was described in the previous chapter. The resources are the n queens which should be assigned to n squares on the n*n board. The constraints of the problem are:

- each column must be assigned exactly one queen.
- each row must be assigned exactly one queen.
- each diagonal can be assigned at most one queen.

A solution method for a resource allocation problem searches the space of possible assignments until a solution is found, that is, an assignment which satisfies all the constraints in the problem. If there are several solutions for the problem, an optimal one can be selected. If it is difficult to formalise a quantitative measure for optimality, qualitative heuristics might be used to find a good sub-optimal solution. In general, heuristics are highly domain-dependent since they require information about the particular problem for which the solution method is developed.

### 4.2 Pandora for Resource Allocation Problems

A resource allocation problem can be naturally formulated in Pandora as an "incremental generate and test" program. A don't-know non-deterministic generator assigns values to decision variables, while constraints, defined by don't-care relations, test these values as soon as possible. If all the executable constraints are satisfied, the computation deadlocks and a new decision variable is assigned a value which, as a consequence, allows other constraints to be executed concurrently. If a constraint does not hold, then the computation backtracks\(^2\) to the most recent choice which, because of the lazy non-determinism, is more likely to be responsible for the non-satisfiability of the constraint(s). An optimization function can be considered as another constraint on the decision variables; it accepts an optimal assignment while rejects a non-optimal one.

For instance, Program 3.10 in the previous chapter, is an "incremental generate and test" program for the n-queens problem. Each decision variable Qi (1≤i≤n) represents the decision of assigning a queen to a square in the ith column of the board. The initial set of possible values for Qi is [1, ..., n]: the rows in column i

\(^2\) assuming that don't-know non-determinism is implemented by backtracking.
on which a queen can be placed. \texttt{perm(List, Queens)} assigns values to the decision variables while \texttt{safe(Queens)} tests these values as they are produced. If a queen is assigned to a column \(i\) (\(Q_i\)) such that there is another queen that was previously placed on the same diagonal, \texttt{safe/1} fails and \(Q_i\) is assigned an alternative row number. There is no concept of optimality in the n-queens problem; any assignment which satisfies the constraints is a solution to the problem. Hence, no optimization function is defined.

### 4.3 Constraint-Based Algorithms for Resource Allocation Problems

As in Prolog, the search strategy in Pandora uses the "constraints" in a \textit{passive} manner to test the feasibility of an assignment after being made. For instance, the n-queens program first places a queen on the board and then checks if it cannot attack other queens that are previously placed on the board, i.e. no two queens are in the same row, column, or diagonal. However, a constraint-based algorithm, which is called \textit{local propagation} of constraints (or more accurately \textit{constant propagation}) (Sussman and Stallman, 1975), applies the constraints \textit{actively} to remove illegal values from the set of possible values for a decision variable before the variable gets bound.

The \textit{set of possible values} for a decision variable consists of those values that are not known to be ruled out by the constraints in the current state of the system's knowledge, and is initially defined by the values satisfying the unary constraints on that variable.

Suppose there is a constraint \(p(X_1, X_2)\) restricting the two decision variables \(X_1\) and \(X_2\), and \(V_1\) is in the set of possible values for \(X_1\). If there is no possible value \(V_2\) for \(X_2\) such that \(p(V_1, V_2)\) holds, then \(V_1\) is removed from the set of possible values for \(X_1\). This may lead to removing values for other decision variables related to \(X_1\) by some constraints.

If at any time only one value exists in the set of possible values for a decision variable, it is immediately assigned to that variable.

There are three possible outcomes for the computation:

(a) each decision variable is assigned a value, in which case the computation
succeeds and the resulting assignment represents the solution to the problem.

(b) at least one decision variable has an empty set of possible values, in which case the computation fails.

(c) one or more decision variables have more than one value in their sets. Then, the computation advances to the next phase.

Constant propagation is generally the simplest and most efficient method for solving constraint problems. However, if the computation terminates with one or more decision variables with sets of possible values, then the method is insufficient to assign a specific value to each decision variable; other methods are needed to solve the problem, such as the relaxation technique (Leler, 1988).

### 4.4 Constraint-Based Algorithms in Pandora

A special programming technique to adopt the active use of constraints in a Pandora program was first introduced in (Bahgat and Gregory, 1989) and is described below.

A decision variable \( X \) with a set of possible values \( \{V_1, V_2, ..., V_n\} \) is implemented by a conjunction of \( n \) goals, ensuring that the variable will be assigned exactly one value from its set of possible values:

\[
\text{cell}(O_{n_1}, V_1, X, \text{begin}, R_2), \text{cell}(O_{n_2}, V_2, X, R_2, R_3), ..., \text{cell}(O_{n_n}, V_n, X, R_n, \text{end})
\]

For instance in the \( n \)-queens problem, each column \( j \) can be represented by the following conjunction, where \( Q_j \) is the number of the row in which a queen is placed in column \( j \):

\[
\text{cell}(O_{n_1}, 1, Q_j, \text{begin}, R_2), \text{cell}(O_{n_2}, 2, Q_j, R_2, R_3), ..., \text{cell}(O_{n_n}, n, Q_j, R_n, \text{end})
\]

\text{cell/5} is a don't-know relation defined by the simple procedure in Program 4.1. The arguments for a \text{cell} goal (the \( i \)th) are: a variable \( O_{n_i} \) which can be bound to either "on" or "off", \( V_i \) which is the \( i \)th possible value for the decision variable, and the logical variable \( X \) representing the decision variable. The fourth and fifth arguments link the goals together in a chain; the ends of the chain are two distinct constants, "begin" and "end".
4. Constraint-Based Reasoning in Pandora

% cell(Oni, Vi, X, Left, Right).
cell(on, Value, Value, _, _). % cl1
cell(off, _, _, Chain, Chain). % cl2

Program 4.1. The don't-know relation cell/5.

The variable Oni remains unbound as long as the value Vi is a candidate value for X. It is bound to "on" (in clause 1) if X is assigned the value Vi, and is bound to "off" (in clause 2) if Vi is no longer a possible value for X.

The ith cell goal becomes deterministic if one or more of the following conditions is satisfied:

(a) Oni is bound.
(b) X is bound to a value Vj that is different from Vi. This eliminates its first clause, unifies Oni to "off", and unifies its fourth and fifth arguments.
(c) all the other cell goals bound their first arguments to "off" because then they will succeed and unify their fourth and fifth argument variables. This makes the ith goal cell(Onj, Vi, X, begin, end) which eliminates its second clause, unifies Oni with "on", and assigns the value Vj to X.

Condition (b) ensures that once the variable is assigned a value, all the cell goals holding the other possible values for the variable succeed using their second clauses and hence only one value is assigned to the variable. For example if a queen is placed on the jth column and ith row, i.e. Qj is instantiated to "i", then all the goals cell(Oni, Vi, Qj, Rl, Rl+1) (l ≠ i) succeed using their second clauses and unify their fourth and fifth argument variables. The ith goal cell(Oni, i, i, begin, end) succeeds using its first clause after unifying Oni with "on".

Condition (c) ensures that when there is only one possible value for a variable, that value is deterministically assigned to the variable. Moreover, if all values become invalid for the variable, each cell goal will attempt to unify its fourth and fifth arguments (using its second clause) and one of them will fail when unifying the constants "begin" and "end".

Condition (a) is used to define the constraints on the decision variables. A constraint on K decision variables is implemented by a don't-care relation(s) which
manipulates the Ons vectors ([On1, On2, ...]) that are representing the sets of possible values for the K decision variables. For example, the inequality constraint $X \neq V_i$ should remove $V_i$ from $X$’s set of possible values (if it exists). If applied on $[\text{On}_1X, ..., \text{On}_nX]$, the goal to the inequality don’t-care relation binds $\text{On}_iX$ to "off" (if it exists); it only fails if $\text{On}_iX$ exists and is bound to "on". Another example is the equality constraint $X = V_i$ which assigns the value $V_i$ to $X$ by binding $\text{On}_iX$ to "on". If $\text{On}_iX$ does not belong to $[\text{On}_1X, ..., \text{On}_nX]$ or is bound to "off", the goal fails.

Suppose $X_1$ and $X_2$ are related with a binary inequality constraint ($X_1 \neq X_2$), and their sets of possible values are represented by $[\text{On}_1X_1, \text{On}_2X_1, ..., \text{On}_nX_1]$ and $[\text{On}_1X_2, ..., \text{On}_mX_2]$ respectively. The inequality constraint states that if a value $V$ is assigned to the variable $X_1$ (or $X_2$), then $V$ should be removed from $X_2$ (or $X_1$)’s set of possible values (if it exists). The inequality constraint is implemented by spawning a $\text{not_eq}(\text{On}_1X_1, \text{On}_1X_2)$ goal for each value $V$ that exists in the sets of possible values for both variables, where $V$ is in the $i$th position in $X_1$’s set and the $j$th position in $X_2$’s set.

The $\text{not_eq}/2$ don’t-care relation is defined in Program 4.2. Its goal suspends until at least one of its arguments is bound. If an argument is bound to "on", the goal terminates after binding its other argument to "off". If one of the arguments is bound to "off", the goal terminates successfully no matter what value is (or will be) assigned to its other argument since the $\text{not_eq}$ constraint is no longer required.

```
mode not_eq(? , ?).
not_eq(off , _).
not_eq(_, off).
not_eq(on, On2) <- On2 = off.
not_eq(On1, on) <- On1 = off.
```

**Program 4.2.** The don’t-care relation $\text{not_eq}/2$.

For instance in the n-queens problem, no two queens should be placed on the same row. Hence, for each $i \neq j$, the following conjunction should be spawned:

\[
\text{not_eq}(\text{On}_1Q_i, \text{On}_1Q_j), \text{not_eq}(\text{On}_2Q_i, \text{On}_2Q_j), ..., \text{not_eq}(\text{On}_nQ_i, \text{On}_nQ_j)
\]

where $[\text{On}_1Q_i, ..., \text{On}_nQ_i]$ and $[\text{On}_1Q_j, ..., \text{On}_nQ_j]$ are the Ons vectors for columns $Q_i$ and $Q_j$ respectively.

Constant propagation takes place during the and-parallel phase of the execution. If
this phase terminates with a deadlock, a suspended cell goal is selected and a choice point is created for it; a new and-parallel phase is then initiated for each search branch, exploring the possible computations for the program.

The number of backtrackings exhibited by the n-queens program using our decision variable representation for each column in the board is given in Table 4.1 under the heading "constraint-based(column)", compared with the results given in the previous chapter

4.5 Generalization of the Algorithm

In the above programming technique, the cell goals representing a decision variable are sharing a variable X and are connected in a chain with distinct constants at its ends, ensuring that exactly one value will be assigned to the variable. It is often more natural to allow the final value of a variable to be a set, rather than a scalar value. For instance, if the variable has several possible values which satisfy all the problem's constraints, then the final solution may assign a set of these values to the variable rather than selecting one of them. Moreover, some applications are more naturally expressed by a multi-dimensional matrix than by linear domain variables. An example for such need will be given in the following section. One advantage of the Pandora approach is that it is not restricted to linear domain variables. The algorithm can be generalized by introducing the set variable which is assigned a set of values, called the assignment set. An implementation for the set variable, which can be assigned one or more values, would be:

\[
\text{greater}_{-eq1}(\text{On}_1, \text{V}_1, \text{X}_2, \text{begin}, \text{R}_2), \\
\text{greater}_{-eq1}(\text{On}_2, \text{V}_2, \text{X}_2, \text{X}_3, \text{R}_2, \text{R}_3), ..., \\
\text{greater}_{-eq1}(\text{On}_n, \text{V}_n, \text{X}_n, [], \text{R}_n, \text{end})
\]

where \text{greater}_{-eq1}/6 is defined in Program 4.3. Instead of the third argument of a cell goal which represents a decision variable, the third and fourth arguments of the \text{greater}_{-eq1} goals represent the assignment set by a difference list to which the

\[3\]These results are for finding one solution to the problem and assume that, when the computation deadlocks, a queen is assigned to the smallest possible row in the smallest column available. In Chapter 6, it will be shown how to achieve such an order of choices.
value $V_i$ is added by the $i$th $\text{greater_eq1}$ goal if it is reduced using its first clause. The $i$th $\text{greater_eq1}$ goal becomes deterministic in the following cases:

(a) $On_i$ is bound.
(b) the first argument of all the other $\text{greater_eq1}$ goals is bound to "off" because then they will succeed and unify their third and fourth argument variables, as well as their fifth and sixth arguments. This makes the $i$th goal $\text{greater_eq1}(On_i, V_i, X, [], \text{begin}, \text{end})$ which eliminates its second clause, unifies $On_i$ with "on", and unifies $X$ with "[Vj]".
(c) the third argument is unified to a non-variable term that is not a list. This eliminates the first clause.
(d) the third and fourth arguments $X_i$ and $X_{i+1}$ are either bound to the same
term, which eliminates the first clause, or bound to two non-unifiable terms, which eliminates the second clause.

Cases (a) and (b) above are respectively equivalent to cases (a) and (c) in the previous section, in which the $i$th $\text{cell}$ goal becomes deterministic. However, unlike the conjunction of $\text{cell}$ goals, more than one $\text{greater_eq1}$ goal may succeed binding its $On$ variable to "on".

$$\text{greater_eq1}(on, V, [V|Tail], Tail, _, _).$$
$$\text{greater_eq1}(off, _, List, List, Chain, Chain).$$

**Program 4.3.** The don't-know relation $\text{greater_eq1/6}$.

Instead of using a conjunction of $\text{cell}$ goals, a decision variable can be considered to be a set variable which must have exactly one value in its assignment set and may therefore be implemented by a conjunction of two simpler properties:

$$\text{vec_ge1}(Ons, Vs, [X], []).$$
$$\text{vec_le1}(Ons, Vs, X)$$

Given $Vs= [V_1, ..., V_n]$: the possible values for $X$, and $Ons= [On_1, ..., On_n]$: the vector representing $X$'s set of possible values, $\text{vec_ge1/4}$ is a recursive program that spawns the conjunction of $n \text{greater_eq1}$ goals as given above, ensuring that $[X]$ is assigned at least one value from its set of possible values. A goal to $\text{vec_le1/3}$ spawns the conjunction:

$$\text{less_eq1}(On_1, V_1, X), \text{less_eq1}(On_2, V_2, X), ..., \text{less_eq1}(On_n, V_n, X)$$

which ensures that at most one value is assigned to $X$. $\text{less_eq1/3}$ is defined in Program 4.4. The $i$th goal becomes deterministic if $On_i$ is bound. It also becomes deterministic if $X$ is bound to $V_j$ such that $V_j \neq V_i$. $\text{vec_le1}$ may terminate
successfully, binding each of On1, ..., Onn to "off" and leaving X unbound. In such a case, vec_ge1 will fail since one of the greater_eq1 goals will become greater_eq1(off, V, [X], [], begin, end) and will thus fail.

    less_eq1(on, V, V).
    less_eq1(off, _, _).

Program 4.4. The don't-know relation less_eq1/3.

4.6 The N-Queens Problem -- Revisited

The n-queens problem has been re-defined using our representation of overlapping set variables. An n*n board can be represented by an n*n array of bits Onij, one for each square in row i and column j. Onij is bound to "on" if the square has a queen, and "off" otherwise.

Each column j is represented by the following conjunction, where Qj is the number of the row in which a queen is placed in column j:

    vec_ge1(Onj, Vs, [Qj], [i]),
    vec_le1(Onj, Vs, Qj) where Onj = [On1j, ..., Onnj]
    Vs = [1, ..., n]

A similar representation is used for each row. Additionally, all the squares in each diagonal are connected to a vector satisfying the vec_le1 property to ensure that each diagonal has at most one queen on it. A pictorial representation of the constraints on square(i, i) is shown in Figure 4.1. It illustrates how each square (and therefore its Onij) is involved in a representation of four overlapping set variables.

![Figure 4.1. Constraints on square(i, i) in the n-queens problem.](image-url)
The above constitutes a dynamic representation of the board. At any time during the computation, the goals in the resolvent represent the squares that are available for placing queens on the board. If Onij is bound to "off", square(i, j) is no longer a possible location for a queen. It is thus removed from the board and so are the goals representing it in the computation.

The top-level query to the program is a goal n_queens(N, Queens) whose relation is defined in Program 4.5. It accepts N: the size of the board, and spawns three concurrently executing processes: generate_columns/3 which spawns the conjunction of goals representing the columns in the board, generate_rows/2 which spawns the conjunction of goals representing the rows, and generate_diagonals/1 which spawns a vec_le1/3 goal for each diagonal.

mode n_queens(?N, ^).  
n_queens(N, Queens) <-  
gr_ate_columns(N, Ons, Queens),  
gr_ate_rows(N, Ons),  
gr_ate_diagonals(Ons).

Program 4.5. The don't-care relation n_queens/2.

The following properties are realized when running the program:

(A) A Lazy Non-Deterministic Choice
The computation deadlocks when at least one row or column on the board is not yet assigned a queen. Moreover, for each row and column that has not been assigned a queen, there are several possible positions to place one.

For example, after spawning all the goals that are representing the set variables, the computation deadlocks waiting for a queen to be placed on the board. Then, a suspended goal for a don't-know relation (less_eq1(Onij, V, X), or greater_eq1(Onij, V, Xi, Xi+1, Left, Right)) is selected to create a choice point. There are two alternative branches for the choice point: one branch binds Onij to "on", placing a queen on square(i, j), and the other branch binds Onij to "off", removing the square from the board.

(B) Eager Propagation of Constraints
As soon as a queen is (deterministically/non-deterministically) placed on a
square(i, j), all other squares in the same row (i), column (j) and diagonals will be removed from the board by binding their On variables to "off"; these squares are no longer possible locations for placing a queen. In Figure 4.2, for example, a queen is placed on square(1,1) and consequently all the squares in row 1, column 1, and the main diagonal are removed from the board.

![Image 4.2](image)

Fig 4.2. the 4*4 board after placing a queen on square(1,1).

(C) **Eager Detection of Failure**

An attempt to remove all the squares in a row or a column results in failure since one of the greater_eq1 goals becomes greater_eq1(\text{off, } _, _, _, \text{begin, end}) which fails. An infeasible placement of a queen may thus be detected before allocating other queens on the board. For instance, placing a queen on square(3,2) in Figure 4.2 above will result in failure since all the squares in column 3 should be removed. This indicates that a solution with queens on square(1,1) and square(3,2) is infeasible no matter where the other queens are placed.

(D) **Eager Detection of Determinacy**

If all the squares in a row/column except one (square(i,j)), are removed from the board, a greater_eq1 goal becomes greater_eq1(Onij, V, X, [], begin, end) which deterministically binds Onij to "on", placing a queen on that square. In figure 4.3, for example, square(4,2) is the only available square in column 2 and thus a queen is deterministically placed on it. Consequently, square(1,3) will be the only square in column 3 and square(3,4) will be the only square in row 3; both will be deterministically assigned queens, resulting in the feasible assignment [2,4,1,3].

![Image 4.3](image)

Fig 4.3. The 4*4 board with a queen on square(2,1).
Saraswat (Saraswat, 1987b) used CP[↓, l, &, ;] to solve the n-queens problem using a network of processes with a topology that is quite similar to ours. However, because his language forks eagerly as explained in Chapter 2, delaying the non-deterministic assignment of a queen to a square had to be explicitly programmed. While this is possible, it made the program considerably more complex. Also, it is more difficult, in general, to construct programs in CP[↓, l, &, ;] from simple units as in our approach.

In Table 4.1, the column with the heading "constraint-based(row+column)" gives statistics on the number of backtrackings exhibited by the n-queens program described in this section until the first solution is found. Due to the overlapping of set variables, the program reduces the search space more than the earlier version that was described in Section 4.4. This is because, during search, either a column or a row may be left with all but one square removed; the program will deterministically assign a queen to this square. The program in Section 4.4 does this only for columns.

<table>
<thead>
<tr>
<th>Queens</th>
<th>Sequential</th>
<th>Pandora</th>
<th>constraint-based(column)</th>
<th>constraint-based(row+column)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>10</td>
<td>7</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>186</td>
<td>46</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>2842</td>
<td>223</td>
<td>24</td>
<td>21</td>
</tr>
<tr>
<td>10</td>
<td>58986</td>
<td>276</td>
<td>24</td>
<td>21</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>873</td>
<td>54</td>
<td>37</td>
</tr>
<tr>
<td>14</td>
<td>-</td>
<td>7262</td>
<td>349</td>
<td>258</td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>-</td>
<td>1833</td>
<td>1271</td>
</tr>
</tbody>
</table>

Table 4.1. The number of backtrackings until finding the first solution to the n-queens problem.

### 4.7 Generating Naval Flying Programmes

An example of a real-life resource allocation problem is the generation of a naval flying programme which comprises the allocation of airborne resources (such as aircraft) to naval activities (such as flights) whilst taking into account a number of constraints (Cunningham, 1988). The inputs to the problem are two sets: a set of flights and a set of available aircraft, and to each flight a decision variable is defined
4. Constraint-Based Reasoning in Pandora

whose value in the resulting programme is the aircraft assigned to that flight.

Each flight as well as aircraft has several attributes. A flight has an identification number to differentiate it from other flights, and has fixed launch and landing times. The type of aircraft that can be assigned to the flight, and the minimum number of members that is required in the crew (number of crew), are also specified.

An aircraft's attributes are its identification number, its type, its endurance: that is the maximum time the aircraft can fly continuously without being maintained, the turnaround time: that is the time required for the aircraft to be maintained between two consecutive flights, the maximum number of flying hours which the aircraft can fly in one day, and a list of its crew members with their total number (number of crew).

There are several requirements or constraints to be considered when assigning an aircraft to a flight. These are categorized below according to the number of flights involved in each constraint, i.e. according to the number of decision variables related in each constraint.

(A) Unary Constraints
A unary constraint is imposed on the aircraft that can be assigned to a certain flight, irrespective of the aircraft assigned to other flights. Typical unary constraints are:

(a) the aircraft has to be of the type required by the flight.
(b) the aircraft's endurance time should be equal to or longer than the flight's length.
(c) the number of crew on the aircraft should be sufficient for the flight.

(B) Binary Constraints
These are constraints relating the decision variables of two flights, such as:

(a) flights that are overlapping in time cannot be assigned the same aircraft nor different aircraft with a common crew member.
(b) an aircraft can be assigned to two non-overlapping flights provided there is sufficient time between them for the aircraft to be refuelled and maintained.

(C) Multiple Constraints
Some constraints affect the values assigned to several flights' decision variables, such as:
(a) the same aircraft can be assigned to several flights as long as it does not exceed its maximum flying hours in one day.
(b) a crew member can fly several flights provided he/she does not exceed the maximum time a crew member can fly in one day.

Fly-Pan: a Pandora program for automatically generating naval flying programmes, is developed using the technique previously described in Section 4.5. An early version of the program was described in (Al-Abed, 1989). Since then, the structure of the program has been changed to mirror the different categories of constraint in the problem. The latest version of the program is described below. Additionally, domain-dependent heuristics have been added to increase the computation's efficiency; these will be described in Chapter 6.

4.7.1 Fly-Pan

Fly-Pan accepts a list of flights and a list of aircraft. Each flight in the input list is a structure of its attributes of the form: flight(Id, Launch, Landing, Type, Crew). Similarly, each input aircraft is of the form: aircraft(Id, Type, Endurance, Turnaround, Max_highs, Crew_no, Crew_list). An example of a valid input to the program which will be used later to describe the program's behaviour is as follows:

Input_flights = [ flight(1,2,10,bo707,3),
                 flight(2,4,9,bo707,3),
                 flight(3,11,15,bo707,5),
                 flight(4,16,19,helicopter,2)]

Input_aircraft = [ aircraft(1,bo707,12,2,12,5,[c1,c2,c3,c4,c5]),
                  aircraft(2,bo707,10,1,15,3,[c1,c6,c7]),
                  aircraft(3,bo707,7,1,8,5,[c8,c9,c10,c11,c12]),
                  aircraft(4,helicopter,3,1,10,2,[c4,c9])]

Fly-Pan assigns an aircraft to each flight such that all the constraints in the problem are satisfied. The output is a list of tuples, each of the form (Flight_Id, Aircraft_Id) representing the resulting assignment.

The top-level query to the program is a goal whose relation flying_pan/3 is defined in Program 4.6. produce_assignment/3 produces a candidate solution for the problem. After the successful evaluation of produce_assignment/3, the
extract_programme/2 goal is evaluated which accepts the produced solution and extracts from it the final form of the programme.

mode flying_pan(? , ?, ^).
flying_pan(Input_flights, Input_aircraft, Flying_programme) <-
    produce_assignment(Input_flights, Input_aircraft, By_flight)&
    extract_programme(By_flight, Flying_programme).

Program 4.6. The don't-care relation flying_pan/3.

produce_assignment/3 comprises five main concurrent, communicating processes as shown in Figure 4.4, in which three of them correspond to the three categories of constraint in the problem.

Fig 4.4. The structure of the produce_assignment/3 relation.

unary_constraints/3 reads in the two lists of input and, for each input flight, it spawns a unary_for_flight/3 goal to check the list of aircraft and produce Poss_aircraft: a list of aircraft satisfying the flight's unary constraints. This list constitutes the initial set of possible values for that flight. For instance, Figure 4.5 is a pictorial illustration of the initial sets of possible aircraft for the flights in the input that was given above. Since aircraft 4 is a helicopter, it cannot be assigned to flights 1, 2, and 3 which require aircraft of type bo707. Similarly, aircraft 1, 2, and 3, whose type is bo707 cannot be assigned to flight 4 which requires a helicopter. Moreover, aircraft 3 cannot be assigned to flight 1 since its endurance is
seven hours while the flight's length is eight hours, and aircraft 2 cannot be assigned to flight 3 because it does not have sufficient crew members for the flight.

It may be worth noting that the initial sets of possible aircraft for all the input flights are generated concurrently and, moreover, several constraints may simultaneously remove the same aircraft from a flight's set of possible aircraft. For example, aircraft 4 does not have sufficient crew members for flight 3 as well as not being of the required type.

Additionally, unary_constraints/3 spawns generate_aircraft/3 which in turn spawns a conjunction of goals for the don't-know relation cell/5 to implement the decision variable for each input flight. For instance, flight 1 in the considered example is represented by:

\[
\text{cell(On1, 1, F1, begin, CF1), cell(On2, 2, F1, CF1, end)}
\]

while flight 4 is represented by one goal, cell(On, 4, F4, begin, end), which succeeds after deterministically assigning aircraft 4 to F4.

The output of unary_constraints/3 is a list of tuples, each of the form poss_aircraft(Flight, Aircraft_structure) where Aircraft_structure is a vector of the possible aircraft for Flight with their corresponding On variables in the cell goals.

\[
\text{mode unary_constraints(? , ?, ?).}
\]

\[
\text{unary_constraints([], _, []).} \quad \% \text{cl1.}
\]

\[
\text{unary_constraints([F1| Input_flights], Input_aircraft,}
\]

\[
\\text{[poss_aircraft(F1, Aircraft_structure)| By_flight]) <-}
\]

\[
\text{unary_for_flight(F1, Input_aircraft, Poss_aircraft),}
\]

\[
\text{generate_aircraft(Poss_aircraft, F1, Aircraft_structure),}
\]

\[
\text{unary_constraints(Input_flights, Input_aircraft, By_flight).} \quad \% \text{cl2.}
\]

**Program 4.7.** The don't-care relation unary_constraints/3.

binary_constraints/2 accepts the list produced by unary_constraints/3 and for each pair of flights it spawns the goal binary_constrs/3 to check the existence of any binary constraints between these flights and to spawn the relevant goals.

binary_constrs/3 is given in Program 4.8. If the two flights are overlapping in
time (clauses 1 and 2), then they cannot be assigned the same aircraft or two aircraft that are sharing a common crew member.  noteq_aircraft/2 spawns the required not_eq/2 goals to prevent assigning the same aircraft to both flights (the not_eq/2 relation was previously defined in Program 4.2).  noteq_crew/3 spawns not_eq/2 goals between any two aircraft sharing a common crew member. Otherwise, i.e. if the flights are not overlapping in time), turnaround_constraint/4 in clause 3 spawns not_eq/2 goal(s) to prevent assigning an aircraft to both flights if its turnaround time is longer than the time between the two flights.

```
mode binary_constrs(? , ?, ?).
binary_constrs( poss_aircraft(F1,A1), poss_aircraft(F2,A2), By_crew)<-
F1 =.. [flight, _, Launch1, Landing1|As1],
F2 =.. [flight, _, Launch2|As2],
Launch2 ≥ Launch1,
Launch2 ≤ Landing1: % overlapping
noteq_aircraft(A1, A2),
noteq_crew(By_crew, A1, A2).

binary_constrs( poss_aircraft(F1,A1), poss_aircraft(F2,A2), By_crew)<-
F1 =.. [flight, _, Launch1, Landing1|As1],
F2 =.. [flight, _, Launch2, Landing2|As2],
Launch1 ≥ Launch2,
Launch1 ≤ Landing2: % overlapping
noteq_aircraft(A1, A2),
noteq_crew(By_crew, A1, A2); % cl2.

binary_constrs( poss_aircraft(F1, A1), poss_aircraft(F2, A2), _ )<-
turnaround_constraint(F1, A1, F2, A2). % cl3.
```

**Program 4.8.** The don't-care relation binary_constrs/3.

The binary constraints are shown in Figure 4.5 by crossed continuous lines connecting the relevant aircraft. Each line is denoting a not_eq/2 goal between the On variables of the connected aircraft. For instance, since flights 1 and 2 are overlapping in time, they can neither be assigned the same aircraft (namely aircraft 1 or 2) nor aircraft that are sharing crew member(s) such as aircraft 1 and 2 which are sharing cl. Although flights 1 and 3 are not overlapping in time, the duration from flight 1's landing to flight 3's launching is one hour which is less than the
turnaround time for aircraft 1. As a result, aircraft 1 cannot be assigned to both flights.

\textbf{By\_crew}, in Figure 4.4, is a list of elements each representing a distinct crew member in the problem. A crew member is itself represented by a list of the aircraft in which he/she is a member. For example the \textbf{By\_crew} list for our sample input is:

\texttt{By\_crew = [ [1, 2], [1], [1], [1, 4], ..., [3]]}

where the first element represents the crew member 'c1', the second represents 'c2', and so on until the representation of 'c12'. As shown in Figure 4.4, \textbf{By\_crew} is produced by the goal \texttt{arrange\_by\_crew/2} and used by the goals \texttt{binary\_constraints} and \texttt{multiple\_constraints}.

\textbf{Figure 4.5.} The initial set of possible aircraft for the input flights.

There are two multiple constraints in the problem: (a) an aircraft can be assigned to several flights as long as it does not exceed its maximum flying hours in one day, (b) a crew member can fly several flights provided he/she does not exceed the maximum time a crew member can fly in one day. The former can be considered as a unary constraint on each aircraft in the problem putting an upper limit on the number of hours it can fly in one day, and thus indirectly affecting the flights it can be assigned to. Similarly, the latter can be considered as a unary constraint on the flying hours for each crew member in the problem on the same day.
multiple_constraints/3 (in Program 4.9) accepts three input arguments: a number denoting the maximum flying hours for a crew member in one day which is 15 hours in our example, a list representing the distinct aircraft in the problem which is produced by arrange_by_aircraft/2, and the list By_crew which represents the distinct crew members in the problem. The maximum flying hours for an aircraft is one of its attributes and is different for different aircraft.

A goal to multiple_constraints/3 dynamically modifies a counter for each aircraft and each crew member in the problem. The counter, at any time, represents the remaining number of hours the corresponding aircraft/crew member can fly on that day; it is initialized with the maximum hours it (he/she) can fly in one day. When a flight is assigned an aircraft, its length is subtracted from the counter of that aircraft as well as the counters of all the crew members on the aircraft. For instance, when assigning aircraft 4 to flight 4 in our example, the remaining flying hours for aircraft 4 is reduced to 7, while it is reduced to 12 hours for c4 and c9.

When a length of a flight is longer than the current value of an aircraft’s counter, the aircraft is immediately removed from the set of possible aircraft for the flight. Similarly, if a crew member's remaining flying hours are less than a flight's length, all aircraft having this person as a crew member are no longer possible values for that flight.

```
mode multiple_constraints(? , ? , ?).

multiple_constraints(By_aircraft, By_crew, Max_crew_hrs) :-
    maxhrs_aircraft(By_aircraft),
    maxhrs_crew(By_crew, By_aircraft, Max_crew_hrs).
```

Program 4.9. The don't-care relation multiple_constraints/3.

By_aircraft in Program 4.9, is produced by arrange_by_aircraft/2. It is a list of tuples, each of the form aircraft(Aircraft, Flight_structure) to represent an aircraft in the problem. Flight_structure is a vector of the flights for which Aircraft is a possible assignment together with their corresponding On variables.

4.7.2 Making Choices

Applying constraint-based techniques, together with the lazy don't-know non-
4. Constraint-Based Reasoning in Pandora

determinism in Pandora, reduce the search space dramatically. However, since the problem under consideration is computationally complex, it is most likely that the and-parallel phase of the computation will not be sufficient to assign an aircraft to each input flight, and choices have to be made for the values of some decision variables. Allocating an aircraft to a flight non-deterministically may not satisfy all the constraints in the problem and the computation will thus fail in this branch of the choice point. Alternatively, the allocation may contribute to a candidate solution for the problem.

Suppose, for example, that aircraft 1 is non-deterministically assigned to flight 1. This removes aircraft 2 from flight 1's set of possible aircraft and, due to the binary constraints, aircraft 1 is removed from the sets of flights 2 and 3, as well as aircraft 2 from flight 2's set. Moreover, the flying hours for crew members c1, c2, c3, and c5 are reduced to 7 hours while, in case of aircraft 1 and crew member c4, reduced to 4 hours. Since the length of flight 2 is 5 hours, the insufficient remaining flying hours for aircraft 1 and crew member c4 is another reason for removing the aircraft from flight 2's set.

Aircraft 3 is now the only possible aircraft for both flights 2 and 3 and an attempt to assign it to both flights results in failure since the maximum flying hours constraint is no longer satisfied. Hence, the computation backtracks and aircraft 2 is assigned to flight 1, removing aircraft 1 and 2 from flight 2's set. Aircraft 3 is, then, assigned deterministically to flight 2 since it is its only possible aircraft. This removes aircraft 3 from flight 3's set because of the maximum flying hours constraint, and flight 3 is finally assigned aircraft 1.

4.8 Related Work

The most widely used methods for solving resource allocation problems are mathematical methods which formulate the problem as a mathematical model using mathematical symbols and expressions (equations and inequations). Then, a standard mathematical algorithm is applied on the model to solve the problem. These methods are usually efficient and require no programming from the user. However, recasting the problem into a mathematical model often makes it impossible to exploit the problem features. It may also increase substantially the number of constraints and variables in order to represent the real problem accurately.
4. Constraint-Based Reasoning in Pandora

(van Hentenryck, 1989); otherwise approximation and simplifying assumptions are required if the model is to be tractable (i.e. capable of being solved). Additionally, these methods are all algorithmic and there is no way to use heuristics that are specific to the problem.

An ATMS system, Fly-Past, for automatically generating naval flying programmes is described in (Cunningham, 1988). It provides a formal language for formulating constraints, but no explicit programming is required from the user. The system combines techniques for assumption-based reasoning and constraint-based reasoning to efficiently solve the task. Information on logical dependencies is stored (and then fetched) to record the derivations and the assumptions under which they hold. Based on this information, the computation can backtrack to the relevant choice. Moreover, when circumstances change, only the affected parts of the programme are re-planned while retaining the unaffected parts. The drawbacks of such a system are: (a) the system has a fixed language for formulating the constraints. This can entail a kind of recasting as in mathematical models; (b) it is difficult to evaluate the memory required to store the logical dependencies during the computation; (c) the system is a black box for its users. Therefore, users cannot specify particular heuristics for solving the problem and it is unlikely that the system can find a good strategy to solve all problems; (d) in the described problem, an aircraft has a maximum flying hours in one day which it cannot exceed, and similarly for a crew member. This constraint requires a dynamic model of the state of each aircraft/crew member during the computation which could not be represented in the simple constraint network of Fly-Past.

Several constraint logic programming languages have been recently developed such as CLP(R) (Jaffar and Lassez, 1987; Jaffar and Michaylov, 1987), Prolog III (Colmerauer, 1987), and CHIP (van Hentenryck, 1989). CLP (Constraint Logic Programming) is a framework for constraint handling in logic programming. It defines a class of programming languages sharing the same essential semantics. The CLP languages as well as Prolog III replace unification in the logic programming language scheme with constraint solvers in specific computation domains. They, thus, generalize the logic programming paradigm with constraint solving operations in which unification is one type. Each language has a pre-defined set of built-in constraints and, given any combination of these constraints, the constraint solver can decide their satisfiability. The constraint solver is a black box for its users. This liberates the users from its internal workings but also prevents them from guiding or influencing it.
CHIP is a special-purpose logic programming language which, among other things, extends Prolog to handle finite-domain constraint solving. The extensions are of two kinds: a low-level representation of a decision variable (termed a domain variable in the CHIP papers), and a special-purpose coroutining mechanism which ensures that constraints are executed at the appropriate time to reduce the set of possible values as soon as possible.

Having a low-level support liberates the user from explicitly programming the active use of constraints. It should also handle constraints more efficiently than creating high-level processes as in Pandora. However, Gregory has experimented with running such Pandora programs on Andorra-I (a prototype implementation of the Andorra model)\textsuperscript{4}. The speed obtained using a single processor was comparable with the published results for CHIP (personal discussion). These results are not very meaningful since both systems are running on different machines and hence we did not include them in the thesis. However, they show that the Pandora technique could be practical. Moreover, CHIP does not support a stream and-parallel evaluation of communicating processes as in Pandora since it executes the goals in a left to right order; only constraints are actively executed by the underlying implementation. As more parallelism is exploited by increasing the number of processors involved in a Pandora computation, the results of running the Pandora programs improves.

An advantage of our approach is the overlapping of set variables. For example, a CHIP program for the n-queens problem was presented in (van Hentenryck, 1989), in which a square can only be a member of one decision variable, namely a column. In our last program for the n-queens problem (see Section 4.6), a square is a member of four set variables: a column, a row, a left diagonal, and a right diagonal. As a result, the CHIP program only reduces the search space similar to our earlier program in Section 4.4.

\textsuperscript{4} after manually translating the programs to the language supported by Andorra-I.
Chapter 5

Pandora for Distributed Discrete Event Simulation

In distributed discrete event simulation, a real system is simulated by a number of logical processes, which communicate via shared channels. Stream and-parallelism provides a useful tool for programming such simulation models. For instance in Pandora, logical processes can be implemented by concurrent goals for don't-care relations which communicate incrementally by exchanging time-stamped messages through shared variables.

The major shortcoming of a distributed simulation is the possibility of deadlock, and several deadlock detection and recovery algorithms have been previously proposed. These algorithms were adopted in simulation programs previously written in concurrent logic programming languages, such as the ones described in (Davison, 1988). In Pandora, deadlock is detected by the underlying computational model and hence does not require additional machinery. Instead of sophisticated deadlock recovery algorithms, a simple programming technique using the Pandora don't-know relations is described in this chapter. This technique cannot be adopted in committed-choice languages as it relies on the lazy search capability of Pandora.

The following section introduces the simulation problem as well as several simulation methods, the last of which is the distributed discrete event simulation (DDES). Section 5.2 illustrates how to implement the DDES method using a concurrent logic programming language such as Pandora, followed by a Pandora program for simulating a primitive computer system in Section 5.3. The deadlock problem together with some well-known deadlock detection and recovery algorithms are defined in Section 5.4. These algorithms were adopted in several
DDES models, such as the ones mentioned in Section 5.6. A simple programming technique for deadlock recovery in a Pandora simulation program is explained in Section 5.5 and compared with other DDES models in Section 5.7.

5.1 The Simulation Problem

The simulation problem is the problem of constructing a logical model that can be executed on a computer, to simulate the behaviour of a real system. Examples of real systems that can be simulated include operating systems, computer networks, and many kinds of servicing environments such as a supermarket or a car wash.

A real system consists of one or more physical processes. Each physical process operates autonomously except to interact with other physical processes in the system. The interaction is by messages. Contents of the message sent by a process depend on the characteristics of the process (its initial state, its activities) and the messages that the process has received so far. A logical model should be able to simulate the correct sequence of message transmissions in the real system.

Deciding on which events to simulate depends on the purpose of the simulation. For instance, if the intention is to assess the overall system's behaviour, then the details of local activities of the physical processes can be ignored; it is the interaction of processes which is important. Typical roles of the simulation are:

a) to generate reports.
b) to collect statistics.
c) to predict performance or future behaviour.

Several simulation methods were developed. These methods can be divided into two types: continuous and discrete, depending on whether the simulated time is changing continuously or in discrete jumps. In discrete simulation methods, all the events separated by a negligible time difference are considered to occur at the same time.

One kind of discrete simulation is called discrete event simulation (Fishman, 1978), in which the change in the simulated time is event-driven. That is, the simulated time is advanced after simulating all the events that are scheduled at the current time, to the time of the next set of events.
There are three main approaches to discrete event simulation, ordered below by increasing degree of parallelism:

1. **Sequential Discrete Event Simulation**
   It is also called *coroutining* and/or *continuation simulation techniques* as in Simula (Birtwistle, 1979), Smalltalk and Scheme. A variable *clock* holds the time up to which the real system has been simulated. A central data structure, called the *event list*, maintains a set of messages with their associated times of transmission, that are scheduled for the current or a future time. At each step, the message with the smallest associated time is removed from the event list, and its transmission in the real system is simulated. This may, in turn, cause other messages to be sent in the future (which then are added to the event list) or cause previously scheduled messages to be cancelled (which are removed from the event list). The clock is advanced to the time of the message transmission that was just simulated. This method is inherently sequential since in each cycle of simulation only one item from the event list is processed and the event list is possibly updated.

2. **Process Interaction Discrete Event Simulation**
   In this method, the logical model is decomposed into a number of *concurrent logical processes*, each representing activities or objects in the real system. Logical processes communicate via messages, and events occur at these times when processes interact.

   The simulated time is represented by a *central clock process*, which controls the times at which activities are simulated. A process simulating an activity at time $T_i$ would send an alarm request to the clock process and receive an alarm signal in reply when the simulated time is $T_j$. It can then start its activity. Events that are scheduled at the same time can be processed simultaneously by the concurrent logical processes. After executing all the events at $T_i$, the logical processes suspend and the clock process advances the simulated time to $T_{i+1}$: the time of the next simultaneous events.

   The degree of parallelism in this method depends on the amount of activity and the number of events that are scheduled at each simulated time instance.

3. **Distributed Discrete Event Simulation**
   Distributed discrete event simulation (DDES) was proposed to allow events that are
5. Pandora for Distributed Discrete Event Simulation

scheduled at different time instances to be processed simultaneously in the logical model.

Similar to the previous method, a DDES simulation model comprises a number of concurrent logical processes that communicate via messages. If a logical process LPi is sending a message to a process LPj, then there is a direct channel from LPi to LPj. The model may be viewed as a network of communicating processes.

The contents of a message sent by a process depends on the characteristics of the process as well as the messages that the process has received so far. Hence, logical processes in the simulation should receive and manipulate messages in the correct sequence as received by their physical counterparts. There is no central clock process in a DDES model. Instead, the synchronous nature of the real system is captured by encoding time as part of each message transmitted. A message "Msg" sent by a process at time "T" will be of the form: "Msg@T", and a sequence of messages [M1@T1, M2@T2, ..., Mn@Tn] sent in a channel from one process to another should be ordered, i.e. T1 ≤ T2 ≤ ... ≤ Tn. Additionally, if a process has several input channels, then it should receive the messages from all its input channels in ascending order according to their time stamps. For instance, if the processes Pi and Pj respectively send the messages [M1i@1, M2i@4, ...] and [M1j@2, M2j@3, ...] to the process Pk, then Pk should receive and manipulate its input messages in the following order:

[M1i@1, M1j@2, M2j@3, M2i@4, ...]

Suppose that the most recently received message by a process has a time component T_i. Then, T_i is said to be the current local simulated time for the process, which may be different for different processes. The global simulated time of the logical model, that is the time up to which the physical system is correctly simulated, is T = minimum{T_i}, where T_i's are the local simulated times of all the logical processes in the model.

5.2 Distributed Discrete Event Simulation in Pandora

A logical process in DDES can be implemented in Pandora by a goal for a don't-care relation. A channel from a process LPi to a process LPj can be represented by a shared variable between the goals representing LPi and LPj; it is an output
argument of LPi's goal and an input argument of LPj's goal. A goal suspends until it receives message(s) in its input argument(s), selects the message with the least time component, performs the required activities, and sends message(s) in its output argument(s).

One way of ensuring that a process receives messages from all its input arguments in the correct order is to time-merge all its input channels into one channel that is received by the process. For example if the process has two input channels X and Y, then a goal time_binary_merge(X, Y, Input) for the don't-care relation in Program 5.1 can be invoked to merge X and Y into one input channel: Input. A time_binary_merge goal suspends until either one of its input arguments is instantiated to an empty list (clauses 1 and 2), or both arguments are instantiated to non-empty lists and the first message in each list has a ground time component (clauses 3 and 4). The goal outputs a list of messages from both its input arguments, increasingly ordered by their time stamps.

```
mode time_binary_merge(? , ?, ^).
time_binary_merge([], Y, Y). % cl1.
time_binary_merge(X, [], X). % cl2.
time_binary_merge([MX@TX|Xs], [MY@TY|Ys], [MX@TX|Z]) <-
  TX ≤ TY:
  time_binary_merge(Xs, [MY@TY|Ys], Z). % cl3.
time_binary_merge([MX@TX|Xs], [MY@TY|Ys], [MY@TY|Z]) <-
  TX > TY:
  time_binary_merge([MX@TX|Xs], Ys, Z). % cl4.
```

**Program 5.1.** The don't-care relation time_binary_merge/3.

In general, processes may have various number of input channels. In order to allow the channels to be dynamically created during the computation, i.e. to allow a dynamic configuration of the model, we introduce the **split stream**: a data structure comprising an arbitrary number of channels. A split stream is either:

(a) "[]": the empty list,

(b) "[M1, ..., Mn|SS]" n > 0, where each Mi is a message and SS is a split stream, or

(c) "split([SS1, ..., SSk])", where each SSi is a split stream.

(d) "split([])": an empty split stream.
For each logical process with multiple input channels, a goal for Program 5.2 is spawned which dynamically merges the input channels into one single channel that is received by that process. The first two clauses correspond to (a) and (b) in the definition of a split stream respectively. Clauses 3 and 4 manipulate a split stream of the form defined in (c) and (d) above.

```
mode time_merge(? , ^).  
time_merge([], []).  % cl1.
time_merge([M1|ICs], [M1|Channel]) <- time_merge(ICs, Channel).  % cl2.
time_merge(split([IC1|ICs]), Channel) <- time_merge(IC1, O1),
time_merge(split(ICs), O2),
time_binary_merge(O1, O2, Channel).  % cl3.
time_merge(split([]), []).  % cl4.
```

Program 5.2. Multi-way time_merge of several channels.

### 5.3 Simulating a Computer System

A primitive computer system, consisting of a central processing unit (CPU) and N peripheral processors, is to be simulated. Jobs arriving from different terminals are merged into a single queue to be processed by the CPU. Each job is allocated a time slot of 3 time units, which determines the maximum time the job can spend in the CPU at one time. If the job is not completed after the termination of the time slot, then it requeues waiting for the CPU to be free again in order to complete its processing. Upon completion, a job waits until served by an arbitrary free peripheral processor then leaves the system. A schematic diagram of the system is in Figure 5.1, while Figure 5.2 presents its simulated model. We assume that the arrival of jobs is uniformly distributed between nine and twelve o'clock, the processing time for a job is uniformly distributed between one and five time units, and the time spent at a peripheral processor is uniformly distributed between one and twenty time units. The time a job spends on waiting to be processed varies with: the load of jobs, the processing time of the preceding jobs, and the number (N) of peripheral processors available. The model computes the average waiting time for a job in the system, given the number of peripheral processors available and the number of incoming jobs.
A Pandora program to simulate the computer system described above has been developed. The program accepts the number of jobs arriving between nine and twelve o'clock as well as the number of peripheral processors, and computes the average waiting time for a job. The top-level query to the program is a goal for the relation `computer_simulation/3` in Program 5.3. It spawns goals for the logical processes illustrated in Figure 5.2.
mode computer_simulation(?, ?, ^).

computer_simulation(JobsNo, PeripheralsNo, AverageWait) <=
random(R), statistics(S1, S2, JobsNo, AverageWait),
R = split([R1, R2, R3]),
job_generator(JobsNo, Arrival, R1, R2),
queue1(Arrival, Incompleted_Jobs, CPU, S1),
cpu_processing(CPU, Incompleted_Jobs, Queue2),
queue2(Queue2, Processors, S2),
processor_generator(PeripheralsNo, Processors, O, R3),
outside(O, OutStream).

Program 5.3. The don’t-care relation computer_simulation/3.

random/1 is a utility program which implements a random number generator using a uniform distribution algorithm between lower and upper limits. These limits depend on the purpose of generating the number. For instance, the arrival time of a job is between nine and twelve o'clock, while the time for processing it on the CPU is between one and five time units.

statistics/4 is another utility program which collects data about the time each job waits before being processed on the CPU as well as its waiting time before being served by one of the peripheral processors. It then computes the average waiting time for a job in the system.

Jobs arrive randomly from the terminals connected to the real system. This is simulated by a split stream of N jobs that is dynamically generated by job_generator/4. Each job is of the form "job(No, Process_Time)@Arrival", where Process_Time is the CPU time required for the job to be completed and Arrival is the job's arrival time. The stream is accepted by queue1/4 which time-merges the jobs into a single stream, ArrivingStream, as illustrated in Program 5.4. Additionally, the goal receives Incompleted_Jobs: a stream of the jobs that have been previously processed on the CPU but did not complete. Incompleted_Jobs is time_binary_merged with ArrivingStream and the result is the queue of jobs waiting to be processed on the CPU, in the order of their arrival to the CPU. queue1/4 spawns an allocate_cpu/3 goal to allocate the jobs to the CPU. Clause 1 of the allocate_cpu/3 relation handles the case when a job arrives while the CPU is free and, thus, is immediately assigned to it. Clause 2 handles the case when a job has to wait until it is the first job on the queue and the CPU
becomes available.

mode queue1(? , ?, ?, ^).
queue1(Arrival, Incompleted_Jobs, CPU, S) <-
  time_merge(Arrival, ArrivingStream),
  time_binary_merge(ArrivingStream, Incompleted_Jobs, Queue1),
  allocate_cpu(Queue1, CPU, S).

mode allocate_cpu(? , ?, ^).
allocate_cpu( [job(No, Process_Time)@T1|Jobs], [Free@T2|CPU],
  [q1(No, 0)|S] ) <-
  T1 ≥ T2:
  Free = job(No, Process_Time)@T1,
  allocate_cpu(Jobs, CPU, S), % cl1.
allocate_cpu( [job(No, Process_Time)@T1|Jobs], [Free@T2|CPU],
  [q1(No, Wait)|S] ) <-
  T1 < T2:
  Free = job(No, Process_Time)@T2,
  Wait is T2 - T1,
  allocate_cpu(Jobs, CPU, S), % cl2.
allocate_cpu([], CPU, []) <-
  end_service(CPU). % cl3.

mode end_service(?).
end_service([]).
end_service([Free@T|CPU]) <-
  Free = end,
  end_service(CPU).

Program 5.4. The don't-care relations queue1/4, allocate_cpu/3, and end_service/1.

cpu_processing/3 simulates the processing of jobs on the CPU in the real system. The simulation begins at a simulated time = 0, at which the CPU will be available for the first arriving job to be processed. When a job is completed or when the time slot (i.e. 3 time units) is finished, whichever is first, the CPU becomes available for the next job on the queue. If the processed job is completed (clause 2 in Program 5.5), it queues until served by a peripheral processor. Otherwise, the job requeues for more CPU processing (clause 3 in Program 5.5).
5. Pandora for Distributed Discrete Event Simulation

mode cpu_processing(^, ^, ^).
cpu_processing([Free@0|CPU], Queue1, Queue2) :-
cpu_serving(Free, Queue1, Queue2, CPU). % cl1.

mode cpu_serving(?^, ^, ^, ^, ^).
cpu_serving( job(No, Process_Time)@T, Q1, [job(No)@Clock|Q2],
[Free@Clock|CPU]) <- Process_Time ≤ 3:
Clock is T + Process_Time,
cpu_serving(Free, Q1, Q2, CPU). % cl2.
cpu_serving( job(No, Process_Time)@T,
[job(No, Remaining_Time)@Clock|Q1], Q2,
[Free@Clock|CPU]) <- Process_Time > 3:
Remaining_Time is Process_Time - 3,
Clock is T + 3,
cpu_serving(Free, Q1, Q2, CPU). % cl3.
cpu_serving(end, [], [], []). % cl4.

Program 5.5. The don't-care relations cpu_processing/4 and cpu_serving/5.

queue2/3, in Program 5.6, represents the second queue in the model. Completed jobs, coming out of the CPU, wait until being served by the available peripheral processors. The second argument of queue2/3 is a split stream representing the free peripheral processors; these are generated by the goal for the processor_generator/4 relation in Program 5.7. allocate_peripheral/3 allocates the waiting jobs to the free peripherals, similar to allocating them to the CPU by allocate_cpu/3 in Program 5.4 above.

mode queue2(?^, ?, ^, ^).
queue2(Jobs, Procs, S) :-
time_merge(Procs, Free_Peripherals),
allocate_peripheral(Jobs, Free_Peripherals, S).

Program 5.6. The don't-care relation queue2/3.

processor_generator/4 accepts the number of peripheral processors in the system and invokes a proc_serving/4 goal for each peripheral processor, to simulate its behaviour in the real system. Finally, jobs leave the system in order of their time stamps.

- 109 -
mode processor_generator(?f, ^, ^, ^).
processor_generator(0, [], [], []). processor_generator(No, Procs, O, R) <- No > 0:
    Procs = split([Proc1, Proc2]),
    O = split([O1, O2]),
    R = split([R1, R2]),
    Proc1 = [Free@0|Available],
    proc_serving(Free, R1, O1, Available),
    N is No - 1,
    processor_generator(N, Proc2, O2, R2).

Program 5.7. The don't-care relation processor_generator/4.

5.4 The Deadlock Problem in DDES

In distributed simulation, the radical departure from sequential simulation is the lack of global control. Each logical process has its own local simulated time, and the system is synchronized by the chronological order of messages received and sent by each process. The major shortcoming of a distributed simulation is the possibility of deadlock. A distributed simulation deadlocks when all of the following conditions are satisfied:

1. every process is either waiting to receive a message or is terminated.
2. at least one process is waiting to receive a message.
3. for any process LP_i that is waiting to receive a message from some process LP_j, there is no message in transit from LP_j to LP_i.

The above conditions can be satisfied if channels are selected on a random basis to carry messages. Hence, a process may never receive messages on one of its input channels and will remain waiting. Deadlock also occurs when there is a cycle in the network of communicating processes. Consider, for example, the computer system that was simulated in the previous section. After merging the newly generated jobs into ArrivingStream (Program 5.4), a circular pattern of waiting processes is formed. This is illustrated in Figure 5.3. time_binary_merge(ArrivingStream, Incompleted_Jobs, Queue1) suspends since Incompleted_Jobs is uninstantiated. cpu_serving(Free, Incompleted_Jobs, Queue2, CPU) suspends since Free is not allocated a job yet by the allocate_cpu/3 goal, while allocate_cpu(Queue1, [Free@0|CPU], S) suspends since Queue1 is uninstantiated. Hence, the system
deadlocks.

Fig 5.3. A circular network of logical processes.

The system will also deadlock when all the jobs have been completely processed on the CPU and no more jobs arrive from the connected terminals (ArrivingStream = []). In this case, time_binary_merge will unify Incompleted_Jobs with Queue1, producing the circular pattern illustrated in Figure 5.4.

Fig 5.4. A circular network formed after all jobs are completely processed on the CPU.

Many approaches have been studied for either avoiding deadlock by sending null messages on rarely used channels or by breaking deadlock after detecting it by circulating markers. In the first algorithm, a null message, null@T_i, is sent on a channel to announce the absence of real messages on that channel up to time T_i. As a result, no process will block waiting for a message to arrive on that channel. The second algorithm allows deadlock to occur but detects it by sending a marker around. The marker visits all the processes in the network. If a process did not receive or sent a message since the last departure of the marker, it is said to be white. Otherwise, it is black. The marker declares deadlock if all the processes in the network remain white during a complete tour in the network. In order to
recover from deadlock, the marker may carry information about the process which is responsible for processing the next event. When deadlock is detected, that process is restarted.

Variations of the above algorithms are discussed and analysed in (Misra, 1986). The analysis reveals that none of these methods seems to work well in all cases but they are all generally suitable for problems in which all or most of the channels have regular message traffic.

The "null messages" and "circulating markers" algorithms have been used in concurrent logic simulation programs, such as the ones described in (Davison, 1988). In Pandora, deadlock is detected by the underlying computational model and does not require additional machinery. In order to recover from deadlock, a simple programming technique using the Pandora don't-know relation is presented below.

5.5 Deadlock Recovery in a DDES Pandora Program

The default manner for breaking deadlock in a Pandora program is by a non-deterministic fork of the computation. A DDES Pandora program comprises a number of processes connected by a network of time_binary_merge goals. Suppose that time_binary_merge/3 is defined by a don't-know relation, as in Program 5.8 below. The computation progresses as long as messages with known time stamps arrive on both input arguments of the goals (clauses 1 and 2). It also progresses if both input arguments are empty lists (clause 3), or one of them is an empty list while the other has at least one message (clauses 4 and 5).

When the computation deadlocks, a choice point may be created for a time_binary_merge/3 goal, sending the message MX@TX in its output stream (clause 1). If later in the computation messages arrive such that TX > TY, the computation backtracks and sends the message MY@TY instead (clause 2). If one of the goal's arguments is bound to an empty list, the goal will non-deterministically unify its other input argument as well as its output argument with empty lists (clause 3). The goal backtracks if later in the computation messages

5assuming that don't-know non-determinism is implemented by backtracking.
arrive in its other input argument and is reduced using clauses 4 or 5, depending on
the input argument in which the messages arrive.

\[
\text{time\_binary\_merge}([\text{MX}\@\text{TX}|\text{Xs}], [\text{MY}\@\text{TY}|\text{Ys}], [\text{MX}\@\text{TX}|\text{Z}]) :-
\]
\[
\quad \text{TX} \leq \text{TY}:
\quad \text{time\_binary\_merge}(\text{Xs}, [\text{MY}\@\text{TY}|\text{Ys}], \text{Z}). \quad \% \text{cl1}.
\]
\[
\text{time\_binary\_merge}([\text{MX}\@\text{TX}|\text{Xs}], [\text{MY}\@\text{TY}|\text{Ys}], [\text{MY}\@\text{TY}|\text{Z}]) :-
\]
\[
\quad \text{TX} > \text{TY}:
\quad \text{time\_binary\_merge}([\text{MX}\@\text{TX}|\text{Xs}], \text{Ys}, \text{Z}). \quad \% \text{cl2}.
\quad \text{time\_binary\_merge}([\text{Xs}], [], []). \quad \% \text{cl3}.
\quad \text{time\_binary\_merge}([\text{MX}|\text{Msgs}], [], [\text{MX}|\text{Z}]) :-
\quad \text{time\_binary\_merge}(\text{Msgs}, [], \text{Z}). \quad \% \text{cl4}.
\quad \text{time\_binary\_merge}([\text{Xs}], [\text{MY}|\text{Msgs}], [\text{MY}|\text{Z}]) :-
\quad \text{time\_binary\_merge}([\text{Xs}], \text{Msgs}, \text{Z}). \quad \% \text{cl5}.
\]

Program 5.8. The don't-know relation time\_binary\_merge/3.

Instead of creating the choice point of an arbitrary time\_binary\_merge/3 goal, one
would like to select a goal that already has an input message with a known time
stamp and to transmit this message to the corresponding process. This may reduce
the chance of backtracking in a distributed simulation. Program 5.9 defines the
binary merge algorithm by a combination of a don't-care and don't-know relations.
The don't-care relation forces a goal to suspend, even on deadlock, until at least one
of its input arguments is an empty list (clauses 1 and 2), it does not matter which
one, or has a message with a known time component (clauses 3 and 4). It then
invokes a goal for the don't-know relation nondet\_time\_merge/3 which outputs
the messages ordered by their time stamps, and may backtrack if later on it is
discovered that they are not in the correct order.

\[
\text{mode} \quad \text{time\_binary\_merge}(?, ?, ^).\]
\[
\text{time\_binary\_merge}([\text{X}], \text{Y}, \text{Z}) \leftarrow
\quad \text{nondet\_time\_merge}(\text{Y}, [], \text{Z}). \quad \% \text{cl1}.
\]
\[
\text{time\_binary\_merge}(\text{X}, [], \text{Z}) \leftarrow
\quad \text{nondet\_time\_merge}(\text{X}, [], \text{Z}). \quad \% \text{cl2}.
\]
\[
\text{time\_binary\_merge}([\text{MX}\@\text{TX}|\text{Xs}], \text{Y}, \text{Z}) \leftarrow \text{data(\text{TX})}:
\quad \text{nondet\_time\_merge}([\text{MX}\@\text{TX}|\text{Xs}], \text{Y}, \text{Z}). \quad \% \text{cl3}.
\quad \text{time\_binary\_merge}(\text{X}, [\text{MY}\@\text{TY}|\text{Ys}], \text{Z}) \leftarrow \text{data(\text{TY})}:
\quad \text{nondet\_time\_merge}([\text{MY}\@\text{TY}|\text{Ys}], \text{X}, \text{Z}). \quad \% \text{cl4}.
\]
5. Pandora for Distributed Discrete Event Simulation

\[
\text{nondet_time_merge([MX@TX|Xs], [MY@TY|Ys], [MX@TX|Z]) :}
\]
\[
\begin{align*}
& TX \leq TY: \\
& \quad \text{time_binary_merge([MY@TY|Ys], Xs, Z).}
\end{align*}
\]
\[
\text{nondet_time_merge([MX@TX|Xs], [MY@TY|Ys], [MY@TY|Z]) :}
\]
\[
\begin{align*}
& TX > TY: \\
& \quad \text{time_binary_merge([MX@TX|Xs], Ys, Z).}
\end{align*}
\]
\[
\text{nondet_time_merge([], [], Z).}
\]
\[
\text{nondet_time_merge([M|Msgs], [], [M|Z]) :}
\]
\[
\text{nondet_time_merge(Msgs, [], Z).}
\]

Program 5.9. Merging two input channels into one ordered channel.

5.6 Related Work

A distributed discrete event simulation model was proposed in (Misra, 1986). The model is represented as a network of a fixed number of processes connected by fixed channels. As the topology of the network is static, it is difficult, for instance, to model the computer system in Section 5.3 for an arbitrary number of jobs and peripheral processors since these are dynamically introduced to the network.

Like our method, a process is not allowed to receive a message with time stamp T until it is certain that no message will ever arrive with a time stamp less than T. This means that a process must be blocked as long as any of its input channels is not carrying messages. Deadlock is either avoided by using the "null messages" algorithm, or is recovered after being detected, using the circulating markers.

Time-warp (Jefferson, 1985) is another DDES method in which the model is represented as a number of processes that can send messages to each other without restrictions. Each process has a single input channel and it proceeds execution assuming that the messages on its channel are ordered by increasing time stamps. However, when a message arrives with a time component T less than some that have already been received, then the Time-Warp mechanism must:

(a) roll back the process to a time just before simulated time T.
(b) execute the new message at simulated time T.
(c) re-execute messages with time stamps greater than T.
(d) cancel all the effects of any output messages that were sent after T during the last forward execution.

- 114 -
Cancellation of the effects of messages that should have never been sent is done by sending \textit{anti-messages}. The mechanism used is reminiscent of distributed backtracking with undo messages. Every request to the time-warp operating system for sending a message creates a message and its anti-message. The anti-message will be saved in case a roll back would occur.

The time-warp method generally requires considerable overhead in the form of state-saving and handling of anti-messages in order to make roll back possible.

The programming technique described in Section 5.5 for deadlock recovery in a Pandora simulation program was originally proposed by Haridi and Brand in (Haridi and Brand, 1988). Haridi et. al. proposed Andorra Prolog: a logic programming language that is based on the basic Andorra model. Unlike Pandora, Andorra Prolog is intended to subsume Prolog. As a consequence, the language includes a cut operator as well as a commit operator (for supporting don't-care non-determinism). Moreover, the order of goals in a conjunction is maintained during execution, and the leftmost goal is selected for forking in the non-deterministic phase.

5.7 Conclusions

In this chapter we propose Pandora for distributed discrete event simulation. Logical processes can be defined by don't-care relations whose goals are executed concurrently. The processes communicate by sending and receiving messages with increasing time stamps on channels that are shared among them. Processes as well as channels can be dynamically introduced to the model.

The major shortcoming of a distributed simulation is deadlock for which a simple solution was introduced. Unlike the "null messages" algorithm, as well as the time-warp mechanism, our algorithm allows deadlock to occur then recovers from it. Deadlock is detected by the underlying computational model and does not require additional machinery, while recovering from deadlock is based on a non-deterministic choice of a message to be transmitted and backtracking if the wrong choice was taken. While the algorithm may lead to backtracking as in the time-warp algorithm, it does not require the overhead of saving anti-messages.
An alternative algorithm for deadlock recovery without creating choice points is proposed in the following chapter; it uses the meta-level Pandora programming tools for breaking deadlock.
Chapter 6

The Pandora Deadlock Handler

In Chapter 3, the operational semantics of Pandora programs was explained. Goals in a parallel conjunction are evaluated concurrently in the and-parallel phase. When the computation deadlocks, the default is to select an arbitrary suspended goal for a don't-know relation and create a choice point for it. An and-parallel phase is then begun for each search branch. This chapter introduces the deadlock handler relation, by which the user can explicitly program the behaviour on deadlock.

The deadlock handler relation provides a powerful way to handle deadlock in a manner suited to the particular application. One way in which it can be used is in combination with the non-deterministic fork, to implement a heuristic search. This is a way to reduce the search space by intelligently selecting a don't-know non-deterministic goal to execute, when several are possible. What constitute an intelligent choice will be specific to a particular application, so it clearly cannot be left to an implementation to decide. Examples of various application-dependent heuristics which are programmed using the deadlock handler relation are illustrated in Section 6.2.

More generally, the deadlock handler relation can be utilized to manipulate suspended goals at the time of deadlock in a more flexible manner. Redundant goals can be removed, new goals can be added, and a group of goals can be replaced by a simpler group. Section 6.3 introduces a non-forking deadlock breaking mechanism using the deadlock handler relation.

Finally, Sections 6.4 and 6.5 discuss the development of constraint logic programming (CLP) systems in Pandora. In these examples, all the features of the
deadlock handler relation are utilized. Namely, redundant constraints are removed, alternative sets of constraints are generated by forking the computation, and complex sets of constraints are replaced by simpler ones which are evaluated in a new and-parallel phase.

6.1 The Deadlock Handler Relation

The operational semantics of Pandora programs, which was defined and used in the previous chapters illustrates the default mechanism for breaking deadlock. Namely, the selection of an arbitrary suspended goal for a don't-know relation and the creation of a choice point for it, using its don't-know procedure. This section introduces the meta-level deadlock handler relation, by which the user can explicitly program the behaviour on deadlock.

The deadlock handler relation is a don't-care relation whose mode declaration is:

\[
\text{mode deadlock_handler(Meta\_Suspended?, Optimum\_Goal\^, Removed\_Goals\^, New\_Goals\^).}
\]

If `deadlock_handler/4` is not defined in the program, the default mechanism for breaking deadlock is applied. Otherwise, its procedure is invoked at the time of deadlock with the input argument, `Meta\_Suspended`.

`Meta\_Suspended` is a list of ground terms, each representing a suspended goal in the current conjunction of goals. A variable `X` in a goal is represented by a unique constant. If shared among several suspended goals, `X` has the same representation in all the ground terms corresponding to these goals.

For instance, if the computation deadlocks with the following goals being suspended:

\[
g1(X, a, Y), g2(Y, Z, Y), g3(f(X), b)
\]

then the input argument for `deadlock_handler` can be:

\[
\text{Meta\_Suspended } = \text{[g1('X', a, 'Y'), g2('Y', 'Z', 'Y'), g3(f('X'), b)]}
\]

Suppose, that the computation deadlocks with the following conjunction of goals:

\[
(g1 \& g2), g3
\]
Then, the suspended goals are \( g_1 \) and \( g_3 \) while \( g_2 \) is not suspended; its evaluation is delayed until \( g_1 \) succeeds. Hence, the input argument for \texttt{deadlock_handler} will be the list \([g_1, g_3]\).

A goal for the deadlock handler relation may either succeed or fail but must not deadlock. If it did deadlock, then the consequential re-entry of the deadlock phase will re-invoke the deadlock handler procedure which may in turn deadlock. This may lead to an infinite computation.

The second, third, and fourth arguments of \texttt{deadlock_handler/4} are output arguments whose bindings depend on the particular application that is programmed. The second argument, \texttt{Optimum_Goal}, allows the programmer to select an input term which represents a particular suspended goal for a don't-know relation; a choice point will be created for the selected goal and a new and-parallel phase will then begin for each search branch.

Alternatively, the third and fourth arguments of \texttt{deadlock_handler/4} provide means for breaking deadlock without forking the computation. Using the third argument, \texttt{Removed_Goals}, the user can specify a list of terms from its input argument. When the evaluation of \texttt{deadlock_handler/4} terminates successfully, the goals corresponding to these terms will be removed from the existing conjunction of goals, effectively replaced by calls to 'true'. For instance, if \texttt{Removed_Goals} is bound to \([g_1, g_3]\) in the above example, then the computation will proceed in a new and-parallel phase evaluating \( g_2 \).

The fourth argument, \texttt{New_Goals}, can be bound to a list of goals to be added to the current conjunction of goals, effectively replacing a call to 'true' by the conjunction of the new goals. Adding new goals can also break the deadlock of the computation by instantiating unbound variables on which goals are suspended. The computation will then proceed in a new and-parallel phase.

While the second argument of \texttt{deadlock_handler/4} provides a forking means of breaking deadlock, the third and fourth arguments provide a non-forking deadlock breaking mechanism. The semantics of the deadlock handler relation restricts the use of exactly one of them at one time. That is, only one of the following cases can result from a successful evaluation of \texttt{deadlock_handler/4}:
6. The Pandora Deadlock Handler

(a) the second argument is bound to an input term, while both the third and fourth arguments are bound to empty lists.
(b) at least the third or the fourth argument is instantiated to a non-empty list while the second argument is unified with the constant '$no_choice' to explicitly indicate a non-forking deadlock breaking mechanism.

The role of each of these output arguments is illustrated by various examples in the rest of the chapter. The definition of the deadlock handler relation in all these examples is sound with respect to the intended meaning of the program. That is, any solution that is computed by the program is a solution according to the intended meaning of the program. Selecting a don't-know relation goal for a non-deterministic reduction always preserves the soundness property. However, if the state of computation is changed by removing existing goals or adding new ones, then the new state of computation should ideally be equivalent to the deadlocked state of computation for the intended interpretation of the program.

Suppose, for example, that the computation is deadlocked with the following suspended goals:

\[ 1 < Y, \ Y < 3 \]

One definition of `deadlock_handler/4`, which preserves the above property (in the domain of integer numbers), may select both of the above goals for removal and add the new goal:

\[ Y = 2 \]

6.2 Programming Heuristic Search

One way of utilizing the deadlock handler relation in a Pandora computation is by implementing a heuristic search. This is a way to reduce the search space by intelligently selecting a don't-know non-deterministic goal to execute, when several are possible. The underlying implementation of Pandora can be enhanced with several heuristics for selecting the goal. However, such heuristics may not necessarily reduce the search space of all programs; what constitute an intelligent choice is usually specific to a particular application.

Using the deadlock handler relation, the user can explicitly program the application-
dependent heuristics. After a successful evaluation of deadlock_handler/4, the second argument of the goal should be bound to the meta-level representation of the selected goal. A choice point for the corresponding goal will be then created and a new and-parallel phase will begun for each search branch.

Several heuristics are defined in this section using the deadlock_handler/4 relation, and their effects on a Pandora computation are illustrated. In all these examples, the third and fourth arguments are bound to '[]'.

6.2.1 Programming the Andorra Deadlock Breaking Mechanism

In a basic Andorra evaluation, the order of goals is preserved during the computation and, when the computation deadlocks, the leftmost goal is selected for forking.

In order to simulate the Andorra deadlock breaking mechanism in a Pandora program with only don't-know relations, two extra arguments are added to each relation in the program, by which the goals in the query are connected in a chain. The left end of the chain is instantiated to a constant, say 'left', while the right end is a variable. For example, Program 6.1 below redefines the Prolog quick sort in Program 2.10 in Chapter 2 after adding the extra arguments. Suppose that the Andorra query is:

```
qsort(L,SortedL,[]), produce(L), consume(SortedL)
```

Then, its corresponding Pandora query would be:

```
qsort(left, R1, L, SortedL, []),
produce(R1, R2, L), consume(R2, R3, SortedL)
```

When a goal is reduced, it replaces itself in the chain with the goals in the body of its clause, in their correct order. If the body of the clause is empty, the goal removes itself from the chain by unifying its left and right connections.

---

6 This is similar to the programming technique described in Chapter 5 for adopting constraint-based algorithms in Pandora.
At any time during the computation, the leftmost goal according to the textual order in the program will have its first argument instantiated to the constant 'left', while the other goals will have an unbound variable in their first argument position. However, the leftmost goal may be anywhere in the current conjunction of goals. In order to select the leftmost goal for forking in the deadlock phase, deadlock_handler/4 can be defined as in Program 6.2. When the computation deadlocks, the deadlock_handler/4 goal searches the list of meta-level representations of the suspended goals until it finds the term whose first argument instantiated to the constant 'left'. This term represents the leftmost suspended goal.

deallock_handler/4 then succeeds after unifying its second argument with the selected term.

Program 6.2. Using the deadlock_handler/4 relation to simulate the basic Andorra model.

6.2.2 Making Choices in the N-Queens Problem

The n-queens problem has been solved by various Pandora programs in the previous chapters. In Chapter 4, the problem was efficiently solved by a special programming technique which applies constraint-based reasoning to remove unnecessary squares from the board once a queen is placed. Moreover, a queen is eagerly assigned to a square provided it is the only remaining square in a row or a
column. When the computation deadlocks, a queen is non-deterministically assigned to an arbitrary remaining square in the board and future failure would retract this assignment and assign the queen to an alternative square.

One improvement to Program 4.5 is to apply the first-fail principle when non-deterministically placing a queen on the board. This principle says that:

"in order to succeed, try first where you are the most likely to fail".

The aim is to detect failure (if it exists) as early as possible in order to avoid unnecessary computation.

In the n-queens problem, a queen can be non-deterministically placed on the most constrained available square on the board. This is the square whose row and/or column have the smallest number of remaining squares. Assigning a queen to this square is more likely to fail than assigning it to any other square. Several heuristics have been applied for identifying the most constrained square; these are described below with the assumption of having row 1 as the topmost row and column 1 as the leftmost column.

**Heuristic 1 (H1)**

1. select the row which has the least number of remaining squares and is not yet assigned a queen. If there are several rows with the same least number of squares, select an arbitrary one of them.
2. place a queen on the leftmost available square in the selected row.

**Heuristic 2 (H2)**

1. select the row which has the least number of remaining squares and is not yet assigned a queen. If there are several rows with the same least number of squares, select the topmost one.
2. place a queen on the leftmost available square in the selected row.

**Heuristic 3 (H3)**

1. count the number of remaining squares in each row and column which are not yet assigned queens. Let LS_Row be the least number of squares in any row and LS_Col be the least number of squares in any column. If 

   \[ \text{LS}_\text{Row} \leq \text{LS}_\text{Col} \]
then select the rows with LS_Row squares. Otherwise, select the columns with LS_Col squares.

2. for each available square in a selected row (or column), assign a cost value that is equal to the number of remaining squares in its column (or row).

3. place a queen on the square with the smallest cost value. If there are several squares with the same smallest cost value, select the topmost leftmost square (in case of the columns select the leftmost topmost square).

For example in Figure 6.1 below, the number of remaining squares in each available row/column in the board is surrounded by an oval on the sides of the board. The topmost row represents row 1 while the leftmost column represents column 1. H1 can select either row 3 or row 4. Suppose that row 4 is selected. A queen is then placed on the leftmost square which is in row 4 and column 1. H2 only selects the topmost row with the least number of squares. Hence, it selects row 3 and a queen is placed on the square in row 3 and column 2. H3 selects both rows 3 and 4. The cost value for each square in these rows is the same, that is 3. Then, the topmost square is selected, which is the square in row 3 and column 2.

![Figure 6.1. A 5*5 board for the n-queens problem.](image)

The number of backtracking that have resulted from running the n-queens program with heuristics (H1, H2, and H3) are illustrated in Table 6.1, for different number of queens (i.e. different N values). The column with a heading "No-H" illustrates the number of backtracings when running the program in Chapter 4 without applying any heuristics.

The deadlock_handler/4 relation in Program 6.4 implements heuristic 1; the other heuristics are defined in a similar manner. As described in Chapter 4, the squares in the same row are connected in a manner which ensures that exactly one queen is present. 

- 124 -
placed in that row. Each square in row i is represented by the following conjunction:

\[
\text{less}_\text{eq1}(\text{Onij}, J, Q_j), \quad \text{greater}_\text{eq1}(\text{Onij}, J, _, _, \text{RLleft}, \text{RRight})
\]

where \( Q_i \) is shared among all the \text{less}_\text{eq1} goals for the squares in row i. The same technique is applied to ensure that exactly one queen is placed in a column and the following goals connect a square to its neighbours in the same column:

\[
\text{less}_\text{eq1}(\text{Onij}, I, Q_j), \quad \text{greater}_\text{eq1}(\text{Onij}, I, _, _, \text{CLeft}, \text{CRight}),
\]

Moreover, the squares in the same diagonal are connected in a manner which ensures that at most one queen is placed in a diagonal. In order to implement heuristic 1, an extra argument was added to the \text{less}_\text{eq1}/3 relation in Program 4.4. The result is the definition in Program 6.3 below. A goal for the \text{less}_\text{eq1}/4 relation will have its first argument bound to either 'r', 'c' or 'd' to denote whether it connects the square to its neighbours in the same row, column or diagonal respectively.

\[
\text{less}_\text{eq1}(_, \text{on}, \text{Val}, \text{Val}).
\]
\[
\text{less}_\text{eq1}(_, \text{off}, _, _).
\]

**Program 6.3.** The \text{less}_\text{eq1}/4 don't-know relation.

<table>
<thead>
<tr>
<th>N</th>
<th>No-H</th>
<th>H1</th>
<th>H2</th>
<th>H3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>21</td>
<td>20</td>
<td>20</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>21</td>
<td>6</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>37</td>
<td>27</td>
<td>32</td>
<td>9</td>
</tr>
<tr>
<td>14</td>
<td>258</td>
<td>25</td>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>1271</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 6.1.** The number of backtrackings in the n-queens problem with and without the first-fail principle.

A goal for \text{deadlock_handler}/4 in Program 6.4 spawns three concurrently executing goals: \text{arrange}_\text{by}_\text{rows}/2, \text{select}_\text{row}/2 and \text{select}_\text{square}/2. \text{arrange}_\text{by}_\text{rows}/2 accepts the meta-level representation of the suspended goals in \text{Meta}_\text{Suspended} and selects from it all the \text{less}_\text{eq1} terms whose first arguments
instantiated to 'r'.

The output of arrange_by_rows/2 is By_Rows: [Rowk, Rowm, ..., Rown], a list representing all the rows in the board which are not yet assigned queens. Each Rowi in By_Rows is a tuple (Goals, Counter), where Goals is a list of the less_eq1(_, _, _, Qi) terms for the squares in row i (i.e. the terms sharing the same meta-level constant in their fourth arguments), and Counter is their total number.

select_row/2 accepts By_Rows while it is being produced and selects one of the rows with the fewest remaining squares, i.e. with the smallest value of Counter, while select_square/2 (Program 6.4) selects the less_eq1 term which represents the leftmost remaining square in the selected row; this is the term with the least column number in its third argument.

Program 6.4. Selecting the leftmost square in the most constrained row in the n-queens board.
6.2.3 Making Choices in Fly-Pan

Using Pandora to program Fly-Pan as described in Chapter 4 reduces the search space dramatically. However, since the problem under consideration is computationally complex, it is most likely that the and-parallel phase of the computation will not be sufficient to assign an aircraft to each input flight, and choices have to be made for the values of some decision variables. In order to increase the efficiency of the system even further, several heuristics are defined in the Fly-Pan deadlock handler (Program 6.5) which are applied in the following order:

1. Selecting a Flight
   There are two alternative methods for selecting a flight from the remaining unassigned ones:

   (a) Flights' Priorities
       The user may optionally specify a priority for each input flight as its first attribute. In this case, an input flight would be represented by the structure:

       \[
       \text{flight(Priority, Id, Launch, Landing, Type, Crew)}
       \]

       The unassigned flight with the highest priority is the one chosen to be assigned an aircraft. Since the flights' priorities are input to the program, the flights can be given different priorities each time the program is invoked, reflecting the changes in the problem's environment without the need to modify the program itself.

   (b) First-Fail Principle
       If the user does not specify priorities for the flights, then the flight with the least number of choices is selected since it is likely to be the most difficult one to assign an aircraft.

select_cells/2 in Program 6.5 accepts Meta_Suspended: the list of ground terms representing the suspended goals and returns Cells: the list of all cell terms in Meta_Suspended. The cell terms represent the aircraft in the sets of possible aircraft of all input flights. choose_flight/2
accepts Cells and returns Flight: a list of those cell terms that represent the set of possible aircraft for the selected flight.

mode deadlock_handler(? , ^ , ^ , ^).

deadlock_handler(Meta_Suspended, Optimal, [], []) <-
    select_cells(Meta_Suspended, Cells),
    choose_flight(Cells, Flight),
    choose_aircraft(Meta_Suspended, Flight, Optimal).

Program 6.5. The deadlock_handler/4 relation in Fly-Pan.

2. Selecting an Aircraft

Applying the first-fail principle, the most constrained aircraft is chosen from the set of possible aircraft for the selected flight and is assigned to it. There are several ways of identifying the most constrained aircraft. In Fly-Pan, it is the aircraft with the smallest number of remaining flying hours since assigning it to the flight is more likely to remove it from other flights' sets of possible aircraft, reducing the search space and leading to a more efficient problem-solving behaviour.

For each input aircraft, a don't-care relation goal:

    current_max(Aircraft, _, RemainingHrs, _)

holds a counter of the remaining flying hours for the aircraft as its third argument while the first argument for the goal is the identity of the aircraft itself. When the aircraft is assigned to a flight, current_max/4 reduces the remaining flying hours for the aircraft by the flight's length and recursively spawns a current_max/4 goal with the new counter of the flying hours then terminates successfully. When the computation deadlocks, current_max/4 will be represented by a ground term in Meta_Suspended.

choose_aircraft/3 in Program 6.6 selects the cell term in Flight which represents the aircraft with the smallest counter value. It spawns three concurrently executing processes:

1. select_remaininghrs/2, which scans Meta_Suspended searching for all the current_max/4 terms and returns RemainingHrs: a list of pairs, each of the form (Aircraft,FlyingHours), representing the input aircraft with their remaining flying hours.

2. aircraft_cost/3, which accepts Flight as well as RemainingHrs and
assigns a *cost value* to each possible aircraft for the selected flight. The cost value for each aircraft is its remaining flying hours. 

aircraft_cost/3 produces Costs: a list of all the cell terms in Flight together with the cost values of the corresponding aircraft.

3. min_cost/2, which selects the aircraft with the minimum number of remaining flying hours.

```prolog
mode choose_aircraft(? , ?, ^).
choose_aircraft(Meta_Suspended, Flight, Aircraft) <-
    select_remaininghrs(Meta_Suspended, RemainingHrs),
    aircraft_cost(Flight, RemainingHrs, Costs),
    min_cost(Costs, Aircraft).
```

**Program 6.6.** Choosing the most constrained aircraft that can be assigned to the selected flight.

The deadlock handler of the Fly-Pan program illustrates the need to inspect the suspended don’t-care relation goals (e.g. *current_max*/4) as well as the suspended don’t-know relation goals in order to select an optimal don’t-know relation goal for forking.

### 6.3 Non-Forking Deadlock Breaking Mechanism

In some applications, a more powerful control than a simple heuristic search is required. Goals should be inspected and manipulated in different ways; redundant goals should be removed, new goals should be added, existing goals should be replaced by alternative ones, and inconsistent goals should be detected and possibly fail the computation.

The goal:

```
deadlock_handler( Meta_Suspended, Optimum_Goal,
    Removed_Goals, New_Goals)
```

may succeed after unifying its third argument, Removed_Goals, with a list of ground terms (possibly empty) from its input argument, Meta_Suspended. The corresponding suspended goals are then removed from the current conjunction of
goals. Additionally, \texttt{deadlock_handler/4} may bind its fourth argument, \texttt{New_Goals}, to a non-empty list of terms; these represent new goals which the user would like to add to the current conjunction of goals. Removing existing goals or adding new goals may break the deadlock of the computation as previously explained in Section 6.1. Then, the computation proceeds in an and-parallel phase.

Several examples are given in this section in which deadlock is broken without creating a choice point. In all these examples, \texttt{deadlock_handler/4} unifies its second argument, \texttt{Optimum_Goal}, with the constant \texttt{'no_choice'} in order to explicitly indicate that no forking is required.

### 6.3.1 Advancing Time in a Process Interaction Discrete Event Simulation

In the process interaction discrete event simulation method, a real system is simulated by a number of concurrent logical processes, each representing activities in the real system. The simulated time is represented by a central clock process which controls the times at which activities are simulated. A process simulating an activity at time $T_i$ would send an alarm request to the clock process and receive an alarm signal in reply when the simulated time is $T_i$. It can then start its activity. Events that are scheduled at the same time can be processed simultaneously by the concurrent logical processes. After executing all the events at $T_i$, the logical processes suspend and the clock process advances the simulated time to $T_i+1$: the time of the next simultaneous events.

One way of implementing the central clock process in Pandora is illustrated in Program 6.7. The first argument of \texttt{clock/3} is the current simulated time. The second argument, \texttt{Events}, is a chronologically ordered list of the alarm signals to be sent. The clock process receives alarm requests from other processes in the system, each of the form \texttt{hold(Delay, Alarm)}. \texttt{Delay} is the delaying period, from the current simulated time, after which an event should take place. For each alarm request (clause 2), the process computes the time for the event to occur, \texttt{EventTime}, and adds the alarm signal \texttt{e(EventTime, Alarm)} to \texttt{Events}. If the scheduled time for the next event in \texttt{Events} is equal to the current simulated time (clause 3), the clock process instantiates \texttt{Alarm} to that time.
mode clock(?, ?, ?).
clock(T, [], []). % cl1.
clock(Time, Events, [hold(Delay, Alarm)|Requests]) <-
    EventTime is Time + Delay,
    order_insert(e(EventTime, Alarm), Events, Events1),
    clock(Time, Events1, Requests). % cl2.

clock(EventTime, [e(EventTime, Alarm)|Events], Requests) <-
    Alarm = EventTime,
    clock(EventTime, Events, Requests). % cl3.

Program 6.7. The clock process in a process interaction simulation.

When all the scheduled events for the present time have been processed, all the
processes suspend waiting for the simulated time to be advanced. A goal for the
deadlock_handler/4 relation in Program 6.8 will then bind its third and fourth
arguments to singleton lists. The member of the list in the third argument is the
meta-level representation of the suspended clock process, while the member of the
list in the fourth argument is a term representing a new clock process with the
advanced simulated time in its first argument. After the successful evaluation of
deadlock_handler/4, the suspended clock process will be replaced by the new
clock process. An alarm signal will be then sent by the new clock process (clause 3
in Program 6.7) to start the next scheduled event.

mode deadlock_handler(?^, ^, ^).
deadlock_handler( Meta_Suspended, "$no_choice",
    [OldClock], [NewClock]) <-
    get_clock(Meta_Suspended, OldClock),
    advance_time(OldClock, NewClock).

mode advance_time(?^).
advance_time( clock(T, [e(EventTime, Alarm)] Events], Request),
    [clock(EventTime, [e(EventTime,Alarm)|Events], Request]).

Program 6.8. The deadlock handler in a process interaction simulation.
6.3.2 Deadlock Recovery in a Distributed Discrete Event Simulation

Unlike the process interaction method, there is no central clock process in a distributed discrete event simulation (DDES). All the processes in a DDES program are executed concurrently and communicate by sending messages on shared channels. Each message is stamped with the time at which it leaves a process. In order to preserve the synchronization of the simulated system, each process should receive its input messages in the same order as received by its physical counterpart. For example, if a process receives a message M1@T1 before a message M2@T2, then T2 should be greater than or equal to T1. The major problem with a distributed simulation is deadlock. This is the situation when all the remaining processes in the system are waiting for messages to arrive on their input channels.

The distributed discrete event simulation method was defined in Chapter 5 and used to simulate a primitive computer system by a Pandora program. In order to preserve the synchronization of the simulated system, all the input channels for a process in a Pandora simulation program are merged into one channel in which messages are in increasing order of their time stamps. For each two input channels X and Y for a process, a \texttt{time\_binary\_merge(X, Y, Z)} goal is spawned to merge the messages in X and Y in the correct order and produces the single channel Z; Z is then merged with other input channels (if any) for the same process. The result is a network of \texttt{time\_binary\_merge} goals connecting the various processes in the model.

In Chapter 5, a simple algorithm for breaking deadlock in a DDES Pandora program was proposed. In this algorithm, an arbitrary message M1@T1 is selected in the deadlock phase and is non-deterministically sent to its receiving process. If later in the computation the process receives a message M2@T2 such that T2 is less than T1, M1@T1 is retracted and M2@T2 is sent instead; any computation based on the transmission of M1@T1 should be undone.

Instead of selecting an arbitrary message, one could select the message M1@T1 which has the least time stamp among all the messages in the current state of computation. This will guarantee that no message will ever arrive in the future with a time component less than T1 and hence the computation will never be undone. Program 6.9 redefines the \texttt{time\_binary\_merge} relation which was previously...
defined in Program 5.9. The new definition of the relation is deterministic and does not include any calls to don't-know relations. A `time_binary_merge(X, Y, Z)` goal suspends in one of the following cases:

1. if X and Y are unbound variables. That is, when no message yet arrived in any of the channels.
2. if X or Y is unbound while the other is bound to an empty list. The empty list implies that no message will arrive in this channel in the future.
3. if X or Y is unbound while the other is bound to a non-empty list of message(s).

```
mode time_binary_merge(? , ?, A).

time_binary_merge([], [], []). 

time_binary_merge([], [Msg|Y], [Msg|Z]) <- 
    time_binary_merge([], Y, Z).

time_binary_merge([Msg|X], [], [Msg|Z]) <- 
    time_binary_merge(X, [], Z).

time_binary_merge([MX@TX|Xs], [MY@TY|Ys], [MX@TX|Zs]) <- 
    TX <= TY:
        time_binary_merge(Xs, [MY@TY|Ys], Zs).
    time_binary_merge([MX@TX|Xs], [MY@TY|Ys], [MY@TY|Zs]) <- 
    TX > TY:
        time_binary_merge([MX@TX|Xs], Ys, Zs). 
```

Program 6.9. The don't-care relation `time_binary_merge/3`.

When the computation deadlocks, `deadlock_handler/4` (Program 6.10) is invoked which, in turn, spawns three concurrently executing processes: `get_time_merges/2`, `select_least/2`, and `send_message/2`. `get_time_merges/2` accepts `Meta_Suspended` and returns a list of all the `time_binary_merge` terms, each in one of the following forms:

1. `time_binary_merge(X, [M@T|Msgs], Z)`
2. `time_binary_merge([M@T|Msgs], Y, Z)`

If no such term exists in `Meta_Suspended`, i.e. all the messages in the simulation were already processed, then `get_time_merges` returns a singleton list of an arbitrary `time_binary_merge` term with one of its first two arguments instantiated to an empty list.
select_least/2 accepts the list produced by get_time_merges and selects the time_binary_merge term with the least time-stamped input message. If there is only one term in the list, that term is selected. The goal corresponding to the selected term will be removed from the current conjunction of goals after the successful evaluation of deadlock_handler/4 and the message with the least time stamp will be transmitted in the simulation when adding the new goals to the current conjunction; these new goals are determined by send_message/2 (clauses 2 and 3).

Clauses 4 and 5 in Program 6.10 implements the case when the selected time_binary_merge term has an empty list in one of its arguments. The send_message/2 goal will then generate new goals to unify the other arguments of the time_binary_merge term with the empty list, terminating the communication along these channels.

```prolog
mode deadlock_handler(?, ^, ^, ^).
deadlock_handler( Meta_Suspended, 'no_choice', [Time_Binary_Merge], New_Goals) <-
    get_time_merges(Meta_Suspended, Time_Merge_Goals),
    select_least(Time_Merge_Goals, Time_Binary_Merge),
    send_message(Time_Binary_Merge, New_Goals).

mode send_message(?^).
send_message( time_binary_merge([MX@TX|Xs], Y, Z),
    [Z = [MX@TX|Zs], time_binary_merge(Xs,Y,Zs)]). %cl2.
send_message( time_binary_merge(X, [MY@TY|Ys], Z),
    [Z = [MY@TY|Zs], time_binary_merge(X,Ys,Zs)]). %cl3.
send_message(time_binary_merge([], Y, Z), [Y = [], Z = []]). %cl4.
send_message(time_binary_merge(X, [], Z), [X = [], Z = []]). %cl5.
```

Program 6.10. The deadlock handler for a distributed simulation.

6.4 Implementing Constraint Logic Programming Systems

A common method for implementing experimental constraint logic programming (CLP) systems is to write meta-interpreters for them in logic programming languages. Among the systems written this way are CLP(Σ*) (Walinski, 1989),
6. The Pandora Deadlock Handler

CAL (Sakai and Aiba, 1987), and CLP*(X) (Hickey, 1989). These systems are all built on top of Prolog but differ in the domains of computation they are handling. The domain of computation, in turn, affects the type of constraints provided by the system as well as the constraint solver which solves them. Since goals in a Prolog conjunction are executed sequentially, none of these systems supports stream and-parallelism.

In Chapter 4, we described techniques for adopting constant propagation in a Pandora program as a means of efficiently solving a constraint problem. The techniques proposed are based on the concept of set variables with their finite domains; these can be used for various constraint problems, including finite combinatorial constraint-solving problems. However, more specialized constraint solvers and optimization mechanisms are required for solving complex type of constraints as well as solving constraint problems in infinite or continuous domains (Bahgat, 1988).

Suppose, for example, that the computation is deadlocked and the following goals are suspended:

\[ 1 < Y, \ Y < X, \ X \leq 3, \ X \text{ is } 5 - Y \]

Since these goals are for don't-care relation primitives, the default Pandora implementation would report a deadlock situation without producing a solution to the query. However, the deadlock handler relation can be defined to inspect the meta-level representation of the suspended constraints and to apply specialized algorithms for solving these constraints. In this example, the suspended goals could be represented by:

\[ \text{Meta\_Suspended} = [1 < 'Y', 'Y' < 'X', 'X' \leq 3, 'X' \text{ is } 5 - 'Y'] \]

Assuming that the domain of computation is the domain of Integers, deadlock_handler/4 may infer from the three inequality goals that \(1 < Y < 3\) and hence \(Y\) is equal to 2. As a consequence, deadlock_handler/4 should succeed after binding its third argument, Removed_Goals, to \([1 < 'Y', 'Y' < 'X']\) and its fourth argument, New_Goals, to \(['Y' = 2]\). A new and-parallel phase will then begin after removing the goals corresponding to the terms in Removed_Goals, and adding the goals corresponding to the terms in New_Goals. In this particular example, \(Y\) will be unified with 2, the "X is 5 - Y" goal will succeed after unifying X with 3, and the computation will terminate successfully.
Program 6.11 illustrates an abstract definition of the deadlock handler in a Pandora constraint program. The \texttt{saggregate/3} goal scans \texttt{Meta\_Suspended} and produces:

1. \texttt{Dont\_Know}: a list of the meta-level terms representing suspended goals for don't-know relations,
2. \texttt{Constraints}: a list of the meta-level terms representing the suspended constraints.

\texttt{consistent/1} checks the satisfiability of the constraints and fails if inconsistent constraint goals are suspended. For example, suppose the following constraint goals are suspended:

\[
X < 2, X > 4
\]

Since there is no value for \(X\) which would satisfy both inequalities, the consistent goal should fail.

If the consistent goal succeeds, \texttt{decide\_on\_behaviour/5} is spawned to decide on the mechanism for breaking deadlock based on the bindings of \texttt{Dont\_Know} and \texttt{Constraints}. If \texttt{Dont\_Know} is a non-empty list of terms which represent suspended goals for don't-know relations (clause 2), then a term representing a don't-know relation goal is selected so that the corresponding goal is non-deterministically reduced. This goal either generates alternative sets of constraints or alternative values for a constrained variable(s).

If no such goal exists (clause 3), \texttt{clp\_solver/2} is invoked to apply specialized algorithms and optimization techniques for solving the suspended set of constraints. These algorithms depend on the particular domain of computation. The \texttt{clp\_solver/2} goal produces a new set of constraints which may include equality constraints that assign values to unbound variables (such as \(Y = 2\) in a previous example). The new set of constraints is then compared with the old one in order to determine which of the old constraints should be removed and what are the constraints which should be added to the current conjunction of goals.

It should be noted that \texttt{clp\_solver/2} is only called when \texttt{Dont\_Know} is an empty list, i.e. when no other choice is available. The reason for delaying the call to \texttt{clp\_solver/2} is that the cost of applying specialized algorithms to solve the constraints is usually high.
mode deadlock_handler(? , ^ , ^ , ^) .

deadlock_handler(Meta_Suspended, Optimal, Removed, New)<-
saggregate(Meta_Suspended, Dont_Know, Constraints),
consistent(Constraints) &
decide_on_behaviour( Dont_Know,Constraints,
Optimal, Removed, New). % cl1.

mode decide_on_behaviour(? , ? , ^ , ^ , ^) .

decide_on_behaviour([Generator|Dont_Know], Constraints, Optimal,
[], []) <-
select_optimal([Generator|Dont_Know], Optimal). % cl2.

decide_on_behaviour([], OldConstraints, '$no_choice', Removed, New) <-
clp_solver(OldConstraints, NewConstraints),
compare(OldConstraints, NewConstraints, Removed, New). % cl3.

Program 6.11. The deadlock handler for implementing a CLP constraint solver.

6.5 Implementing Hierarchical Constraint Logic Programming Systems

CLP languages, as well as other constraint systems, only allow the programmer to specify constraints that must hold. In many applications, such as interactive graphics, page layout, and decision support, one needs to express preferences as well as strict requirements. A new scheme for extending CLP to include both required and preferential constraints, with arbitrary levels of preference was first proposed in (Borning et al., 1988). The scheme was called the hierarchical constraint logic programming (HCLP) scheme.

Consider an interactive graphics example from ThingLab (Borning, 1981). Suppose there is a horizontal line displayed on the screen, and one endpoint is going to move with the mouse. There is a required constraint that the line remains horizontal, a preference that one endpoint of the line follows the mouse, and a weaker preference that the endpoints of the line remain fixed. The weak preference gives stability to the line as it is moved so that, for example, it does not triple in length as we move the endpoint by the mouse. The constraints in this example can be expressed in an HCLP language as illustrated in Program 6.12. The arguments of move_horiz_end2/3 are terms representing the old and new states of the
horizontal line, and a third term that is the x-y distance by which one endpoint should be moved.

A constraint in an HCLP program is labelled by a symbolic name which denotes the level of preference of that constraint, and all the symbolic names that are used in the program are declared at the beginning of the program. For instance in Program 6.12,

```prolog
level([require, prefer, weak]).
```

declares three level of preferences: require, prefer, and weak. The symbolic names should appear in the declaration in a decreasing order according to their levels of preference. Hence, the above declaration denotes that the constraints with a require label are more preferred than the ones with a prefer label, and these in turn are more preferred than the ones with a weak label. Moreover, the first symbolic name in the declaration is always reserved to label the required constraints, i.e. the constraints which should be satisfied by all the solutions to the problem.

```prolog
/* set up symbolic constraint strengths */
levels([require, prefer, weak]).
move_horiz_end2( line_segment(OldX1, OldY1, OldX2, OldY2),
  line_segment(NewX1, NewY1, NewX2, NewY2),
  delta(DX, DY)) :-
  require OldY1 = OldY2,  require NewY1 = NewY2,
  prefer NewX2 = OldX2 + DX,  prefer NewY2 = OldY2 + DY,
  weak OldX1 = NewX1,  weak OldY1 = NewY1,
  weak OldX2 = NewX2,  weak OldY2 = NewY2.
```

**Program 6.12.** An HCLP program.

A solution to an HCLP query is a substitution for the unbound variables in the query such that, after it is applied, all the required constraints must be satisfied. In addition, the substitution must satisfy the preferential constraints as much as possible, respecting their relative strengths. Different solutions can be obtained by considering different combinations of preferential constraints, together with the required ones. Each HCLP system includes a comparator to compare those solutions and select the best ones according to the definition of the comparator. The
following is a definition of one kind of comparators, called the locally-predicate-better comparator.

**Definition:**
A solution $\theta$ is *locally-predicate-better* than another solution $\sigma$ if both of the following conditions are satisfied:

1. each constraint up to some level $k - 1$ that holds after applying $\sigma$ also holds after applying $\theta$.
2. each constraint at level $k$ is either satisfied by $\theta$ or not satisfied by any of the solutions $\theta$ and $\sigma$. Moreover, there is at least one constraint at level $k$ that is satisfied by $\theta$ but is not satisfied by $\sigma$.

Consider, for example, the following query for Program 6.12:

```
move_horiz_end2( line_segment(3, 4, 5, 4) ,
    line_segment(NewX1, NewY1, NewX2, NewY2),
    delta(2, 1))
```

One solution to the query would be:

```
{ NewX1/3, NewY1/4, NewX2/7, NewY2/4}
```

The above solution does not satisfy the preferential constraints:

- prefer $\text{NewY2} = \text{OldY2} + \text{DY}$
- weak $\text{OldX2} = \text{NewX2}$

A locally-predicate-better solution would be:

```
{NewX1/3, NewY1/5, NewX2/7, NewY2/5}
```

since both solutions satisfy the required constraints as well as the preferential constraint:

- prefer $\text{NewX2} = \text{OldX2} + \text{DX}$

while only the latter solution satisfies the constraint:

- prefer $\text{NewY2} = \text{OldY2} + \text{DY}$

A meta-interpreter for HCLP with the LPB comparator was implemented in Prolog and its algorithm was explained in (Borning et al, 1988). Program 6.14 defines an implementation of the system in Pandora using the deadlock handler relation.
In order to manipulate preferential constraints, an HCLP program is compiled into a Pandora-HCLP program. The level/1 declaration does not appear in the compiled form of the program but is used to map the symbolic names onto integers 0..n, where n is the number of preferential levels of constraints. The required constraints in the source program become unlabelled in the compiled form and each preferential constraint C with a label Strength is transformed into the goal:

\[
\text{constraint}(S\_\text{Value}, \ C)
\]

S_Value is the integer value to which Strength is mapped. Finally, the definition of the constraint/2 don't-know relation in Program 6.13 is added to the Pandora-HCLP program.

\[
\text{constraint}(\_\_, \ \text{Constraint}) : - \\
\quad \text{call}(\text{Constraint}). \quad \% \ cl1. \\
\quad \text{constraint}(\_\_, \ \_\_). \quad \% \ cl2.
\]

Program 6.13. A preferential constraint in a Pandora-HCLP program.

Unlike an HCLP program, goals in a Pandora-HCLP program are executed in parallel in the and-parallel phase. A constraint/2 goal is always non-deterministic and thus remains suspended during the and-parallel phase, while required constraints can be evaluated concurrently. When the computation deadlocks, a goal for the deadlock handler relation (Program 6.14) is evaluated. saggregate/4 scans Meta_Suspended and produces:

1. **Dont_Know**: a list of the meta-level terms representing suspended goals for don't-know relations which non-deterministically produce alternative sets of constraints,
2. **Required**: a list of the meta-level terms representing the suspended required constraints,
3. **Non_Required**: a list of the constraint/2 terms representing the suspended preferential constraints.

consistent/1 checks the satisfiability of the required constraints and fails if inconsistent constraint goals are suspended. If the consistent goal succeeds, decide_on_behaviour/6 is spawned to decide on the mechanism for breaking deadlock based on the bindings of Dont_Know and Required. If Dont_Know is a
non-empty list of terms representing suspended goals for don't-know relations (clause 2), then a term from Dont_Know is selected so that the corresponding goal is non-deterministically reduced.

If Dont_Know is an empty list and there are suspended required constraints (clause 3), clp_solver/2 is invoked to apply specialized algorithms and optimization techniques for solving the suspended required constraints. The clp_solver/2 goal produces a new set of required constraints which are compared with the old ones in order to determine which of the old constraints should be removed and what are the constraints which should be added to the current conjunction of goals. Additionally, compare_preferential/4 checks the preferential constraints for any inconsistent constraints with the new required constraints in order to remove them.

Clause 4 in Program 6.14 is the main difference between the deadlock handler for usual CLP systems and the deadlock handler for an HCLP system. This clause handles the case when all the required constraints are solved.

select_strongest/2 accepts the list of preferential constraints, Non_Required, and selects a constraint(Strength, C) term whose Strength value denotes that C is the most preferred preferential constraint. After the successful evaluation of deadlock_handler/4, the reduction of constraint/2 will non-deterministically add the constraint C to the current conjunction of goals.

```prolog
mode deadlock_handler(? , ^ , ^ , ^).
deadlock_handler(Meta_Suspended, Optimal, Removed, New) <-
    saggregate(Meta_Suspended, Dont_Know, Required, Non_Required),
    consistent(Required) &
    decide_on_behaviour( Dont_Know, Required, Non_Required
                        Optimal, Removed, New).
    % cl1.

mode decide_on_behaviour(? , ? , ? , ^ , ^ , ^).
decide_on_behaviour( [Generator|Dont_Know], Required, Non_Required,
                     Optimal, [], []) <-
    select_optimal([Generator|Dont_Know], Optimal). % cl2.
decide_on_behaviour( [], Required, Non_Required, '$no_choice',
                     Removed, New) <- Required \== []:
    clp_solver(Required, NewRequired),
    compare_preferential(Non_Required,NewRequired, Removed, RTail),
```

- 141 -
The Pandora Deadlock Handler

\begin{verbatim}
compare(Required, NewRequired, RTail, New).
\quad \text{\% cl3.}
decide_on_behaviour([], [], Non_Required, Optimal, [], []) \leftarrow
\quad \text{\% cl4.}
select_strongest(Non_Required, Optimal).
\end{verbatim}


6.6 Conclusions

The deadlock phase in a Pandora computation provides a synchronization point in a stream and-parallel evaluation which could be exploited in many ways. The meta-level deadlock handler relation has been defined in this chapter by which the programmer can explicitly program the behaviour on deadlock. Executing the deadlock handler relation goal each time the computation deadlocks is time consuming and requires additional support in the underlying implementation. However, if properly programmed, the user-defined control over the computation may lead to an intelligent application-dependent problem-solving behaviour which can outweigh its cost.
Chapter 7

PANDA: A Prototype PANDora System

When designing a new programming language, an experimental implementation is usually required as soon as possible in order to test the programming capabilities of the language and explore potential applications. These experiments may also affect the future design of the language. PANDA: an experimental prototype system for Pandora, has been developed. It comprises a compiler for Pandora programs to Parlog procedures, together with a simple run-time system. PANDA is implemented in Parlog and runs on the Sequential Parlog Machine SPM (Foster et al, 1986). The main features of the system are explained in this chapter.

7.1 Pandora Programs for PANDA

A source program for PANDA may include two kinds of relation: don't-care relations and don't-know relations. The syntax of these relations differ from the syntax described and used throughout the thesis in the following respects:

(a) a don't-know procedure for a don't-know relation is preceded by a
dont-know declaration:

dont_know r/n.

where r is the relation name and n is its arity.

(b) in addition to its don't-know procedure, a don't-know relation is also defined by a don't-care procedure in the program.

In Chapter 3, it was explained that a compile-time determinacy analysis algorithm
analyses the heads of the clauses in a don't-know relation procedure as well as the primitive calls in the guards and produces a sequence of simple run-time tests by which a deterministic goal can be detected and executed in the and-parallel phase. Otherwise, the goal suspends on variable(s) until it becomes deterministic or the deadlock phase begins. Since PANDA does not yet include such an algorithm, a don't-care procedure should be defined by the programmer for each don't-know relation. The don't-care procedure for a don't-know relation must suspend the goal if it is non-deterministic. Ideally, the don't-care procedure should also: fail the goal if all clauses are non-candidates for it, and reduce it when all the clauses in a '.'-separated group of clauses are non-candidates except one and all the clauses in the previous groups are non-candidates.

For example, Program 7.1 illustrates the definition of the cell/3 don't-know relation in a Pandora program for PANDA. The decision graph resulting from the determinacy analysis of the don't-know procedure was previously presented in Figure 3.1 in Chapter 3. The don't-care procedure in Program 7.1 below is a direct translation of the graph7.

```
% A don't-know procedure
dont_know cell/3.
cell(on, _, _). % cl1.
cell(off, Chain, Chain). % cl2.

% A don't-care procedure
mode cell(? , ?, ?).
cell(on, _, _). % cl3.
cell(off, Left, Right) <-
   Left = Right. % cl4.
cell(X, _, _) <- X == on, X == off:
   fail. % cl5.
cell(X, Left, Right) <- Left == Right:
   On = on. % cl6.

Program 7.1. The definition of the cell/3 don't-know relation for PANDA.
```

As explained in Chapter 3, the determinacy test may not be complete. That is, it may not detect some deterministic goals for the relation and suspend them until the

7The four clauses in the procedure directly implement the first four exits in the graph while exits 5, 6, and 7 are implied by the suspension of some of the existing clauses.
deadlock phase. The incompleteness of the determinacy analysis algorithm does not affect the correctness of the program since the deterministic goal will be eventually selected for reduction in the deadlock phase (unless it is removed as a result of evaluating a deadlock_handler goal). The simplest don't-care procedure for a don't-know relation is the one which suspends any goal for the relation whether it is deterministic or non-deterministic. For instance, a don't-care procedure for the cell/3 relation which always suspends the cell goal would be defined as follows:

\[
\begin{align*}
\text{mode} & \quad \text{cell}(?, ?, ?). \\
\text{cell}(\text{On, Left, Right}) & \leftarrow \text{data}(\_): \text{true}.
\end{align*}
\]

where data/1 is a Parlog primitive (and hence a Pandora primitive) whose goal suspends until its argument is bound to a non-variable term and then succeeds.

Program 7.2 defines the flying_pan/3 don't-care relation. The program illustrates that there is no change in the syntax of a don't-care relation for the PANDA system from its syntax previously described in the thesis.

\[
\begin{align*}
\text{mode} & \quad \text{flying_pan}(?, ?, ^). \\
\text{flying_pan} & \leftarrow \text{produce_assignment} \& \text{extract_programme}(\text{By_flight}, \text{Flying_programme}).
\end{align*}
\]

Program 7.2. The definition of the flying_pan/3 don't-care relation for PANDA.

### 7.2 PANDA's Main Strategy

A Pandora computation differs from a Parlog computation in the way of handling deadlock. While deadlock is a run-time error in Parlog, it initiates a deadlock phase in a Pandora computation, after collecting all the suspended goals. PANDA is a system which compiles a Pandora program to an equivalent Parlog one and evaluates the query to the original program by initiating a Parlog computation. In order to collect the suspended goals when the computation deadlocks and starts a deadlock phase, the evaluation of each goal during the and-parallel phase is controlled using the Parlog control metacall call/3, whose mode is (?,^,?). A call
always succeeds. Provided the Control argument remains an uninstantiated variable, the metacall evaluates its first argument Goal (a term denoting a relation call) and binds its second argument Status to a list representing the status of the evaluation. When the evaluation of Goal is complete, Status will be bound to [success|_] or [failure|_] and the metacall itself will succeed. If Goal is suspended, Status will be bound to [deadlock|S1] and the metacall will suspend until the evaluation of Goal is resumed then the metacall will resume execution using S1 to echo the future status of the evaluation.

Each procedure in the source program to PANDA is compiled to a Parlog procedure such that, when the goal call(Goal, Status, C) reduces Goal using a candidate clause, Goal succeeds after returning the conjunction of goals in the body of the clause; the evaluation of the new goals can be then controlled by the Parlog metacall.

If the goal call(query, Status, Control) suspends after instantiating Status to [deadlock|S1], where Query is the query to the program, a list of terms representing the suspended goals in the current conjunction of goals is constructed. Then, all the suspended goals terminate successfully so that call(query, Status, Control) itself succeeds, and a deadlock phase is initiated. The mechanism for constructing the list of terms for the suspended goals and then terminating the evaluation of these goals is explained in Section 7.4.1.

If the computation deadlocks and deadlock_handler/4 is not defined in the source program, a term representing a suspended goal for a don't-know relation is selected and a choice point created for it. In each or-branch, a copy of the list of suspended goals is made and the copy of the selected goal is unified with the head of a clause in its don't-know procedure. Then, the goal's copy is replaced by the goals in the body of the clause and a new and-parallel phase is begun, evaluating the resulting conjunction of goals in the same way as before.

If deadlock_handler/4 is defined in the source program, its procedure is invoked in the deadlock phase and, based on the bindings of its output arguments, the deadlock is broken. deadlock_handler/4 is evaluated as a normal Parlog goal and does not require an additional machinery in the PANDA system. Therefore, the
source form of the procedures invoked by deadlock_handler/4 as well as the procedure of deadlock_handler/4 itself are added to the compiled form of the program.

7.3 The PANDA Compiler

The PANDA compiler compiles a Pandora program of the form defined in the previous section to a Parlog program and then invokes the SPM compiler to further compile the resulting program to object code that is executable by the SPM run-time system.

In order to compile a don't-care procedure (for either a don't-care relation or a don't-know relation) to a Parlog procedure that can be evaluated by PANDA, an extra output argument is added in the first argument position of the relation. For instance, if the mode declaration of the don't-care procedure is:

```prolog
mode r(?).
```

then the mode declaration of the corresponding Parlog procedure would be:

```prolog
mode r(? , ?).
```

For each clause in the original don't-care procedure, a Parlog clause is generated. The conjunction of guard goals in the original clause (if any) remains the same in the produced Parlog clause while the body of the Parlog clause is an empty conjunction of goals that can be represented by the call true. For each call to a user-defined relation in the body of the original clause, a new distinct variable is added in its first argument position and all the resulting body goals are collected in a list forming the first argument in the head of the produced Parlog clause.

For instance, Program 7.3 presents the compiled form of the cell/3 don't-care procedure. It should be noted that no extra argument is added to the calls 'fail' and '/2' since they are primitive calls and not user-defined relation calls.
mode cell(^, ?, ?, ?).
cell([], on, _, _).
cell([Left = Right], off, Left, Right).
cell([fail], X, _, _) <- X == on, X == off: true.
cell([(X = on)], X, Left, Right) <- Left == Right: true.

Program 7.3. The compiled form of the don't-care procedure for the cell/3 relation.

From the above description of the compilation process, it can be noted that a sequential and/or a parallel conjunction in the guard of the original clause is not altered in the corresponding Parlog clause. However, a sequential conjunction in the body of a clause is compiled to the term:

\[ \text{seq(LeftConj, RightConj)} \]

where LeftConj is the list of goals that are to the left of '&', and RightConj is the list of goals to the right of '&'. For instance, Program 7.4 presents the compiled form of the don't-care procedure for the flying_pan relation.

mode flying_pan(^, ?, ?, ^).
flying_pan([seq([produce_assignment(X1, F, A, By_flight)],
               [extract_programme(X2, By_flight, Programme)]),
            F, A, Programme).

Program 7.4. The compiled form of the don't-care procedure for the flying_pan relation.

The compilation of a don't-know procedure consists of two phases. During the first phase, the guard operator in the clauses (if any) is replaced by the parallel conjunction operator '\('. Hence, all the calls in the right hand side of a clause are treated as body calls.

The second phase produces the final compiled form of the procedure. The arity in the procedure's dont-know declaration is incremented by one and, for each clause in the don't-know procedure, a corresponding clause is generated in the same manner as generating a clause in the compiled form of a don't-care procedure. Namely:

1. each sequential conjunction in the right hand side of the clause is replaced by the term seq(LeftConj, RightConj).
2. an extra argument is added to the first argument position in the head of the clause and is bound to a list of the goals in the right hand side of the original clause, after adding to each user-defined relation goal a new distinct variable in its first argument position.

3. the body of the resulting clause is empty.

Suppose that the first clause in the don't-know procedure of the cell relation is:

\[ \text{cell}(X, \_\_\_): X = \text{on}: \text{true}. \]

Then, the compiled form of the clause is:

\[ \text{cell}([(X = \text{on}), \text{true}], X, \_\_, \_\_). \]

Finally, if deadlock_handler/4 is defined in the source program, then the don't-care procedures in the source program are added to the produced Parlog program.

### 7.4 The PANDA Run-Time System

The run-time system can be logically divided into three modules: the query-level module, the and-parallel module, and the deadlock-handling module. In order to evaluate a Pandora query, the *query-level module* is invoked by:

\[ \text{run_query}(\text{Query}, \text{Query}_\text{Vars}, \text{Bindings}) \]

where *Query* is the query conjunction to the Pandora program, *Query_Vars* is a term having all the variables in the query for which the user would like to know the resulting bindings, and *Bindings* is an output argument which will be bound to the resulting bindings of *Query_Vars* (or a copy of it). For instance, a query to the n-queens program with *n* equals 8 is:

\[ \text{run_query}(\text{n}_{\text{queens}}(8, \text{Queens}), \text{Queens}, \text{Bindings}) \]

When invoked, the query-level module compiles the query, *Query*, by adding an extra argument to each user-defined relation goal in its first argument position and compiles sequential conjunctions to the form described in the previous section. Then, the compiled form of the query is executed by invoking the other modules. *run_query/3* is defined in Program 7.5; *run det/2* is the top-level relation in the
and-parallel module, while run_non_det/3 is the top-level relation in the deadlock-handling module.

\[
\text{mode run_query(?}, ?, ^). \\
\text{run_query(\text{Query, Query_Vars, Bindings})<} \\
\text{comp_query(\text{Query, CompQuery}),} \\
\text{run_det(CompQuery, Suspended_Goals),} \\
\text{run_non_det(Suspended_Goals, Query_Vars, Bindings).}
\]

Program 7.5. The top level relation calls of the PANDA run-time system.

### 7.4.1 The And-Parallel Module

The and-parallel module evaluates the goals in the and-parallel phase. The top-level relation of the module is \text{run_det/2}, whose mode declaration is:

\[
\text{mode run_det(GoalStatement?, Suspended_Goals^).}
\]

A call to \text{run_det/2} fails if the evaluation of any goal in the current conjunction of goals fails. Otherwise, it terminates successfully after producing \text{Suspended_Goals}: a list of all the suspended goals. If the evaluation of the current conjunction of goals terminates successfully, \text{Suspended_Goals} will be bound to an empty list.

This section describes the behaviour of PANDA in the and-parallel phase. It starts by illustrating the evaluation of each individual goal in the current conjunction of goals, followed by an explanation of how the entire computation is controlled during that phase.

### 7.4.1.1 Reducing Goals in the And-Parallel Phase

A goal is reduced in the and-parallel phase using its don't-care procedure. An attempt to reduce a goal would result in one of the following:

- (a) success: there is at least one candidate clause for the goal. The goal is then reduced by the selected candidate clause.
- (b) failure: all clauses in the don't-care procedure are non-candidates.
- (c) suspension: there is no candidate clause for the goal but at least one clause is suspended.
For each goal \( G \) in the current conjunction of goals, the following goal is invoked:
\[
\text{execute\_goal}(G, \text{Sus\_Head}, \text{Sus\_Tail}, \text{Enq})
\]

Ignoring the remaining arguments for the time being, \text{execute\_goal} reduces \( G \) by the clause:
\[
\text{execute\_goal}(G, \text{Sus\_Head}, \text{Sus\_Tail}, \text{Enq}) \leftarrow \text{call}(G, S, C): \text{determined}, \text{Co}, \text{Body}),
\]
\[
\text{execute\_conjunct}(\text{Body}, \text{Sus\_Head}, \text{Sus\_Tail}, \text{Enq}).
\]

The input head unification and the guards of the clauses in the don't-care procedure of \( G \) are evaluated. If a candidate clause is found, \( G \) is reduced and \( \text{call}(G, S, C) \) succeeds after binding \( S \) to \([\text{success}\_\_]\). Then, \text{determine}/3 (defined in Program 7.6 below) extracts the body of the selected candidate clause; this is represented by a list in the first argument of \( G \) (clause 2). As the goals in the body of the clause are extracted, they are evaluated in parallel with the other goals in the original conjunction of goals. If \( G \) is a primitive call, \text{determine}/3 (clause 1 in Program 7.6) returns an empty list of body goals.

It should be noted that \( \text{call}(G, S, C) \) is a safe guard goal in the clause of \text{execute\_goal} since it only evaluates the safe guard goals in the clauses of \( G \) and hence does not bind any variable in \( G \).

If all the clauses are non-candidates, \( S \) is bound to \([\text{failure}\_\_]\) which causes the failure of \text{determine}/3 and, consequently, of the current and-parallel phase. If no candidate clause is found but at least one clause is suspended, \( S \) is bound to \([\text{deadlock}\_S1]\) and \text{execute\_goal} suspends. When a suspended clause is reactivated, \( S1 \) will echo the future status of the evaluation.

\[
\text{mode determine}(?, ?, ^). \]
\[
\text{determine}([\text{success}\_\_], G, []) \leftarrow \text{primitive}(G): \quad \% \text{G is a primitive call.}
\]
\[
\text{true;}
\]
\[
\text{determine}([\text{success}\_\_], G, \text{Body}) \leftarrow \text{arg}(G, 1, \text{Body}); \quad \% \text{G is a user-defined relation call.}
\]
\[
\text{determine}([\text{Other}\_S1], G, \text{Body}) \leftarrow
\]

- 151 -
Other \( \Rightarrow \) failure: \% G is suspended and then re-activated.

determine(S1, G, Body).

**Program 7.6.** Controlling the evaluation of one goal in the and-parallel phase.

In order to be able to collect the suspended goals when the computation deadlocks, each goal in the current conjunction of goals is allocated a position in the Suspended_Goals list. Suppose, the current conjunction of goals is:

\[ G_1, G_2, ..., G_n \]

Then, the following conjunction is spawned:

\[
\begin{align*}
\text{execute_goal}(G_1, \text{Suspended}_\text{Goals}, SG_1, \text{Enq}), \\
\text{execute_goal}(G_2, SG_1, SG_2, \text{Enq}), \\
\vdots \\
\text{execute_goal}(G_n, SG_{n-1}, [], \text{Enq}).
\end{align*}
\]

The second and third arguments of a goal

\[
\text{execute_goal}(G, \text{Sus}_\text{Head}, \text{Sus}_\text{Tail}, \text{Enq})
\]

mark the beginning and end of G's position in the Suspended_Goals list. When G is successfully reduced, its positions in Suspended_Goals is inherited by the goals in the body of the selected clause. If the clause body is empty, this position is removed by unifying Sus_Head with Sus_Tail. As a consequence, if all the goals in the current conjunction of goals terminate successfully, Suspended_Goals will be bound to '[]'.

### 7.4.1.2 Controlling the Termination of an And-Parallel Phase

The top-level relation in the and-parallel module, the run_det relation, is defined in Program 7.7. It spawns two concurrent processes: call/3 and detect_deadlock/2.

call/3 evaluates execute_conjunct/4, while its status argument \( S \) is checked by the concurrent goal detect_deadlock. execute_conjunct is responsible for spawning the conjunction of execute_goal goals, described in the previous section. execute_conjunct succeeds if all execute_goal calls terminate successfully and fails if any execute_goal fails. In the former case, detect_deadlock succeeds (clause 2) causing the successful termination of the and-parallel phase, while in the latter case
it fails and hence the and-parallel phase terminates with failure.

If all the remaining execute_goal calls suspend, execute_conjunct deadlocks. Then, detect_deadlock instantiates its second argument to 'enquire' and terminates successfully (clause 3).

```
mode run_det(? , ^).
run_det( GoalStatement, Suspended_Goals )<- 
call( execute_conjunct( GoalStatement, Suspended_Goals, [], Enq), S, C),
detect_deadlock( S, Enq).

mode detect_deadlock(? , ^).
detect_deadlock([success|[], _].
detect_deadlock([deadlock|[], enquire).
```

Program 7.7. Controlling the evaluation in the and-parallel phase.

The second argument of detect_deadlock, Enq, is shared with execute_conjunct/4. This, in turn, is inherited by all the execute_goal calls as their last argument. The full definition of execute_goal is given in Program 7.8 below. When the computation deadlocks and Enq is bound to 'enquire', each execute_goal succeeds after adding its first argument to Suspended_Goals (clause 2).

```
mode execute_goal(? , ^ , ?, ?).
execute_goal( G, Sus_H, Sus_T, Enq )<- call(G, S, C):
determine( S, G, Body),
execute_conjunct( Body, Sus_H, Sus_T, Enq).
execute_goal( G, [G|Suspends], Suspended, enquire).
```

Program 7.8. execute_goal/4 relation in the and-parallel module.

### 7.4.1.3 Evaluating Sequential Conjunctions

A sequential conjunction of goals is represented by the term:

```
seq( LeftConj, RightConj)
```

During the and-parallel phase, the goals in LeftConj are evaluated in parallel using the corresponding don't-care procedures. This is handled by the following clause
in the procedure of `execute_conjunct/4'.

\[\text{execute_conjunct(} [\text{seq(LeftConj, RightConj)}|Gs],\]
\[\text{Sus\_Head, Sus\_Tail, Enq)}<-\]
\[\text{execute_conjunct(LeftConj, SusLeftConj, [], Enq),}\]
\[\text{seq\_right(} \text{SusLeftConj, RightConj, Sus\_Head, Sus\_T1, Enq),}\]
\[\text{execute_conjunct(Gs, Sus\_T1, Sus\_Tail, Enq).}\]

`seq\_right/5` (defined in Program 7.9 below) ensures that the evaluation of `RightConj` is started after the evaluation of `LeftConj` terminates successfully (clause 1). If the computation deadlocks before `LeftConj` terminates (clause 2), `seq\_right` generates the term: `seq(SusLeftConj, RightConj)`, where `SusLeftConj` is the list of goals, which have remained from the evaluation of `LeftConj`.

```
mode seq\_right(? , ?, ^, ?, ?).
seq\_right([], RightConj, Sus\_H, Sus\_T, Enq)<-
  execute\_conjunct(RightConj, Sus\_H, Sus\_T, Enq); % cl1.
seq\_right(SusLeftConj, RightConj, [seq(SusLeftConj, RightConj)|T], T, _).
  execute\_conjunct(Gs, Sus\_T1, Sus\_Tail, Enq).
```

**Program 7.9.** Controlling the evaluation of a sequential conjunction in the and-parallel phase.

### 7.4.2 The Deadlock-Handling Module

The *deadlock-handling module* implements the deadlock phase in a Pandora evaluation. Don't-know non-determinism is implemented by applying the clauses in the don't-know procedure of a goal in their textual order, and backtracking in case of failure. As will be illustrated in Section 7.4.2.3, the module can be easily modified to apply the alternative clauses for the goal simultaneously, providing an or-parallel evaluation for the various branches of a choice point.

The module is activated when the and-parallel phase terminates successfully. That is, when the evaluation of the current conjunction of goals either succeeds or deadlocks. The top-level relation of the module is `run\_non\_det/3`, whose mode declaration is:

```
mode run\_non\_det(Suspended\_Goals?, Query\_Vars?, Bindings^).
```
Suspended_Goals is the list of suspended goals in the current conjunction of goals. Query_Vars is the term having the variables in Suspended_Goals whose bindings from successful computation(s) should be returned to the user in Bindings.

7.4.2.1 Implementing the Deadlock Breaking Capabilities of Pandora

The run_non_det relation is defined in Program 7.10. It includes three clauses which are separated by sequential clause search operators. The first clause terminates the goal successfully when the current conjunction of goals is empty. The substitution which results from this successful computation is returned by unifying the second and third arguments of the run_non_det goal. The second clause handles the case when deadlock_handler/4 is defined in the program, while the third clause implements the default deadlock breaking mechanism when deadlock_handler/4 is not defined.

If deadlock_handler/4 is not defined in the program (clause 3), a suspended goal for a don't-know relation is selected by the select_dont_know goal, and a list of the clauses in its don't-know procedure is generated by clauses/2. Then, the selected goal is non-deterministically reduced by the choicepoint/7 goal. The details of the non-deterministic reduction of the selected goal is explained in the following sections.

If deadlock_handler/4 is defined in the program (clause 2), its procedure is invoked with the input argument Meta_Suspended: a ground copy of the list of suspended goals. Meta_Suspended is generated by the freeze/2 call. freeze/2 is a Parlog primitive, whose mode of use is (?,^). A call to freeze/2 generates a ground copy of its input argument. After the successful termination of deadlock_handler, a goal for decide_on_behav/6 is spawned to break the deadlock of the computation depending on the bindings of the output arguments of deadlock_handler/4.
mode run_non_det(?f, ?, ^).
run_non_det([], Solution, Solution); % cl1.
run_non_det(Suspended_Goals, Vars, Solution)<-
defined('deadlock_handler/4'):
  freeze((Suspended_Goals, Vars), (Meta_Suspended, Meta_Vars))&
  deadlock_handler(Meta_Suspended, Optimal, Removed, New)&
  decide_on_behav(Optimal, Removed, New, Meta_Suspended,
                  Meta_Vars, Solution); % cl2.
run_non_det(Suspended_Goals, Vars, Solution)<-
  select_dont_know(Suspended_Goals, Goal, Remaining, BodyH, BodyT),
  clauses(Goal, Cs)&
  choicepoint(Cs, Goal, BodyH, BodyT, Remaining, Vars, Solution). % cl3.

Program 7.10. The PANDA deadlock handler.

The procedure of decide_on_behav/6 is illustrated in Program 7.11. Clauses 1 and
4 in the program act as safe guards against semantic errors in the definition of the
deadlock handler relation. If neither a forking nor a non-forking deadlock breaking
mechanism is selected by the deadlock_handler goal (clause 1), a semantic error is
reported and the computation aborts. Selecting both options for breaking deadlock
is another semantic error and is detected by clause 4.

If a suspended goal for a don't-know relation is selected for forking (clause 2), then
the selected goal is non-deterministically reduced using its don't-know procedure.
melt/2 is a Parlog primitive whose mode of use is (?^). A call to melt/2 accepts
a term that was previously produced by a freeze/2 call and returns the same term
after replacing the ground representations of its variables by unbound variables. In
other words, a call to melt/2 has the opposite effect of a call to freeze/2.

remove_dont_know in clause 2 removes the selected don't-know relation goal from
the list of suspended goals since it will no longer be suspended and will be non-
deterministically reduced.

Alternatively, the deadlock_handler goal may select ground terms from its input
argument whose corresponding goals should be removed from the current
conjunction. The goal may also specify new goals to be added to the conjunction
(clause 3). After deadlock_handler succeeds, remove_goals/3 is spawned to
produce Meta_Remaining: a list of ground terms which represent the remaining
goals in the current conjunction after removing the goals selected by deadlock_handler. If new goals should be added to the current conjunction, these are appended to Meta_Remaining. Then, the resulting conjunction of goals is evaluated in a new and-parallel phase.

```

decide_on_behav('$no_choice' , [], [], _, _, _) <-
    write('ERROR in the deadlock handler:
    no deadlock breaking mechanism is selected')
    abort; % cl1.

decide_on_behav(Meta_Opt , [], [], Meta_Conj, Meta_Vars, Sol) <-
    melt((Meta_Opt, Meta_Conj, Meta_Vars),
    (Optimal, Conj, Vars))
    remove_dont_know(Conj, Optimal, Remaining, BodyH, BodyT),
    clauses(Optimal, Cs),
    choicepoint(Cs, Optimal, BodyH, BodyT, Remaining, Vars, Sol). % cl2.

decide_on_behav('$no_choice', Removed, Meta_New,
    Meta_Conj, Meta_Vars, Solution) <-
    remove_goals(Removed, Meta_Conj, Meta_Remaining),
    append( Meta_New, Meta_Remaining, New_Meta_Conj)&
    melt((New_Meta_Conj, Meta_Vars), (New_Conj, Vars))&
    run_det(New_Conj, Suspended_Goals),
    run_non_det(Suspended_Goals, Vars, Solution); % cl3.

decide_on_behav(_, _, _, _, _, _) <-
    write('ERROR in the deadlock handler:
    a non-deterministic goal selected for forking while
    goals selected for removal/new goals to be added')
    abort. % cl4.
```

Program 7.11. Selecting the appropriate deadlock breaking mechanism in the deadlock phase.

### 7.4.2.2 A Sequential Implementation of Or-Nondeterminism

In order to non-deterministically reduce a don't-know relation goal G, the clauses in its don't-know procedure are tried according to their textual order, starting from the
topmost one until the evaluation succeeds (clause 2 in Program 7.12).

An attempt to reduce G using a clause C (clause 3 in Program 7.12) first copies the current conjunction of goals (CGStatement). Then, the run_one_copy/7 goal unifies the copy of G (CG) with the head of C (clause 4). If unification succeeds, the goal is reduced to the goals in the body of the clause; these are extracted from the first argument of the goal. The resulting conjunction of goals is then evaluated, starting a new and-parallel phase. If unification fails, or subsequent failure takes place, run_one_copy fails. Then, a fresh copy of the conjunction of goals is produced and the head of another clause in the don't-know procedure of G is unified with the new copy of G. If all attempts to reduce G fail, choices/8 fails.

Program 7.12. A sequential non-deterministic evaluation of a don't-know relation goal.

In general, the goals in the body of the clause can be added anywhere in the current
conjunction of goals, since the order of goals in a parallel conjunction is irrelevant. However, suppose that the reduced goal is to the left of a sequential conjunction operator. Then, the goals in the body of the clause should be also placed to the left of the sequential conjunction operator. For this reason, CBodyH and CBodyT are input arguments of run_one_copy which mark the beginning and end positions in the current conjunction of goals, where the body goals of the clause should be added.

7.4.2.3 A Parallel Implementation of Or-Nondeterminism

The deadlock-handling module can be easily changed to provide a parallel evaluation of the alternative branches of a choice point. Program 7.13 redefines the choicepoint relation after modifying it to run in parallel all the attempts of reducing a goal using the clauses in its don't-know procedure. The substitutions that are produced by successful computations are collected in a difference list whose head and tail are the last arguments of choicepoint/8. If an attempt to reduce a goal with one of its clauses fails, the detect_failure/3 goal (clause 3) will unify the head and tail of the part of the "solutions list" that was assigned to that attempt, i.e. no solutions will be produced by that attempt.

```prolog
choicepoint([], _, _, _, _, _, Solutions, Solutions).  % cl1.

choicepoint([C|Cs], G, BodyH, BodyT, GStatement, Vars, SolH, SolT)<-
    ( copy(  (G, BodyH, BodyT, GStatement, Vars),
      (CG, CBodyH, CBodyT, CGStatement, CVars))&
      call(run_one_copy(C, CG, CBodyH, CBodyT, CGStatement, CVars,
        SolH, SolH1), S, _)
    ),
    detect Failure(S, SolH, SolH1),
    choicepoint(Cs, G, BodyH, BodyT, GStatement, Vars, SolH1, SolT).  % cl2.

mode detect Failure(? , ^, ?).  % cl3.
detect Failure([failure|_], Solutions, Solutions);  % cl3.
```

- 159 -
The run_query and run_non_det relations are redefined in Program 7.14. The third argument of run_query/3, Solutions, is a list of all solutions to the query. This list is represented by a difference list when invoking the run_non_det procedure, with the tail of the list being the empty list. The first clause of run_non_det adds a solution to the head of the list after a successful computation. The remaining clauses are similar to the ones previously defined in Section 7.4.2.1 except for collecting a list of solutions instead of one solution. If all attempts to solve the query fail, Solutions will be bound to '[]', which will fail the query_success/1 goal and consequently the run_query goal.

```
mode run_query(? , ?, ^).
run_query(Query, Vars, Solutions) <-
    comp_query(Query, CompQuery),
    run_det(CompQuery, Suspended_Goals),
    run_non_det(Suspended_Goals, Vars, Solutions, []),
    query_success(Solutions).

mode query_success(?).
query_success([Sol|Solutions]).

mode run_non_det(? , ?, ^, ?).
run_non_det([], Solution, [Solution|SolTail], SolTail);
run_non_det(Suspended_Goals, Vars, SolH, SolT)<-
    defined('deadlock_handler/4');
    freeze((Suspended_Goals, Vars), (Meta_Suspended, Meta_Vars))&
    deadlock_handler(Meta_Suspended, Optimal, Removed, New)&
    decide_on_behav(Optimal, Removed, New, Meta_Suspended, 
        Meta_Vars, SolH, SolT);
run_non_det(Suspended_Goals, Vars, SolH, SolT)<-
    select_dont_know(Suspended_Goals, Goal, Remaining, BodyH, BodyT),
    clauses(Goal, Cs)&
    choicepoint(Cs, Goal, BodyH, BodyT, Remaining, Vars, SolH, SolT).
```

Chapter 8

The Pandora Abstract Machine

In Chapter 7, an experimental prototype system for Pandora was explained. For a more efficient implementation, PAM: an abstract machine for Pandora has been designed. It is based on the JAM abstract machine for committed-choice languages and modifies it to support "lazy" don't-know non-determinism and the Pandora deadlock breaking capabilities.

In order to provide a sufficient background for introducing PAM, overviews of the WAM and the JAM abstract machines are described in Sections 8.1 and 8.2, respectively. The WAM machine was designed for the execution of Prolog and, hence, supports don't-know non-determinism. The JAM machine, on the other hand, was designed for the execution of committed-choice languages. It, thus, supports committed-choice non-determinism and stream and-parallelism. The execution models of the WAM and the JAM are described, together with the main features of the machines themselves; more detailed descriptions can be found in other references.

8.1 An Overview of the Warren Abstract Machine (WAM)

The WAM (the Warren Abstract Machine) (Warren, 1983; Gabriel et al, 1985; Ait-Kaci, 1990) was designed in 1983 by David H.D. Warren for executing Prolog programs. Since then, the WAM machine has been widely used as the basis for implementing most Prolog-like logic programming languages.
8.1.1 The WAM Execution Model for Prolog Programs

The WAM adopts a goal-oriented execution model which provides the basic control mechanisms for executing Prolog programs. As explained in Chapter 2, the Prolog computation rule selects the leftmost goal for reduction and the select rule selects the topmost clause in the procedure corresponding to the goal. The head of the clause is then unified with the goal. There are two possible outcomes to unification:

   (a) unification fails: a failure event is then processed.
   (b) unification succeeds: execution advances to the next stage.

During unification, a set (possibly empty) of substitutions for the goal's variables is produced. If unification succeeds, a new conjunction of goals is generated from the original one after replacing the reduced goal with the body of the clause and applying the resulting substitutions on the new conjunction. The computation rule is then re-invoked to select the next leftmost goal.

Execution succeeds when there are no more goals to reduce; the substitutions to the variables in the original query forms the desired result.

If unification fails, a failure event is processed: the computation backtracks to its state before entering the failed clause and all substitutions produced during this unification are undone. The select rule is then re-invoked to select the next clause in the goal's procedure and apply it to the goal.

A choice point is created when a goal with several clauses in its procedure is selected for reduction. It contains a pointer to the code for the next clause to try should the current clause fail, plus all the information necessary to restore the state of computation to the time when it was created. A choice point remains while there are remaining unexplored clauses for that goal and is then removed. As an optimization, the choice point can be removed when the last clause for the goal is selected (termed the last clause optimization). Execution always backtracks to the most recently created choice point and selects another clause for its goal. If there is no choice point left to backtrack to, execution fails.
An environment is created when a clause with more than one goal in its body is selected. The environment comprises all the variables that are shared among the goals in the clause's body, as well as pointers to the position and environment of the next goal to be reduced after the successful reduction of the current goal. The environment remains while there are remaining body goals to be reduced, and is discarded otherwise. As an optimization, the environment can be discarded before reducing the last goal in the body of the clause (termed the last call optimization).

8.1.2 The WAM Abstract Machine

The WAM consists of a stack-based memory architecture and an instruction set. A Prolog program is compiled to a sequence of instructions from the WAM instruction set whose execution manipulates registers and data areas in the memory. The data areas are arranged in memory in the order illustrated in Figure 8.1. The functions of these data areas are:

- **Code Area** contains instructions (and other data) comprising the compiled form of the program.
- **Heap** contains structured terms and lists created during program execution.
- **Stack** contains two types of object: choice points and environments, which correspond to the choice points and environments in the execution model.
- **Trail** contains pointers to variables that have been bound during unification and that must be unbound on backtracking.
- **PDL** a push down list used for the general unification routine as a call stack.
In addition, the WAM specifies a number of registers, most of which point to the data areas. These are summarized below.

- **P** code pointer. It points to the next instruction to be executed.
- **CP** continuation code pointer. It points to the next goal to be executed after successfully reducing the current goal.
- **E** points to the current environment.
- **B** points to the current choice point.
- **TR** points to the top of trail.
- **H** points to the top of heap.
- **S** points to a sub-structure or a sub-list in the heap while it is being unified with another term.
- **A1, A2, ...** argument registers.

When a procedure is invoked, that is when a goal for its relation is to be reduced, **P** will be pointing to the instruction to be reduced in the procedure's compiled code, while the argument registers **A1, ..., An** will be loaded with the arguments of the goal. **CP** will be pointing to the next goal to be executed after a successful
execution of the current goal.

When a structured term or a list is being unified with some other term, the S register is used to traverse the different locations within the structure or the list. If a structure or a list is created during unification, it is pushed onto the heap. The top of heap is pointed at by the H register.

A data word in the abstract machine comprises a value field and a tag indicating its type. There are four main data types in WAM: references (pointers to other data words), structures, lists, and constants. An unbound variable is usually represented as a reference to itself.

During unification, a number of rules should be followed in order to avoid a dangling reference, i.e. a reference to a memory location that has been deallocated (for example by backtracking). These rules are:

1. when unifying two unbound variables, the most recently created variable is bound to reference the other variable. In WAM, the age of variables relative to each other can be determined by a simple address comparison of their locations in the memory.
2. no data word in the heap can reference a data word in the stack.
3. structures and lists are only created in the heap.

8.1.2.1 Environments in WAM

A variable in WAM is classified as temporary if it meets both of the following conditions:

(a) its first occurrence is in the clause head, or in a structure, or in the last goal in the clause body.

(b) it can occur in at most one goal in the body of the clause. If it occurs in the clause head, then it can only occur in the first body goal.

A variable is classified as permanent if it does not meet the above conditions.

For example, in clause 2 of the partition/4 relation in Program 8.1, X is a temporary variable while Xs, A, Lower, and Upper are all permanent variables.
The value of a temporary variable can be saved in an argument register which does not contain an argument of the current goal. Such argument registers are classified as temporary registers.

Since a permanent variable is shared among several goals in the clause, its value should be maintained until all of these goals are reduced, i.e. until it is no longer needed. If the value of a permanent variable is saved in a temporary register, it may be overwritten during the execution of the goals in the body of the clause. Therefore, an environment is pushed onto the stack when reducing a goal with a clause which has more than one goal in its body. It consists of data words for the permanent variables in the clause, together with a continuation which comprises the current values of the environment pointer (E) and the continuation code pointer (CP). The continuation specifies the clause and position (instruction within the clause) to return to on successful execution of the current goal.

An environment is discarded before entering the last goal in a clause body. If the arguments of the last goal are referencing a variable in the environment, that variable is moved to the heap (is said to be globalized) before discarding the environment so that to avoid a dangling reference.

For example, consider the query:

```
    g1, partition([2,4], 3, Y, Z), g2
```

An environment is pushed onto the stack when reducing `partition([2,4], 3, Y, Z)` using clause 2. It comprises of a pointer to the previous environment in the stack, a pointer to `g2`, and data words for Xs, A, Lower, and Upper. After successfully executing `2<=3`, Lower and Upper are globalized, the environment is discarded, and the goal `partition([4], 3, Lower, Upper)` is reduced.
8.1.2.2 Choice Points in WAM

A choice point in the abstract machine corresponds to a choice point in the execution model. It is created when a goal with a procedure of more than one clause is to be reduced. It contains a pointer to the code for the next clause to try should the current clause fail, plus all the information necessary to restore the state of computation to the time when the choice point was created; these are the values of the following registers:

- **CP** the continuation code pointer.
- **E** points to the current environment.
- **B** points to the current choice point, i.e. the one previous to this choice point.
- **TR** points to the top of trail.
- **H** points to the top of heap.
- **A1, A2, ...** argument registers.

While reducing a goal with a clause, unbound variables might get bound. The binding is called *conditional* if the variable was created before the clause activation, i.e. if the variable is in the goal's arguments. Otherwise, i.e. if the variable is local to the clause, the binding is called *unconditional*. A variable is trailed when it is bound conditionally.

When the clause fails, execution backtracks to the most recent choice point. Variables that are trailed since the creation of the choice point are reset to unbound and registers are returned to their values as saved in the choice point; then the next alternative clause is tried.

Resetting the registers when backtracking will remove any variables created during the execution of the failed clause. For this reason, there is no need to trail unconditionally bound variables.

If the clause being entered is the last clause to try for this goal, then the choice point is removed before entering the clause. Otherwise, the choice point's code pointer is updated to point to the clause following the one being entered.

For example, a choice point is created when reducing `partition([4,2], 3, Y, Z)`
8. The Pandora Abstract Machine

for Program 8.1. Unifying the goal with the head of clause 1 fails. Then, clause 2 is activated. An environment for clause 2 is created and the goal is unified with the head of the clause. The following bindings are produced:

conditional bindings = \{Y/[4|Tail]\}
unconditional bindings = \{X/4, Xs/[2], A/3, Lower/Tail, Upper/Z\}

It should be noted that when unifying Upper and Z, Upper will reference Z since it is the younger variable, while Z remains unbound.

An attempt to reduce the goal 4 < 3 then fails and the execution backtracks, resetting Y to unbound, removing the choice point, and activating clause 3.

If a choice point is created after creating an environment, then the environment is said to be protected by the choice point and can only be removed when the choice point is removed.

8.2 An Overview of Jim Crammond's Abstract Machine (JAM)

The JAM (Jim Crammond's Abstract Machine) has been designed for executing committed-choice languages. The first design of the machine was based on an "and-or tree" model of execution (Crammond, 1988). Recently, the JAM has been implemented by a C-emulator for running Parlog programs (Crammond, 1990). The implementation is based on an extended form of an "and-tree" model, which is described below.

8.2.1 The JAM Execution Model for Parlog Programs

The JAM adopts a process-oriented execution model. A process is responsible for finding a candidate clause for a goal and then creating further processes for the goals in the body of the selected clause. If all the clauses in the procedure of the goal are non-candidates, the process terminates with failure.

The following description of the execution model only considers programs with
flat guards. The execution of deep guards will not be explained since it is irrelevant to the Pandora abstract machine.

A process starts execution by trying input unification and the sequences of tests in the guard of the first clause in the procedure of its goal. This can have three outcomes:

(a) unification or a guard test fails: the goal process then tries next clause.

(b) unification or a guard test suspends: the process saves the variable which causes the suspension in a suspension set and tries next clause.

(c) unification and all guard tests succeed: the process commits to this clause and advances to the next stage.

If there are no clauses left to try, but there are some variables saved in the suspension set, then the process suspends on these variables. Subsequently, when any one of these variables gets bound, the process is resumes execution starting from the first clause again.

If there is no variable to suspend on, i.e. all the clauses have failed, then the goal process fails unless the last tried clause is terminated with a sequential clause separator ';', at which case only the current '.'-separated group of clauses has failed and the process starts executing the following group in the same way.

Once a process has committed to a clause, it performs any output unification; if that fails then the process terminates with failure. Otherwise, the process creates new processes to execute the remaining body goals.

Suppose the body contains parallel conjunctions separated by sequential conjunction operators, for example:

\[ H \leftarrow B_1, B_2 \& B_3, B_4 \& B_5, B_6, B_7. \]

then the process creates child processes for the goals in the first parallel conjunction and suspends waiting for all of its children to terminate. If they all succeed, the process resumes to execute the next conjunction. If one child terminates with failure, the parent also fails and kills any remaining children.
For the last parallel conjunction, a tail recursion optimization can be employed: the processes that are created for these goals are promoted to be sibling processes to the current one, except for the last body goal which the process executes itself. Hence, the parent process for the current one waits for these new children to terminate as well as the children it created itself.

It follows from the tail recursion optimization that, if the body consists of only one parallel conjunction, then this process will create a sibling process for each body goal, except for the last one which it will execute itself. Consequently, in programs which do not use sequential conjunction operators, all the processes are created as siblings to the root process that initiated execution of the top-level query.

Execution succeeds if all the processes terminate with success. If a process fails, execution fails.

8.2.2 The JAM Abstract Machine

The JAM has been designed and implemented to run on a shared-memory multiprocessor architecture. Unlike the WAM machine, several goals may be executed simultaneously on different processors. Each processor has its own set of data structures which are accessed by the other processors. These data structures are:

- **Process Stack**: contains process structures (which correspond to the processes in the execution model) and is managed by a free list management scheme.
- **Argument Stack**: contains two types of object: goal arguments, and environments.
- **Heap**: contains structured terms, lists, and variables that are created during program execution.
- **Run Queue**: contains runnable processes that are waiting for idle processors to execute them.
- **Code Area**: same role as in WAM.

In addition, each processor has a number of local registers, most of which point to the processor's own data areas. These are introduced throughout the section.
The code area contains the instructions (and other data) comprising the compiled form of the program. It can either be shared by all processors or, alternatively, it can be copied in the private memory of each processor.

The heap contains structured terms, lists, and variables that are created during program execution. When one of these data objects is created, it is placed on top of the local heap which is pointed to by an H register for the processor.

The organization of the shared memory is illustrated in Figure 8.2. It is divided by data structures and each data structure is then divided into blocks; one for each processor.

Fig 8.2. Data areas with process structures in the JAM.

### 8.2.2.1 The Run Queue

When processes are created, they are in runnable state. These processes are placed on a run queue, waiting to be executed by idle processors. Each processor has its own run queue to which it adds new processes that are spawned from the process that is currently running. Alternatively, the running process is added to the run queue and the processor executes a newly created process. Processes are also added to the run queue when they are re-activated after being suspended on variable(s) or suspended on children.

When a processor becomes idle, it takes a runnable process from its own run
queue and executes it, unless its run queue is empty in which case it searches for work in other processors run queues.

Each processor has two registers pointing to its own run queue: \textbf{QF}, which points to the front of the queue, and \textbf{QB}, which points to the back of the queue. Processes are added to the back of the queue by the owner processor only, while they can be taken off the front of the queue by any processor. Both \textbf{QF} and \textbf{QB} must be accessible to all processors (to test for an empty queue) but the significant advantage of the queue is that locking is only required when taking a process from the queue and not when adding a process to it; only the owner processor can modify the \textbf{QB} pointer.

There are two situations where a processor can avoid accessing the run queue(s) to find work:

1. \textit{suspending the current process on children}  
The processor can execute the last child after placing other children on the run queue and suspending the parent.

2. \textit{terminating the last child}  
When the last child of a process terminates successfully, then the processor can resume the execution of the parent directly, rather than adding it to the run queue and searching for work.

\subsection*{8.2.2.2 The Process Stack}

The \textit{process stack} is a stack of process structures. A \textit{process structure} is a fixed size data structure that contains sufficient information for a (goal) process to be executed (see Section 8.2.2.4 below).

Each processor manages its process stack by a free list management scheme. The \textit{free list} is a linked list of process structures which contain information for (goal) processes whose evaluation terminated. Hence, these process structures can be reused for new (goal) processes. The processor has two registers: \textbf{PP} pointing to the top of stack, and \textbf{PF} pointing to the first process in the free list; it is then continued through the \textbf{Link} pointers in the free process structures.

When a new process is requested, if \textbf{PF} is a null pointer a new process is pushed
onto the stack, otherwise the process pointed to by $PF$ is returned after $PF$ is assigned the value of the Link pointer of that process. When a process structure is released, it is prepended to the free list (regardless of which process stack the process structure belongs to), unless it is on top of the stack in which case it is popped off. In addition, each processor has a PS register that points to the process structure of the process it is currently executing.

### 8.2.2.3 The Argument Stack

The argument stack contains goal arguments and environments (described in Sections 8.2.2.5 and 8.2.2.6 respectively), i.e. the variable sized parts of a process. The argument stack is a stack from the point of view of allocation; all new items are pushed on top of the stack, which is pointed to by an SP register. However, it is a stack that can contain "holes" so that items can be deallocated even though they may not be on top of the stack. Holes are created by marking the cells being deallocated as free cells, for example by setting them to the value 0. Whenever, cells on top of the stack are popped off, a check is made to see if this has exposed a hole, in which case the hole is also popped off the stack.

A processor can deallocate items from another processor's argument stack. This does not require any synchronisation; a hole is simply created regardless of whether or not the cells are on top of another processor's stack.

### 8.2.2.4 The Process Structure

The process structure is a data structure that contains sufficient information for a process to be executed when it is runnable and, to communicate its outcome to its parent when it terminates. It consists of the following fields:

- **Parent** a pointer to the parent's process structure.
- **Code-Pointer** it initially points to the first instruction in the procedure of the process' goal. When a ';' is reached, it is set to the first instruction in the clause after ';'. Hence, if the process subsequently suspends, it will resume from the clause after ';'.
- **Reference-Count** number of active child processes. It is used by child processes to determine when to re-activate this process.
8. The Pandora Abstract Machine

- **Arguments**
  - a pointer to the arguments of the process on the argument stack.
- **Argument-Count**
  - number of arguments for this process.
- **Environment**
  - a pointer to the process environment that is saved on the argument stack.
- **Link**
  - if the process is suspending on variable, Link points to the next process structure suspending on the same variable. It may also point to the next process structure in the free list as previously explained.

In general, fields in the process structure are only updated when a process is running and by the processor that is executing it. The exception to this is the **Reference-Count** field which can also be modified by child processes as will be later illustrated; updates to this field require exclusive access to ensure correct behaviour.

### 8.2.2.5 Goal Arguments in JAM

Arguments for a (goal) process are stored on the argument stack while:

(a) the process is in the run queue waiting to be executed by an idle processor.
(b) the process is suspended on variable(s).

When a process is created and placed in the run queue, its arguments are stored on the argument stack. The process structure’s **Arguments** field points to the first argument and **Argument-Count** gives the number of arguments saved. The arguments are then loaded into the processor’s argument registers $A_1, \ldots, A_n$, when the process begins execution.

In the case of the last goal in a clause body, and similarly for the last child process created in a sequential conjunction, the processor does not add the process to the run queue but rather directly executes it. Hence, the goal arguments are placed in the argument registers and are not stored on the argument stack.

During input unification and execution of guards, the compiler ensures that the argument registers are not modified, other than being dereferenced. If a process is
about to suspend on variable(s), the processor checks the **Arguments** pointer to see whether the arguments have already been saved and, if not, it pushes the values of the argument registers on the stack.

The arguments are discarded from the argument stack (if they have been previously saved) when the process commits to a clause.

### 8.2.2.6 Environments in JAM

The definition of a temporary variable in the JAM is derived from its definition in the WAM after replacing the concept of a "goal" with a "parallel conjunction of goals". Thus, the definition of a **temporary** variable becomes:

- (a) its first occurrence is in the clause head, or in a structure, or in the last parallel conjunction of goals in the clause body.
- (b) it can occur in at most one parallel conjunction of body goals. If it occurs in the clause head, then it can only occur in the first parallel conjunction of goals in the clause body.

A **permanent** variable is one that does not satisfy the above conditions.

For example, in clause 2 of the `qsort/3` relation in Program 8.2, `Xs` and `Upper` are temporary variables, while `X`, `Sorted`, `Tail`, `L1`, and `L2` are permanent variables.

```prolog
mode qsort(?, ^, ?).
qsort([], Sorted, Sorted). % cl1.
qsort([X|Xs], Sorted, Tail) <-
    partition(X, Xs, L1, L2)&
    qsort(L1, Sorted, [X|Upper]),
    qsort(L2, Upper, Tail). % cl2.
```

**Program 8.2.** A Parlog quick sort of a list.

An **environment** is saved on the argument stack when a clause with a sequential conjunction in the body is entered. The environment holds the values of permanent variables, i.e. variables that are needed when the process resumes execution after being suspended on children.

The process structure's **Environment** pointer is initially set to null, but points to the environment on the argument stack after being saved. The size of the
environment is stored in the environment itself, on the argument stack, rather than in the process structure. The current environment is also pointed to by the processor's E register. Environments are deallocated before entering the last parallel conjunction of goals in the clause body (*last call optimization*), or as a result of the clause failing.

One significant difference between the JAM environments and environments in Prolog implementations is that unbound variables are not stored in the JAM environments, only references to their locations in the heap. This loses the advantage of avoiding placing some variables on the heap but, in practice, the number is few since any variable that occur in the last parallel conjunction in the body of a clause would need to be globalized anyway. Moreover, suppose that variables are saved in the JAM environments and two environment variables are unified. Then, the most recently created one should reference the older one to avoid a dangling reference when the newer environment is discarded. In Prolog, the variables' ages are determined by address comparison, but this is more difficult to determine where there are several argument stacks.

### 8.2.2.7 The Variable Table

Each processor has a *variable table* in its private memory that implements the variable set in the execution model. It stores variables that the process may suspend on. When execution of a clause cannot continue because a variable is not bound, the variable is checked against existing entries and added to the table if not present. The NV register holds the current number of entries in the table; it is set to 0 when a process is *scheduled*, i.e. is to be executed by the processor, and is reset on each commit.

When no candidate clause is found for the goal or a sequential clause separator is encountered, and the NV register is greater than 0, the process suspends and the processor schedules another process in the run queue.
8.2.2.8 Suspending on Variable(s)

A process suspends on an unbound variable by adding itself to the variable's suspension list: a linked list which starts at the variable itself. The variable's tag indicates that it is unbound while its value is a pointer to the most recent process to be suspended on that variable. The suspension list is continued by the Link field in the process structure which (if non-null) points to the next process suspended on that variable. This operation requires exclusive access to the variable in the same way as assigning a value to the variable.

Waking up processes is the converse of this procedure. On binding a variable, the old value field (if non-null) is taken to be a suspension list and the processes on it are added to the local run queue. Only the retrieval of the suspension list (and binding of the variable) need to be done atomically; no locking is required when actually traversing the suspension list waking up processes.

When suspending on more than one variable (i.e. NV>1), there is the problem of ensuring that only one of the variables actually wakes the process and thereafter other variables no longer refer to the process, which may commit and be deallocated before those variables are bound.

One solution to this problem was first proposed in (Houri and Shapiro, 1986), which extends the concept of a suspension list to a hybrid suspension list. When a process suspends on multiple variables, an indirect pointer to the process is created. It is called a hanger and is saved in the heap. Suspension notes are also created and added to the hybrid suspension lists of the variables on which the process is to suspend. A suspension note is a two word cell on the argument stack consisting of a pointer to the hanger of the process and a pointer to the next member (suspension note or process structure) of the variable's hybrid suspension list.

Waking a process suspending on multiple variables involves locking its hanger, adding the process to the run queue, then setting the hanger to a null pointer before releasing the lock. As other variables are bound which access the hanger, they will simply ignore it since it is null and thus will not re-execute the process. The suspension note in the woken list is, then, deallocated.
Combining the efficient suspension mechanism on one variable with the more complex mechanism of suspending on multiple variables requires the "wake procedure" to determine (by an address comparison) whether the current element in the hybrid suspension list is on a process stack or a heap. An element on the process stack indicates a process suspended on one variable only, which can be woken directly; otherwise an indirect pointer must be locked and accessed. Figure 8.3 illustrates an example of such a hybrid suspension list. In the remaining of the chapter, the term "suspension list" will be also used to denote a hybrid suspension list.

8.2.2.9 Suspending on Children

When a process is created, the Reference-Count field in the process structure is set to 1. Reference-Count is incremented whenever the process creates or inherits a new child process and, is decremented when a child process terminates or when the process becomes suspended on children.

When a process terminates, the processor decrements Reference-Count of the parent and checks to see if the count is 0. If so, then it had the last reference to the parent and must therefore resume its execution, "waking" it up.

Changes to the Reference-Count field of a process requires exclusive access except when the count is 1, at which case there is only one reference to the process.
8.2.2.10 Scheduling

After the initialization of the JAM, each processor will be in one of three states: execute state, search state, and deadlock state. The state of a processor is denoted by its status flag.

A processor is in execute state as long as it is executing a process. The processor enters the JAM "scheduling algorithm" and changes its state to search state if it runs out of work or else if an event is signalled.

Events are occasions when all processors must synchronize and execute specific functions. Examples of events are: interrupts, garbage collection, and termination. If any process in the process tree fails or the root process succeeds, a termination event is signalled so that all processors terminate.

A processor checks an event flag on each process reduction, that is a global register which is accessed by any processor. If the flag is set, the processor queues the current process on its own run queue and invokes the scheduling algorithm which, in turn, enters the event handler and executes the appropriate event(s).

If a running process terminates (and the parent cannot be continued) or suspends on variable(s), the processor enters the scheduling algorithm to search for work. It first examines its own run queue. If it is empty, the processor searches for work in other processors' run queues. If a process is found, it is loaded and the processor exits from the scheduling algorithm to execute the process, after changing its status back to execute state.

In order to take work from a run queue, the processor first examines the queue pointers to check if the queue is empty. If non-empty, a lock is set which prevents other processors from stealing a process. Once locked, the test for empty queue is repeated in case another processor has stolen the last process before the lock was obtained. Then, a process is taken from the queue and the lock is released.
If all run queues are empty, deadlock should be detected. The "deadlock detection algorithm" distinguishes one processor as being the master processor while the other processors as slaves. On failing to find work, the master enters a third processor state: deadlock state, and then waits for each slave to follow suit. If all processors enter deadlock state, then deadlock has occurred. Otherwise, if one of the slaves enters execute state instead, then the master re-enters search state and waits for all the slaves in deadlock state to undeadlock before proceeding to rescan the remote run queues.

Slave processors cannot enter deadlock state unless the master has done so. Therefore, if the master is not deadlocked, the slave makes another scan of the remote run queues. Otherwise, the slave enters deadlock state and waits for the master's state to change.

When deadlock occurs, the master processor creates a process to invoke a predefined "$deadlock" goal; thus it leaves the deadlock state and, so, other processors re-enter the search state. Figure 8.4 illustrates the scheduling algorithm with events.

Fig 8.4. The JAM scheduling algorithm.
8. The Pandora Abstract Machine

In this section, we introduce the PAM (the Pandora Abstract Machine) for executing Pandora programs on a shared memory multi-processor architecture. An earlier design of the machine which considers deep guards was described in (Bahgat, 1990). The PAM execution model is first described, followed by an illustration of the data structures and operations of the abstract machine.

8.3 The Pandora Abstract Machine

8.3.1 The PAM Execution Model for Pandora Programs

The PAM adopts a process-oriented execution model with two kinds of process: don't-care processes, responsible for reducing don't-care relation goals, and don't-know processes to reduce don't-know relation goals.

Execution alternates between two phases: the and-parallel phase and the deadlock phase. In the and-parallel phase, a don't-care process behaves similar to a process in the JAM (see Section 8.2.1) and may succeed, fail, suspend on variable(s), or suspend on children. A don't-know process starts execution by considering the first '.'-separated group of clauses in the procedure of its goal and testing whether the goal is deterministic. This can have three outcomes:

(a) all clauses are non-candidates:  
the process tries the following '.'-separated group of clauses in the same way, unless this is the last group in which case the process fails.

(b) all clauses except one are non-candidates:  
the process commits to that clause and advances to the next stage.

(c) the goal is non-deterministic:  
the process suspends on the variable(s) that cause the non-determinism of the goal.

If a process suspends on variables and subsequently any one of them gets bound, the process resumes execution, testing the determinism of the goal in the same way again.
Once a process commits to a clause, it unifies the goal with the head of the clause; if that fails then the process fails. Otherwise, the process creates new processes to execute the goals in the right hand side of the clause.

If the body of the clause contains parallel conjunctions separated by sequential conjunction operators, then these are treated similar to the JAM. That is, the process creates child processes for the goals in the first parallel conjunction and suspends waiting for all its children to terminate. If they all succeed, the process resumes to execute the next conjunction. If a child process fails, the parent process also fails. For the last parallel conjunction, sibling processes are created except for the last goal which the current process executes itself (tail recursion optimization).

Additionally, a **deadlock list** of processes is formed. A (don't-care/don't-know) process adds itself to the deadlock list when it suspends on variable(s), and removes itself from the list when it resumes execution.

There are three possible outcomes to the and-parallel phase:

(a) success: all processes have terminated with success. Execution, then, terminates successfully.
(b) failure: one or more processes have failed. Then, a failure event is processed.
(c) deadlock: all the remaining processes in the process tree are suspended. Execution advances to the next phase.

If the computation deadlocks, a **deadlock phase** is begun in which a new don't-care process is created and added to the process tree, having the root process as its parent. The new process executes a goal for a predefined relation, $\text{deadlock}/1$, which implements the Pandora deadlock handler defined in Chapter 6. The input argument of the goal is the deadlock list.

Executing $\text{deadlock}/1$ may remove processes that are suspended on variable(s); these are forced to terminate successfully. If all children of a process are removed, the parent process resumes execution. Moreover, $\text{deadlock}/1$ may spawn processes for executing new goals.

Alternatively, $\text{deadlock}/1$ may select a don't-know process from the deadlock list and create a choice point for it. Once selected, a don't-know process resumes
execution by reducing the goal with the first clause in its procedure in the same way as described before. That is, it unifies the goal with the head of the clause and creates new processes for the goals in the right hand side of the clause. A new and-parallel phase is then begun.

A choice point for a don't-know process contains: a pointer to the process, a pointer to the next clause to try in the don't-know procedure of its goal, plus all the information necessary to restore the state of computation to the time when the choice point was created.

If a don't-care or a don't-know process fails, a failure event is processed in which the computation backtracks to the most recently created choice point: processes that are added to the process tree after the choice point are killed while old processes restore their state to the time when the choice point was created. Then, the (goal) process, for which the choice point was created, tries the next alternative clause in the procedure of its goal.

A choice point remains while there are remaining unexplored clauses in the procedure of the corresponding don't-know relation goal and is removed when the last clause is entered.

If a failure event is processed and there is no choice point left to backtrack to, execution fails.

8.3.2 Comparison with the WAM and the JAM Models

It should be noted that the PAM execution model extends the JAM model in two ways:

1. it introduces a deadlock phase, in which a choice point for a (don't-know relation goal) process may be created, or suspended processes may be removed and/or new processes may be added.

2. failure of a process in the JAM model indicates failure of the execution, while in Pandora it causes backtracking to the most recent choice point. If no choice point left to backtrack to, execution fails.
If the PAM execution model is compared with the WAM model, one may conclude the following:

1. the computation rule in WAM always selects the leftmost goal for reduction, while goals in Pandora are reduced by concurrent processes.
2. as soon as a Prolog goal with more than one clause in its procedure becomes the leftmost goal, a choice point is created for it and the goal is reduced. In Pandora, a choice point is only created in the deadlock phase and for an arbitrary (don't-know relation goal) process, which may well not be the leftmost goal in the conjunction.

8.3.3 The PAM Abstract Machine

The behaviour of an and-parallel phase in the PAM execution model is similar to the JAM provided no choice point is created in a previous deadlock phase. Otherwise, as in WAM, special care has to be taken to be able to restore the state of computation to the time when a choice point was created in case of future failure. New data structures and operations are introduced to the JAM in order to support features of Pandora which are not in committed-choice languages. Some of the JAM data structures and operations are also modified.

As in JAM, only one processor is labelled as master; other processors are slaves. Each processor has its own set of data structures which are accessed by other processors. The role of the process structure's fields in PAM is described in Section 8.3.3.1. The functions of other data structures borrowed from JAM remain the same, while the additional data structures have the following functions:

- **Choice-Point Stack** contains choice point structures.
- **Trail** contains necessary information to undo conditional bindings.
- **Deadlock List** contains pointers to variables with suspension lists. It corresponds to the deadlock list in the execution model.
8.3.3.1 The Process Structure

There are two kinds of process in the PAM: don't-care and don't-know processes. The process structure of both kinds consists of the same fields (illustrated in Figure 8.5). The Parent, Reference-Count, Arguments, Argument-Count, Environment, and Link fields play the same role as in JAM. However, the role of the Code-Pointer field is different in each kind of process. While it (initially) points to the first instruction in the procedure of the don't-care process’ goal, it points to the first instruction in the determinacy code of the don't-know process’ goal. The determinacy code of a don't-know relation procedure is a sequence of instructions produced by the determinacy analysis algorithm (described in Chapter 3) to evaluate a deterministic goal in the and-parallel phase and to suspend a non-deterministic goal on the variable(s) causing its non-determinism.
If the procedure of a don't-know relation includes sequential clause search operators (';'), each ';'-separated group of clauses is analysed independently and a determinacy code is produced for it. If a ';' is reached when evaluating the process and no variable to suspend on, Code-Pointer is set to point to the first instruction in the determinacy code for the next ';'-separated group of clauses. Subsequently, if the process suspends, it will resume from the code after ';;'.

The Type field indicates the kind of the process. It is set to null in a don't-care process while, in a don't-know process, it points to the first instruction in the procedure of the process' goal. When a don't-know process for a non-deterministic goal is selected in the deadlock phase, it resumes execution from the instruction pointed at by the Type field.

For procedures with sequential clause search operators, an optimization could be employed. Whenever the Code-Pointer is modified, Type is also modified to point to the first clause in the next ';'-separated group of clauses.

The Age field in both kinds of process identifies the last time a process field is updated. The role of this field is explained in detail later in the Chapter.

### 8.3.3.2 The Choice-Point Stack

The choice-point stack is a stack of choice point structures in the order of their creation. This stack together with the argument stack correspond to the WAM stack.

A choice point may be created during the deadlock phase for a (don't-know relation goal) process. It contains a pointer to the process structure (Proc), a pointer to the code for the next clause to try in the don't-know procedure should the current clause fail (next-cl), plus the necessary information to restore the state of computation to the time when the choice point was created; these are the values of the following registers:

- **TR** points to top of trail.
- **H** points to top of heap.
- **PP** points to top of process stack.
- **SP** points to top of argument stack.
8. The Pandora Abstract Machine

- DL  points to end of deadlock list.
- PF  points to the first process in the current free list.

Proc points to the process structure of the don't-know relation goal for which this choice point is created. The process structure contains a pointer to the arguments of the goal. It also contains a pointer to the parent process, which in turn has pointers to the environment (Environment) and code (Code-Pointer) to jump to on successful completion of the current goal. Therefore, these information need not be saved explicitly in the choice point structure.

Each processor has its own choice point stack and a register B which points to the base of the stack. In a choice point structure, the tops of the processor's data structures are saved; their values are different for different processors. Choice point structures are pushed on all processors' stacks at the same time during the deadlock phase and they all represent one choice point for a (don't-know relation goal) suspended process. There is no need to save Proc and next-cl in all processors' choice point structures since they are the same for the same choice point; these two fields are only saved in the master's choice point. As a result, a choice point structure in the master's stack consists of eight fields while it consists of only six fields in a slave's stack.

8.3.3.3 The Trail

The trail in the PAM plays the same role as in the WAM. It contains the necessary information to undo conditional bindings when the computation backtracks.

Three types of object may conditionally get bound and, hence, need trailing. These are:

1. variable: a pointer to the conditionally bound variable, together with its old value are saved in the trail.
2. process structure field: a pointer to the process structure which contains the conditionally bound field(s), together with a copy of the old values of its process fields are saved in the trail.
3. hanger: a pointer to the conditionally reset hanger, together with its old value are trailed.

- 187 -
An unbound variable might be pointing to a (hybrid) suspension list; its value may be conditionally updated by either instantiating the variable or adding a new process to its suspension list. When backtracking, the variable should point back to its original suspension list that existed before the creation of the most recent choice point. As a result, trailing a variable saves its old value as well as its address.

Not only a variable can be bound conditionally, but process fields in a process structure may also be conditionally updated. Suppose a parent process $P_0$ is suspended on children $P_1$ and $P_2$, which are both suspended on variables. Then, a choice point is created, $P_1$ resumes execution until termination and decrements $P_0$'s Reference-Count from 2 to 1. If a failure event is processed, the original value of $P_0$'s Reference-Count (equals 2) should be recovered before trying the next alternative clause for the choice point.

If a process is woken up after being suspended on multiple variables, its hanger should no longer point to it so as not to re-execute the process when other variables are bound. Updating the hanger's value is conditional if a choice point was created during the period when the process was suspending. In this case, the hanger's original value, that is the pointer to the process, should be recovered if the computation backtracks.

### 8.3.3.4 The Deadlock List

Each processor in the abstract machine maintains a deadlock list together with a register DL pointing to the end of the list. The union of the deadlock lists in the abstract machine correspond to the deadlock list in the execution model.

Instead of adding each suspended process in the conjunction of goals to the deadlock list, an optimization is employed: when a process is the first to suspend on a variable, a pointer to that variable is added to the end of the list. Other suspensions on the same variable do not affect the deadlock list. As a result, each element in the deadlock list represents a list of processes that are suspended on the same variable.

When a process suspends on multiple variables, pointers to all of the variables
which did not have previous suspension lists are added to the deadlock list. The suspension of other processes on any of these variables will not affect the deadlock list.

8.3.3.5 The Trailing Mechanism

The following issues should be considered when trailing:

(a) a process field can be updated several times between two consecutive choice points. Suppose a process P0 was suspended on three children, i.e. its Reference-Count was equal to 3, then a choice point was created and P0's Reference-Count was decremented twice because two of its child processes were woken up and terminated successfully. P0's Reference-Count is now equal to 1. When backtracking, we need finally to recover the value 3; the intermediate value 2 is irrelevant. Similarly, several processes may suspend on a variable after the most recent choice point, each of them updates the variable's suspension list but only the variable's value when the choice point was created needs to be recovered before trying the next clause.

(b) in WAM, a simple address comparison is performed to check whether a variable's binding is conditional or unconditional. If the variable's address is less than the choice point, i.e. the variable was created before the choice point, then the binding is conditional; otherwise, it is unconditional. In a multi-processor implementation, if the variable is not owned by the processor that is binding it then an address comparison requires checking the addresses of other processors' areas. This test is even more expensive when updating process fields because processes tend to migrate a lot among processors while variables are usually instantiated by their own processors.

In point (a) above, trailing multiple bindings for a variable (or a process field) after the most recent choice point inefficiently consumes time and memory. It also requires keeping the order of bindings that might be saved in different processors' trails, and undoing them in the reverse order so that to finally recover the original value of the variable (or process field).

In order to overcome the above problems, we define a new concept: a data
object's age, which corresponds to the variable's physical address in WAM. Unlike its physical address, a data object's age should be modifiable so that only the first binding of the object after a choice point is *conditional*; other bindings before creating a new choice point are considered *unconditional* and thus do not require trailing.

A new global register, **choice-point count**, is also introduced. It reflects the current number of choice point structures on a choice-point stack. The value of **choice-point count** is tested by all processors but is only modified by the master. It is incremented every time a new choice point is created, and decremented whenever a choice point is removed.

### 8.3.3.5.1 Trailing a Process Structure

In a process structure, an extra field, **Age**, is added to denote the *process' age*. When a process is created, the **Age** field is initialized to the value of **choice-point count** at that time. If a process field is to be updated and the process' **Age** is equal to the current value of **choice-point count**, it is an *unconditional* update. If **Age** is less than **choice-point count**, then the update is *conditional* and should be trailed.

In order to trail a process structure, the processor first examines the **Age** field to check if a conditional update is taking place. If **Age** is less than **choice-point count**, a lock is set which prevents other processors from trailing the process. Once locked, the test for the process' age is repeated in case another processor has trailed the process before the lock was obtained. Then, the whole process structure is trailed, the value of **Age** is modified to be equal to **choice-point count**, the field in the process structure is (conditionally) updated, and the lock is released. Other updates to any of the process fields before creating a new choice point will be considered unconditional. The result is trailing a process structure at most once between two consecutive choice points.

Trailing the whole process structure is justified by the following facts:

(a) Once a process is created, except for the **Parent** field, all the fields in the process structure can be modified during its lifetime whether by the processor executing it or the processors executing its children.

(b) Pandora is designed for applications in which the main theme of
computation is don't-care non-deterministic while don't-know non-determinism is required lazily as the last solution to deadlock situations. In such applications, most of the process structure's fields are expected to be updated, and thus trailed, during the same and-parallel phase.

(c) trailing a field by itself when it is to be conditionally updated requires an extra space to save the age of each field instead of having only one field indicating the age of the whole process.

An extra advantage of trailing the whole process structure is the ability to re-use the structure of a terminated process for a newly created one even if its physical address in the process stack is less than the most recent choice point.

### 8.3.3.5.2 Trailing a Variable

When a variable is created, it is tagged as unbound while its value is initialized to be equal to the value of choice-point count at that time. This represents the variable's age. When a variable is pointing to a suspension list, the variable's age is denoted by the age of the most recent process in its suspension list.

When an unbound variable is assigned a value or when a process suspends on it, its value is conditionally updated if the variable's age is less than the current value of choice-point count. Otherwise, the update is unconditional.

Figure 8.6(a) illustrates an unconditional update of X's value field. Initially X was created with age 1. Process P0 then suspends on it. Since X's age is equal to choice-point count, no trailing is required.

![Figure 8.6(a)](image_url)

Fig 8.6(a). X's age when choice-point count = 1.
Figure 8.6(b) illustrates P1's suspension on X when choice-point count is equal to 2 while X's age is equal to (P0's age) 1. Since X is older than the current choice point, it is trailed before adding P1 to its suspension list. X's age is now equal to 2. Thus, P2's suspension on X is unconditional.

![Diagram](image)

Fig 8.6(b). X's age when choice-point count = 2.

As in the case of trailing a process structure, trailing a variable requires an exclusive access to it.

### 8.3.3.5.3 Trailing a Hanger

When a process suspends on multiple variables, a suspension note is added to the suspension list of each of these variables (see Section 8.2.2.8). A suspension note is pointing indirectly to the process via its hanger. The age of the suspension notes as well as the hanger is equal to choice-point count when the process has suspended. If the process' age is less than choice-point count, then it is trailed and its Age field is modified before being suspended.

In the JAM, waking a process that was suspended on multiple variables would set its hanger to null. In the PAM, the hanger is set to the value of the Age field in the process structure (while its tag is set to unbound). If Age is less than choice-point count, then updating the hanger is conditional and should be trailed. An exclusive access to the hanger is required when trailing it and setting its value.

When a suspension list is woken up, a suspension note in the woken list can be
"only" deallocated if its age is younger than the most recent choice point.

8.3.3.6 Collecting Suspended Processes in the Deadlock Phase

Whereas deadlock is considered a run-time error in committed-choice languages, it plays an important role in Pandora. When the computation deadlocks, the master processor scans the deadlock lists of all processors and generates Suspended_Processes: a list of the processes that are still suspended on variable(s).

An element in a deadlock list points to a variable. If the variable is still unbound, then a pointer to each process in the variable's suspension list is added to Suspended_Processes.

When collecting a process that is suspended on multiple variables, there is the problem of ensuring that the process is collected only once and thereafter other variables no longer refer to the process. A similar technique to the one used for waking up such process (see Section 8.2.2.8) is used for collecting it in the deadlock phase. After adding the process to Suspended_Processes, its hanger is temporarily set to null; other variables simply ignore it since its value is null and thus will not recollect the process.

After all the deadlock lists are scanned, i.e. all processes suspended on variable(s) are collected, the hangers should be set back to point to their corresponding processes. Therefore, a temporary trail is used by the master processor to temporarily trail the original value of a hanger before setting it to null and to set it back to its value after scanning all deadlock lists. This operation does not require an exclusive access to the process structure (or its hanger) since the master processor is the sole processor that is active at that time.

8.3.3.7 The Pandora Deadlock Handler

After generating Suspended_Processes, the master processor loads its argument register A1 with the Suspended_Processes list and creates a process to evaluate a
goal for the predefined $\text{deadlock/1}$ relation, which the master executes itself. $\text{deadlock/1}$ implements the Pandora deadlock handler defined in Chapter 6.

If $\text{deadlock_handler/4}$ is not defined in the program, $\text{deadlock/1}$ searches for a don't-know process in Suspended_Processes and sets a global register, choice-point flag, to point to that process. A don't-know process is a process whose Type field has a non-null value.

If $\text{deadlock_handler/4}$ is defined, Meta_Suspended is generated. Meta_Suspended is a list of meta-level representation of Suspended_Processes. Each process in Suspended_Processes is represented by a tuple of the form:

\[(\text{Goal}, \text{Reference})\]

where Goal is a ground term representing the goal of the process (described in Chapter 6), and Reference is a pointer to the process itself. Additionally, a dictionary, Meta_Var_Dictionary, of all the variables in the arguments of these processes (i.e. the variables in the deadlock lists), together with their meta-level representation in Meta_Suspended, is generated. Then, $\text{deadlock/1}$ creates a child (don't-care) process for $\text{deadlock_handler/4}$, whose input argument is Meta_Suspended, and suspends on children.

Failure of $\text{deadlock_handler/4}$ is the same as failure of any other (goal) process (explained in Section 8.3.3.9). If $\text{deadlock_handler/4}$ succeeds, the $\text{deadlock/1}$ process resumes to inspect the bindings of $\text{deadlock_handler/4}$'s output arguments.

$\text{deadlock/1}$ may remove processes in Suspended_Processes. A process is removed by setting its Argument-Count to '-1' to indicate that the process is dead and decrementing the Reference-Count of its parent process. When a variable on which the process is suspended gets bound, the process will not be executed since it is dead. Updating the Argument-Count field of the removed process as well as decrementing its parent's Reference-Count may require trailing.

If new goals should be added to the conjunction of goals, Meta_Var_Dictionary is inspected and any meta-level representation of a variable in these goals is replaced by a reference to the corresponding variable. Then, $\text{deadlock/1}$ spawns sibling processes for the new goals and terminates successfully.
Alternatively, a don’t-know process may be selected for a non-deterministic execution; **choice-point flag** is then set to point to that process.

### 8.3.3.8 Scheduling

The scheduling algorithm in the PAM extends the JAM scheduler (see Section 8.2.2.10) with a deadlock phase.

When deadlock occurs **choice-point flag** is examined. If the flag is set to null, the master processor generates Suspended_processes, as described in Section 8.3.3.6, and loads its argument register A1 with this list. It then creates a process to invoke $deadlock/1; thus it leaves the deadlock state and, so other processors re-enter the search state. The parent of the created process is the root process.

If the **choice-point flag** is non-null when examined, each processor pushes a new choice-point structure onto its choice-point stack and sets its PF to null to start a new process free list. Then, a slave processor re-enters the search state while the master executes the process for which the choice point was created after incrementing **choice-point count** and resetting the **choice-point flag** to null. Figure 8.7 illustrates the PAM scheduler.

In order to non-deterministically execute the selected (don’t-know relation goal) process, the process should be removed from the suspension list(s) on which it is suspended. Removing the actual process structure from the suspension list would require modifying the pointer to the process from the previous element in the list. Alternatively, a copy of the process structure with age equals to the current value of **choice-point count** is made to act as a **substitute** (don’t-know relation goal) process while the original process is marked to be dead.
8.3.3.9 Failure and Backtracking

During the evaluation of a don't-care process, when a clause fails in input matching or in a guard evaluation, next clause is entered. A pointer to the next clause to try is saved in the processor's FL register. If there are no more clauses to try or variables to suspend on, then a failure event is signalled.

Similarly, when evaluating a don't-know process in the and-parallel phase, failing
to reduce or suspend the process involves entering the determinacy code for the
next '.'-separated group of clauses. If there are no more groups to try or variables
to suspend on, then a failure event is signalled.

When a failure event is signalled, each processor backtracks to the most recent
choice point. Objects that are trailed are returned their old values from the trail, and
registers are reset to their values as saved in the choice point. PF is set to null and
the run queue is emptied.

An object does not need to be locked when restoring its value from a processor's
trail since it is trailed only once after the most recent choice point.

After backtracking, a slave processor loops searching for work, while the master
processor loads the process for which the choice point was created and executes it.
If the clause being entered is the last clause to try for the goal, then the master
resets PF of all processors to their values as saved in the most recent choice point,
and decrements choice-point count by one before entering the clause (last
clause optimization). Otherwise, the master updates the next-cl field in the choice
point structure to point to the clause following the one being entered.

Since environments and choice points are saved in separate stacks, and a choice
point is a fixed size structure, the location of the most recent choice point on a
processor's stack can be computed by indexing from B using the current value of
choice-point count.

8.4 Summary

This chapter describes an abstract machine for Pandora. It extends the JAM with
the minimum requirements to support features of Pandora which are not in
committed-choice languages.

Three data structures have been introduced: the choice-point stack, the trail, and the
deadlock list. The choice-point stack and the trail support don't-know non-
determinism in Pandora. They play the same role as their counter parts in a
standard Prolog implementation such as the WAM. The deadlock list supports the
Pandora deadlock-handler. It is used to collect the remaining suspended goal
processes on deadlock.

A new concept has been defined, a "data object's age", to play the role of the physical address for a data object in the WAM. Unlike the physical address, a data object's age has to be modifiable to accommodate the multiple updates to the value of the data object between two consecutive choice points. It should also be independent of which processor the data object belongs to since a data object can be created by one processor in its own data structure and accessed by other processors.

An extra benefit from the age concept is the decentralization of the trail and choice-point stack. Each processor has its own trail and choice-point stack which it manages. This has removed the necessity to lock the trail when adding (or removing) an item to (from) it and has minimized the periods when a processor remains idle.

The deadlock list has been introduced to be able to collect suspended processes on deadlock. Instead of scanning the heap to collect the processes that are suspended on variables, the deadlock list points to variables with suspension lists. Thus, an element in the deadlock list represents a list of suspended processes.

If the meta-level deadlock handler relation is defined, two extra data structures should be generated and manipulated, namely: Meta_Suspended and Meta_Var_Dictionary. One way of implementing these structures was proposed. However, future research should measure their cost and possibly investigate alternative implementations for supporting meta-level programming in the deadlock phase.

The PAM machine is highly tuned towards the execution of committed-choice programs, i.e. programs that do not require the creation of choice points. The only overhead introduced by the PAM is a comparison operation for each binding to check whether it is conditional. Since all bindings in these programs are unconditional, no trailing is required and, hence, the overhead is expected to be low when compared to running the same programs on the JAM. For illustration purpose, Table 8.1 presents the run-time (in seconds) of several committed-choice programs on the Andorra-I system (Costa et al, 1990b) with and without the comparison operation per each binding. Andorra-I is is a prototype implementation of the basic Andorra model which runs both or-parallelism (closely following the
or-parallel Prolog system Aurora (Lusk et al, 1990)) and stream and-parallelism (closely following the JAM).

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Andorra-I</th>
<th>Andorra-I + comparison</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>fibonacci_15</td>
<td>1.53</td>
<td>1.54</td>
<td>0.6%</td>
</tr>
<tr>
<td>naive_reverse_150</td>
<td>3.61</td>
<td>3.7</td>
<td>2.5%</td>
</tr>
<tr>
<td>merge_sort_200</td>
<td>1.57</td>
<td>1.61</td>
<td>2.5%</td>
</tr>
<tr>
<td>boyer</td>
<td>6.05</td>
<td>6.11</td>
<td>1.0%</td>
</tr>
</tbody>
</table>

Table 8.1. The overhead of a comparison operation per each binding on committed-choice programs.

The average overhead of the comparison operation on committed-choice programs, as shown in Table 8.1, is 1.65%. However, this percentage should be slightly higher when comparing the performance of the PAM with the JAM, since the JAM is a compiled system while Andorra-I is an interpreter (i.e. slightly slower).
Chapter 9

Conclusions

To conclude, a summary of our results is presented and areas of related research are outlined. Finally some directions are suggested for future development of this research.

9.1 Summary

The objective of this research has been to design a new logic programming language which achieves three main goals:
   1. to extend committed-choice languages with search capabilities.
   2. to open up new interesting application areas to logic programming.
   3. to be feasible to implement on both single and multi-processor architectures.

Pandora: a non-deterministic parallel logic programming language, has been developed. Two kinds of relation can be defined in a Pandora program: don't-care and don't-know relations. A don't-care relation has the same syntax and semantics as a Parlog procedure. Hence, Pandora supports a committed-choice stream and-parallel evaluation of conjoined goals for don't-care relations. A typical application which requires such evaluation strategy is systems programming (Foster, 1988).

Having don't-know relations enriches the language with intelligent search capabilities. Deterministic goals for don't-know relations are evaluated in parallel. When the computation deadlocks, an arbitrary goal for a don't-know relation can be selected and a choice point created for it. For each search branch, a new and-parallel evaluation is then begun, searching for solution(s) to the program. As a
result, Pandora adopts an "incremental generate and test" evaluation strategy which can significantly reduce the search space in many cases, resulting in a more intelligent problem-solving behaviour than a sequential evaluation, as in Prolog, or a fully parallel evaluation, as in CP[↓,↓,↓,↓,↓]8. This strategy has opened up interesting application areas to logic programming. Two of these applications have been described in the thesis, namely constraint programming and the distributed discrete event simulation of systems of parallel communicating processes.

In addition to a simple non-deterministic fork, Pandora provides a meta-level deadlock handler relation by which the user can explicitly program the behaviour on deadlock in various (application-dependent) ways. One way in which the deadlock handler relation can be used is in combination with the non-deterministic fork, to implement a heuristic search. Another use is to implement a non-forking deadlock breaking mechanism by removing existing goals and/or adding new goals to the current conjunction of goals. In general, deadlock phases can be viewed as milestones at which the state of computation can be checked and suspended goals can be manipulated in a flexible manner, depending on the needs of the application in hand. This mixture of object-level and meta-level programming has proved to be useful in several applications discussed in the thesis, the last of which is the implementation of constraint logic programming and hierarchical constraint logic programming systems.

Another advantage of "lazy" don't-know non-determinism is to make the combination of stream and-parallelism and don't-know non-determinism feasible to implement, since a non-deterministic choice is only made when the concurrent evaluation has terminated. PANDA: a prototype implementation of Pandora, was developed. PANDA is written in Parlog and runs on top of the Parlog SPM system. It comprises a compiler for translating Pandora programs into Parlog procedures and an interpreter which evaluates queries to the compiled Pandora programs using the Parlog meta-call.

If a goal for a don't-know relation is selected for a non-deterministic evaluation in the deadlock phase, a copy of the conjunction of suspended goals is made for each search branch and is concurrently evaluated after reducing the copy of the selected goal with a clause in its procedure.

---

8 briefly described in Chapter 2.
Although copying is expensive, it provides the flexibility of easily swapping between a sequential backtracking implementation of don't-know non-determinism and a parallel evaluation of the alternative branches of a choice point.

PAM, an abstract machine for Pandora on shared-memory multi-processor architectures, has been designed. It extends the JAM abstract machine for committed-choice languages with compilation and memory management techniques that are used in sequential Prolog implementations based on the WAM. In addition, PAM introduces several new concepts, data structures and operations which lead to a homogeneous integration of the implementation techniques adopted from both machines.

PAM is highly tuned towards the execution of committed-choice programs, i.e. programs that do not require the creation of choice points. The only overhead introduced by the PAM, compared to the execution of these programs on the JAM, is a comparison operation for each binding to check whether it is conditional. As mentioned in Chapter 8, the average overhead of the comparison operation on committed-choice programs is slightly higher than 1.65%.

### 9.2 Related Research

Kernel Andorra Prolog (Haridi and Janson, 1990) is a language closely related to Pandora and, like Pandora, it is inspired by the basic Andorra model. It is intended to subsume Prolog, committed-choice languages and, eventually, constraint logic programming languages. The design of the language is still in progress.

Clauses in a Kernel Andorra Prolog procedure may include a sequential conjunction operator and the pruning operator: commit and cut. Similar to committed-choice languages, the commit operator is used to commit to any clause whose guard is satisfied, while a cut operator commits to the first clause with a satisfied guard, following the textual order of the clause. In both cases, other candidate clauses are pruned on commitment.

Kernel Andorra Prolog programs can be run using an extended form of the basic Andorra model, called the extended Andorra model (Warren, 1990). In the basic Andorra model, a non-deterministic reduction of a goal is delayed until all deterministic computations take place. However, the extended Andorra model
allows and-parallel execution of non-deterministic goals but delays the propagation of their bindings to shared variables. If the execution of a conjunction of goals deadlocks, the non-deterministic bindings to shared variables are propagated.

In order to propagate a non-deterministic binding to a shared variable $X$, suspended goals in the conjunction are copied to the different search branches. Ideally, only goals which consume the binding of $X$ are copied while the evaluation of other goals is not duplicated in the search branches.

Unlike Pandora, Kernel Andorra Prolog does not support a user-level manipulation of suspended goals when the computation deadlocks. Moreover, the extended Andorra execution model is relatively complex and it remains to be seen how efficiently it can be implemented.

### 9.3 Future Research

Pandora was successfully used to explore the class of applications that cannot be conveniently expressed in either Prolog or Parlog. However, experience has suggested that the current design of the language could be improved. For example, there are several differences between the behaviour of don’t-know and don’t-care relations: (a) don’t-care relations cannot be used to create choice points; (b) they use a different method of suspension; and (c) they effectively have a commit operator in every clause. For the new class of applications, it would be more elegant to unbundle these features so that they can be used in different combinations.

Another design issue is to define the behaviour of negated calls in Pandora programs; an issue which was not considered in the thesis.

Exploring wider application areas that can be supported by Pandora will guide the work on developing and refining the language. For instance, our experience in constraint programming has revealed that Pandora can be used for problems involving constraints on finite domains. Significantly, this class of applications cannot be programmed efficiently in either standard Prolog or Parlog. In Parlog, the search has to be programmed explicitly, while in Prolog, all constraints must repeatedly be checked to detect which can be solved. In contrast, Pandora provides both built-in support for search and a coroutining strategy which ensures the required propagation of constraints. However, it is rather more difficult to write a
constraint logic program in Pandora than a "generate and test" program for the same problem. A more convenient notation is required and so, we are currently designing a specialized version of Pandora for handling constraints on finite domains. The language includes a set of operators for dynamically defining (overlapping) domain variables over the domain of reals, and provides a set of constraint primitives which can actively reduce the domains of variables. User-defined constraints can be defined using these constraint primitives. An initial implementation for the language has been developed which compiles its programs into Pandora programs and runs them on a Pandora implementation. The design of the constraint language has hardly begun and, for reasons of efficiency, it may be desirable to implement the language by extending the Pandora implementation with some low-level support for handling domain variables.

Several design issues in the PAM abstract machine have been successfully examined using the Andorra-I system however, the PAM has not yet been implemented. Additionally, future research may investigate various means of supporting meta-level programming in the deadlock phase.
References


References


- 206 -
References


Science, University of Texas at Austin.


References


References


