# Contents

1 Introduction .................................................. 1

1.1 Motivation ...................................................... 1
1.2 Objectives ...................................................... 3
1.3 Preliminary Examples .......................................... 4
  1.3.1 Narrowing .................................................. 4
  1.3.2 Constraints ................................................ 5
  1.3.3 Choice and Split Constructs ......................... 7
1.3.4 Enumeration .............................................. 10
  1.3.5 Floating Point Variables ......................... 11
1.4 Organization ................................................. 13
1.5 Historical Background and Related Work ................. 14
  1.5.1 Non-Determinism ....................................... 14
  1.5.2 Interval Arithmetic ................................... 16
  1.5.3 Constraint Programming Languages ............... 18
  1.5.4 Logic Programming .................................... 19
  1.5.5 Constraint Logic Programming (CLP) Languages ........ 19
1.5.6 Previous Brandeis Research in Parallel Versions of Logic Programming Languages .................. 23
1.5.7 Final Remarks .................................................. 25

2 Basic Operations on Intervals .................................. 26

2.1 Numbers ......................................................... 26
2.1.1 Interval Arithmetic .......................................... 28
2.2 Operations using Intervals .................................... 29
2.3 The Intersection operator $\cap$ .............................. 30
2.4 Open and Closed Intervals ................................... 30
2.5 Narrowing ...................................................... 32
2.5.1 Properties of the Narrowing Function .................. 37
2.5.2 Increasing the Efficiency of Narrowing ............... 38

3 The Language ...................................................... 41

3.1 Grammar ......................................................... 42
3.2 Data Structures for Description of Algorithms .......... 42
3.3 Nucleus of the Constraint Solver ........................... 44
3.4 The Procedure Solve .......................................... 45
3.4.1 The procedure Narrow .................................... 47
3.5 Passive Constraints ........................................... 54
3.6 Data Types ...................................................... 54
3.6.1 Variable Types ............................................. 54
3.6.2 List Processing with Arrays ............................. 57
3.7 Imperative Statements ........................................ 58
3.8 Procedures ...................................................... 58
3.9 Domain Splitting and Tree Traversal Strategies .................................. 59
3.10 Non-deterministic Constructs ................................................................. 64
    3.10.1 Choice ......................................................................................... 64
    3.10.2 Split ......................................................................................... 65
    3.10.3 Fail ......................................................................................... 65
    3.10.4 Either ... Or ........................................................................... 66
    3.10.5 Some ....................................................................................... 66
    3.10.6 Redo ....................................................................................... 67
3.11 Modified Procedure Solve ....................................................................... 68

4 Examples .................................................................................................... 70

4.1 Deterministic Programs ........................................................................... 71
    4.1.1 Installment Problem ................................................................... 72
    4.1.2 Circuit Analysis-1 ...................................................................... 72
    4.1.3 Circuit Analysis-2 ...................................................................... 73
    4.1.4 Polynomial Interpolation ............................................................. 77
    4.1.5 Diophantine Equations and a Problem Suggested by Fermat ...... 79
    4.1.6 Geometric Examples .................................................................. 82
    4.1.7 Scheduling Problems .................................................................. 84
4.2 Non-deterministic Programs .................................................................... 86
    4.2.1 Fibonacci Numbers .................................................................... 87
    4.2.2 N Queens ................................................................................. 90
4.3 Hidden Markov Models ........................................................................... 92
4.4 Computationally Difficult Problems ....................................................... 100
    4.4.1 3-SAT .................................................................................. 101
7.3 Results .............................................. 168
7.4 Enhancements ........................................ 169
  7.4.1 Connected Components ............................ 169
  7.4.2 Equal Partitioning of Constraints ............... 169

8 Shared Memory MIMD Implementation ............... 172
  8.1 Data Structures ..................................... 173
  8.2 The Parallel Procedure Solve ........................ 174
  8.3 Analysis ............................................ 179
  8.4 Processor Scheduling ................................ 181
  8.5 Implementation Details .............................. 187
  8.6 Example — N Queens ................................ 193
  8.7 Results ............................................. 198

9 Graphical User Interface .............................. 199
  9.1 Program Input and Modification .................... 201
  9.2 Program Execution .................................. 205
  9.3 Trace and Debug Windows ............................ 210

10 Conclusions ......................................... 211
  10.1 The Aim of the Theoretical Analysis ............... 212
  10.2 Future Work ....................................... 213
    10.2.1 Distributed Implementation .................... 214
  10.3 Final Remarks ..................................... 215
Bibliography ............................................ 224
List of Figures

1.1 The Fahrenheit-Celsius example using the choice construct. ............ 8
1.2 The Fahrenheit-Celsius example using the split construct. ............ 9
1.3 The program showing Fahrenheit and Celsius relationship. .......... 11
1.4 Enumeration search tree using depth-first strategy. ............... 12
1.5 Brandeis research in parallel logic programming and constraint logic programming ......................................................... 25

2.1 The Intersection operation ....................................................... 31
2.2 The Narrow Operations .......................................................... 33
2.3 Narrowing steps for the constraint \( x \times y = z \) ....................... 35
2.4 Intervals for variables in \( x \times y = 1 \) .................................. 35
2.5 Intervals for variables in \( x = y^2 \) ...................................... 36
2.6 An example for illustrating cyclic phenomena in narrowing. ........ 40

3.1 A subset grammar for the language ................................. 43
3.2 An example ................................................................. 44
3.3 List structure corresponding to the program in Figure 3.2 ........ 45
3.4 Nucleus of the constraint solver ........................................... 46
3.5 The procedure \textit{solve}. .................................................. 48
3.6 The procedure \textit{enumerate}. ............................................. 49
3.7 The procedure \textit{split intervals}. ......................................... 50
3.8 The procedure \textit{narrow}. .................................................. 52
3.9 The procedure \textit{add constraints}. ......................................... 53
3.10 Types Example ........................................................................ 56
3.11 Output of the program ............................................................ 56
3.12 Lists ......................................................................................... 57
3.13 Imperative Statements .............................................................. 58
3.14 An example for illustrating splitting strategies. ............................... 59
3.15 Depth-first traversal ................................................................. 61
3.16 Round-robin traversal ............................................................... 63
3.17 Fair Splitting ............................................................................. 64
3.18 Redo example ........................................................................... 67
3.19 The Revised procedure \textit{solve}. ............................................... 69

4.1 The Installment program ........................................................... 71
4.2 Circuit problem .......................................................................... 73
4.3 Circuit Program .......................................................................... 74
4.4 Full-adder Circuit ....................................................................... 75
4.5 Program Verifying the Full-adder Circuit ......................................... 76
4.6 Polynomial Interpolation ............................................................... 78
4.7 Diophantine Equations ................................................................. 79
4.8 Narrowing operations for the Fermat’s example ............................... 81
4.9 Precedence Diagram for the Scheduling Problem ............................. 84
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.10</td>
<td>A Non-deterministic Program</td>
<td>86</td>
</tr>
<tr>
<td>4.11</td>
<td>Depth-first Traversal</td>
<td>87</td>
</tr>
<tr>
<td>4.12</td>
<td>Fibonacci Program</td>
<td>88</td>
</tr>
<tr>
<td>4.13</td>
<td>Fibonacci Program</td>
<td>89</td>
</tr>
<tr>
<td>4.14</td>
<td>N-Queens</td>
<td>91</td>
</tr>
<tr>
<td>4.15</td>
<td>Finite state automaton for the coin example</td>
<td>92</td>
</tr>
<tr>
<td>4.16</td>
<td>DAG for the coin example</td>
<td>94</td>
</tr>
<tr>
<td>4.17</td>
<td>Program to compute the states for the coin tossing example</td>
<td>97</td>
</tr>
<tr>
<td>4.18</td>
<td>Plot for different regions of the probabilities for varying $n$ and $q = 0.9$</td>
<td>98</td>
</tr>
<tr>
<td>4.19</td>
<td>Plot for different regions of the probabilities for varying $q$ and $n = 30$</td>
<td>99</td>
</tr>
<tr>
<td>4.20</td>
<td>3-SAT Satisfiability</td>
<td>102</td>
</tr>
<tr>
<td>4.21</td>
<td>3-SAT Example</td>
<td>105</td>
</tr>
<tr>
<td>4.22</td>
<td>A Job-Shop scheduling problem with 3 machines and 4 jobs</td>
<td>114</td>
</tr>
<tr>
<td>4.23</td>
<td>Program to compute the dot-product</td>
<td>119</td>
</tr>
<tr>
<td>4.24</td>
<td>Specialized version of the dot-product program when $N = 4.$</td>
<td>120</td>
</tr>
<tr>
<td>4.25</td>
<td>Specialized version of the dot-product program with a known input vector</td>
<td>120</td>
</tr>
<tr>
<td>5.1</td>
<td>The $Solve$ automaton</td>
<td>124</td>
</tr>
<tr>
<td>6.1</td>
<td>The modules of the sequential interpreter</td>
<td>136</td>
</tr>
<tr>
<td>6.2</td>
<td>The $Intervals$ data structure</td>
<td>136</td>
</tr>
<tr>
<td>6.3</td>
<td>The $constraints_array$ data structure</td>
<td>138</td>
</tr>
<tr>
<td>6.4</td>
<td>The $Node_ptr$ data structure</td>
<td>138</td>
</tr>
<tr>
<td>6.5</td>
<td>The $Proc_ptr$ data structure</td>
<td>139</td>
</tr>
<tr>
<td>6.6</td>
<td>The $Choice Point$ data structure</td>
<td>140</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>7.1</td>
<td>Flowchart of the data-parallel solver</td>
<td>146</td>
</tr>
<tr>
<td>7.2</td>
<td>Processor configurations during narrow and intersection</td>
<td>150</td>
</tr>
<tr>
<td>7.3</td>
<td>Example illustrating the operations in the data-parallel solver</td>
<td>152</td>
</tr>
<tr>
<td>7.4</td>
<td>Example illustrating convergence</td>
<td>154</td>
</tr>
<tr>
<td>7.5</td>
<td>Algorithm for the data-parallel solver</td>
<td>155</td>
</tr>
<tr>
<td>7.6</td>
<td>Algorithm for compaction</td>
<td>156</td>
</tr>
<tr>
<td>7.7</td>
<td>Algorithm for split and copy</td>
<td>158</td>
</tr>
<tr>
<td>7.8</td>
<td>Example Illustrating the Split and Copy Phases</td>
<td>159</td>
</tr>
<tr>
<td>7.9</td>
<td>Algorithm for active sets</td>
<td>160</td>
</tr>
<tr>
<td>7.10</td>
<td>Revised algorithm for split and copy</td>
<td>161</td>
</tr>
<tr>
<td>7.11</td>
<td>Maple program to compute the time formula</td>
<td>163</td>
</tr>
<tr>
<td>7.12</td>
<td>Probability relationships</td>
<td>164</td>
</tr>
<tr>
<td>7.13</td>
<td>Processor Set Relationships</td>
<td>167</td>
</tr>
<tr>
<td>7.14</td>
<td>Speed-up results</td>
<td>168</td>
</tr>
<tr>
<td>7.15</td>
<td>Improved algorithm for narrowing</td>
<td>171</td>
</tr>
<tr>
<td>8.1</td>
<td>An example program</td>
<td>173</td>
</tr>
<tr>
<td>8.2</td>
<td>List structure for the program</td>
<td>174</td>
</tr>
<tr>
<td>8.3</td>
<td>The Procedure <code>solve</code></td>
<td>176</td>
</tr>
<tr>
<td>8.4</td>
<td>Execution tree for the program</td>
<td>177</td>
</tr>
<tr>
<td>8.5</td>
<td>The procedure <code>enumerate</code></td>
<td>178</td>
</tr>
<tr>
<td>8.6</td>
<td>Parse tree</td>
<td>180</td>
</tr>
<tr>
<td>8.7</td>
<td>Labeling the execution tree</td>
<td>182</td>
</tr>
<tr>
<td>8.8</td>
<td>Time steps for the execution tree</td>
<td>182</td>
</tr>
<tr>
<td>8.9</td>
<td>Program for labeling the execution tree</td>
<td>183</td>
</tr>
</tbody>
</table>
8.10 Sample execution tree with input representation .................................. 184
8.11 Speed-up curves for varying length ......................................................... 185
8.12 Speed-up curves for varying enumeration times ........................................ 186
8.13 Speed-up curves for varying narrow times .............................................. 186
8.14 Shared memory MIMD machine execution model ..................................... 187
8.15 The procedure \texttt{psolve} ................................................................. 189
8.16 The code executed by each processor ....................................................... 190
8.17 The procedure \texttt{narrow} ............................................................... 192
8.18 The procedure \texttt{add\_constraints} .................................................. 193
8.19 Program for the 5 Queens ............................................................. 194
8.20 Constraints for the 5 Queens .......................................................... 196
8.21 Details of the narrow operation ....................................................... 197
8.22 Speed-ups ................................................................. 198

9.1 Graphical User Interface ............................................................... 200
9.2 The 5-Queens problem ............................................................ 202
9.3 Statement types in the language ..................................................... 203
9.4 Strategies for selecting a variable to be split ...................................... 205
9.5 Splitting of a variable ............................................................. 206
9.6 One of all solutions ................................................................. 207
9.7 Execution trace ................................................................. 208

10.1 Outline of similarities between parallel shared-memory and distributed
version .................................................................................................. 215
Chapter 1

Introduction

1.1 Motivation

It is fair to state that computer languages usually differ from each other by characteristics that can be viewed as positive, according to certain precepts, and negative according to others. The design of an “ideal” language involves combining the most of the positive features and the least of the negative ones. This is more easily said than done because languages are designed to facilitate problem solving, and the problems under consideration are inherently different from each other.

There is however a large class of problems, combinatorial in nature, for which one could hope to design a language combining the most desirable features of existing languages. This is certainly less of an indomitable task than aiming at the design of a general language directed to general problem solving. To be more explicit, the combinatorial problems arise in finite domains, scheduling problems, and alike. It is posited that the desirable features required in such a language are: assignment, non-determinism, constraints, and intervals. The significant number of problems described in this disserta-
tion and that have been solved using the proposed language reinforce that proposition.

The explicit presence of the assignment in the language simplifies significantly its implementation. Non-determinism enables the succinct description of multiple choices thus facilitating space searches. Constraints summarize the equations and relations that have to be satisfied to obtain an answer for the problem being considered. Finally, intervals enable the en masse processing of a range of values instead of a single value.

In the process of developing the proposed language, the following features were kept in mind:

i) the programs in the language should be easily compilable and executable using sequential and parallel computers, and ii) the language should exhibit similarities with widely used imperative languages.

A potential question also arises. Why not use an existing Prolog-based constraint language? An answer to that question includes considerations like: an interpreted or a compiled language like CLP(\mathbb{R}), Prolog III & IV is a major project involving the design of complex data structures for unification, stack, trail, and garbage collection; such language also requires the development of specialized and intricate solvers using Simplex, Gaussian elimination, etc.; consideration should be given to type conversion and interaction among types; finally, one should not neglect that the significant overhead needed to obtain generality reduces the potential for use of parallelism.

Therefore, in designing a nucleus of a CLP language that is easy to parallelize — and analyze its speed-up — one has to reduce the basic features to a few that can indeed take advantage of parallelization. In that context, unification is ruled out (DwKaMi 84). and replaced by simple assignments. The choice statement, (the counterpart of OR-parallelism in logic programming), then becomes the prominent parallel construct to
represent non-deterministic situations.

The factors considered in the design of the proposed language and its implementation are:

- parallelism is exploited at different granularities (i.e., data parallel and shared memory parallelism)
- simplicity in implementation due to the widespread usage of assignments,
- parameter passing only involves creating new variables and generating equality constraints,
- explicit descriptions of non-deterministic searches,
- availability of a graphical user interface enabling users to easily execute and debug programs.

The second and third considerations are introduced since it is known that full and safe unification can be computationally expensive and that general operation is often not required for the class of problems considered in this thesis.

1.2 Objectives

Having motivated the reader for the worthiness of the design and implementation of the language proposed herein, I can now put forward the objectives of the thesis:

1. Demonstrate the usefulness of non-determinism, constraints, and interval variables by proposing a small number of language constructs that can be easily incorporated to an imperative language. These features offer great potential for parallelism.
2. Exploit different kinds of OR-parallelism on shared memory multi-processing machines.

3. Show how data-parallel algorithms are advantageous and provide valuable speed-ups in the processing of interval constraints.

4. Provide a significant anthology of examples of the language usage in various applications, including numerical problems, scheduling, hidden Markovian models, etc.

5. Design and develop a graphical interface with which users can write and debug programs.

In what follows, I present preliminary examples that introduce the uninitiated reader to the major features of the language proposed in this work. Following these examples, I will provide historical notes that place this present work in the context of existing paradigms and languages.

1.3 Preliminary Examples

The prominent features of the proposed language are initially explained in this section by means of examples. They illustrate the major characteristics of the language, namely: 

narrowing, constraints, choice and split constructs, tree traversal and branching, and floating point numbers.

1.3.1 Narrowing

The first example is presented to acquaint the reader with some of the key properties of interval variables. An interval variable is specified by a pair of numbers, the lower
and the upper bounds. The bounds may be either integers, floating point numbers, or the symbols $\infty$ and $-\infty$. The latter are markers that are taken into consideration when the operations are performed using interval variables as operands. Most operations using interval variables often have as effect reducing one or both bounds of those variables. The process of bound reduction is called *narrowing*. Consider the equality constraint $x = y$. Suppose $x$ is initialized with the interval $[-30, 40]$ and the variable $y$ is uninitialized, i.e., $y$ has the interval $[-\infty, \infty]$. The narrowing operation for the equality operator reduces the bounds of $y$ to $[-30, 40]$. On the other hand, if the initial interval for the variable $y$ is $[20, 50]$, then the narrowing reduces the bounds for both the variables $x$ and $y$ to $[20, 40]$.

As a result of the narrowing, interval constraints can lead to a failure (e.g., if the variable $y$ in the above equality constraint is in the range $[50, 60]$.) Then all the non-deterministic situations implying in failure are dealt with as in logic programming, leading whenever possible to a backtracking state in a sequential implementation, or the termination of a branch in parallel implementation. In the case of SIMD implementation, the narrowing operation for an entire set of constraints is performed in parallel. In contrast the MIMD implementation deals with narrowing in a sequential manner. More on this topic will be discussed in Sections 3.10 and 8.2.

### 1.3.2 Constraints

The next example introduces the concept of *constraints* and illustrates a fundamental difference between imperative and constraint programming languages. Consider the formula (i.e., a constraint) used to translate temperatures from centigrades to degrees
Fahrenheit:

\[ F = 1.8 \times C + 32 \]  \hspace{1cm} (1.1)

where \( F \) and \( C \) represent the Fahrenheit and Celsius temperatures respectively. In an imperative language an assignment representing this formula only allows the translation from centigrades to degrees Fahrenheit. By considering \( F \) and \( C \) as logical variables, the value of \( C \) can be determined when \( F \) is known, or the value of \( F \) can be determined when \( C \) is known. When both \( F \) and \( C \) are known the equality can be viewed as a boolean expression whose value is either \textit{true} or \textit{false}. Finally, if both \( F \) and \( C \) are unknown the result of the evaluation is the formula itself.

In the language proposed in this thesis, the variables \( F \) and \( C \) can be interval variables. Suppose \( C \) is an interval in the range \([0, 10]\). The constraint specified in equation 1.1 then becomes:

\[ C = [0, 10] \land (F = 1.8 \times C + 32). \]  \hspace{1cm} (1.2)

To compute the interval for the variable \( F \), the constraint in equation 1.2 is decomposed into primitive constraints:

\[ C = [0, 10] \land (t_1 = 1.8 \times C) \land (F = t_1 + 32) \]  \hspace{1cm} (1.3)

where \( t_1 \) is a temporary variable, initialized with the interval \([-\infty, \infty]\). New intervals for the variables are computed using interval narrowing operations. The narrowing operation for the constraint \( t_1 = 1.8 \times C \) results in an interval \([0, 18]\) for the variable \( t_1 \). The constraint \( F = t_1 + 32 \) yields the interval \([32, 50]\) for the variable \( F \). When \( C \) is in the interval \([-\infty, \infty]\), then necessarily the interval for \( F \) becomes \([-\infty, \infty]\) and vice versa. The details of the narrowing as applied to different operators and predicates will be presented in Section 2.5.
Having explained the basic characteristics of narrowing and constraints, one can now state the fundamental principles governing their usage:

1. If there are any solutions to a problem specified using interval variables, then the solutions must lie within the bounds specified by the output variables, and

2. There are no solutions outside the bounds specified by the output variables.

An important characteristic of interval variables is that arithmetic operations are performed in accordance to the above principles.

1.3.3 Choice and Split Constructs

Having explained the implicit narrowing and the notion of constraint, I now briefly describe the choice construct. That non-deterministic construct is utilized to create a search tree that is then traversed using the commands under the scope of the choice statement. In a sequential implementation, the first of the choices is chosen and the remaining ones placed onto a stack. When the current execution path leads to a failure, or if more solutions are desired, the program execution continues with the remaining choices.

The choice statement implicitly triggers a depth-first traversal of the search tree. The program shown in Figure 1.1 uses the choice construct to compute the temperature in Fahrenheit given various choices for the Celsius variable. The results are also shown in the figure.

The split construct can be viewed as a more general version of choice. Let $L(x)$ and $U(x)$ denote the current lower and upper bounds of an interval variable $x$. Then

\[
\text{split } x \quad \langle\text{splitting instructions}\rangle
\]

repeatedly splits the variable $x$ as stipulated by the $\langle\text{splitting instructions}\rangle$ until leaves
main()
{
    var C, F: interval;
    {
        choice {
            \{C = 10\}; \{C = 50\}; \{C = 100\}
        };
        F = 1.8 * C + 32;
        write C, F
    }
}

Results:
Solution #1: C = 10, F = 50
Solution #2: C = 50, F = 122
Solution #3: C = 100, F = 212

Figure 1.1: The Fahrenheit-Celsius example using the choice construct.

are found. By a leaf, it is meant an interval variable whose \( L(x) = U(x) \). In the case of integer variables, the check for a leaf is obvious, whereas in the case of floating point variables, a precision level \( \epsilon \) should be specified for verifying the predicate \( U(x) - L(x) \leq \epsilon \).

The <splitting instructions> for integer interval variables can be specified as shown in equations 1.4 and 1.5 below. In the former case, the domain the interval is split into two parts, the so-called first and rest components. In the latter case called a middle split, the interval is also subdivided into two parts whose interval ranges are approximately the same (for example, the interval [1, 4] is split into [1, 2] and [3, 4].)

\[
\text{split } x \ (L(x), L(x)) \ (1 + L(x), U(x)) \tag{1.4}
\]

\[
\text{split } x \ (L(x), (L(x) + U(x))/2) \ (1 + (L(x) + U(x))/2, U(x)) \tag{1.5}
\]
```c
main()
{
  var C, F: interval;
  { bind C [0, 10];
    int C;
    F = 1.8 * C + 32;
    split C (L(C), L(C)) (1 + L(C), U(C));
    /* Split according to the first and rest */
    write C, F
  }
}
```

Figure 1.2: The Fahrenheit-Celsius example using the `split` construct.

The program shown in Figure 1.2 uses the `split` construct to subdivide the domain of the variable `C` in the `first and rest` splitting mode. The results in this case are shown in Figure 1.4a with the shaded nodes representing the solutions. However, if the `split` construct

```
split C (L(C), (L(C) + U(C))/2) (1 + (L(C) + U(C))/2, U(C))
```

is used instead, the results will be as shown in Figure 1.4b.

It should be mentioned that there are two ways to implement non-deterministic programs containing the `choice` and `split` statements. In a sequential implementation, the alternatives of a choice statement are pushed onto a stack. When a failure point occurs, or if more solutions are desired, one of the alternatives is popped from the stack and execution continues. In a parallel implementation, all the alternatives are explored in parallel.

In shared memory MIMD computers, the non-deterministic statements are explored in parallel and the narrowing operations are performed sequentially. As it will be seen in Chapter 7, the SIMD implementation is deterministic, in the sense that only one
branch of the choice tree is explored and the failure or success of the narrowing are then communicated to the caller.

1.3.4 Enumeration

Enumeration is utilized when the language user wishes to explore all the possible values within the range of one of more interval variables. The enumeration of a set of variables \( \{x_1, x_2, \ldots, x_n\} \) involves the following selections:

- the ordering \( O \) for enumerating the variables,
- the strategy \( S \) by which a given variable should be split,
- the mode \( M \) for traversing the tree constructed using the ordering \( O \) and whose subtrees have been generated by the split strategy \( S \).

Therefore, enumeration is achieved by specifying the triplet \( <O, S, M> \).

In a program using the proposed language, the ordering \( O \) is that specified by the user, say, in a \textit{write} statement. As seen in the subsection 3.9 on the \textit{split} construct, \( S \) can, for example, specify \textit{first and rest} or \textit{middle} type of interval subdivisions. The mode \( M \) for traversing the tree generated by \( O \) and \( S \) can be, for example, \textit{depth-first}, \textit{breadth-first}, or others that will be discussed in detail in Section 3.9.

Consider the triplet \( <O = \{C, F\}, S = \text{first and rest}, M = \text{depth-first}> \). The enumeration constructs are as specified in the program shown in Figure 1.3. The results of the program appear in Figure 1.4a.

Alternatively, the triplet \( <O = \{C, F\}, S = \text{middle}, M = \text{depth-first}> \) corresponds to the program generating the results shown in Figure 1.4b.

The enumeration process offers a great potential for parallelism. The solutions for the two subproblems that result when a variable is split can be explored in par-
main()
{
    var C, F: interval;
    {
        DFMODE;  /* Depth-first traversal for selecting the variable to split */
        FRSPLIT;  /* Split an interval into first and rest */
        bind C [0, 10];
        int C;
        F = 1.8 * C + 32;
        write C, F
    }
}

Figure 1.3: The program showing Fahrenheit and Celsius relationship.

parallel using shared memory MIMD machines or data parallel SIMD machines. This is particularly advantageous in the case of middle splitting.

1.3.5 Floating Point Variables

In the above descriptions we purposely avoided the enumeration of floating point variables since the splitting using the first and rest branching is not applicable in the floating point case. If a variable is not constrained to be an integer (like the variable $F$ in the above examples), splitting the domain of the variable (using middle split) is carried out until the bounds are within a required precision. For example, the precision is specified as follows:

\[
\text{PRECISION 0.00005.}
\]

When the width of the interval’s domain is less than or equal to the specified precision, i.e., $U(x) - L(x) \leq \text{precision}$, the interval is not split any further, and the only inference we can make is that if a solution exists, it will be within the interval bounds.
Figure 1.4: Enumeration search tree using depth-first strategy.
1.4 Organization

This subsection describes in detail the structure of the thesis and summarizes how the goals in Section 1.2 are achieved.

The basic operations on intervals and the properties of a narrowing function are described in Chapter 2. The proposed language is presented in Chapter 3 embodying the attributes from both imperative and constraint logic programming languages. The constraints in the language are checked for satisfiability using the already explained interval narrowing techniques.

The language’s operational and denotational semantics are described in Chapter 5. Chapter 6 presents the data structures used in the sequential implementation of the language. Parallelism is exploited in multiple ways. The algorithms in Chapter 7 use data parallel techniques applicable to a SIMD computer. Chapter 8 describes how the non-deterministic constructs are explored in parallel on a shared memory MIMD machine. The split operations also fall in this category.

It was felt that the most pedagogical way to present the examples is after the language description, and before presenting the methods utilized in the implementations. This is justified since the examples in Chapter 4 will be reconsidered in the various parallel versions of the language.

A brief presentation of the nature of the examples is described in the sequel. A detailed representation of floating point variables is given in Section 2.1 and an example of the use of floating point variables is provided in Section 4.3 where hidden Markov models are studied.

Three computationally hard problems are studied in this work: 3-SAT, scheduling, and constraint satisfaction problems.
The 3-SAT problem consists of finding the satisfiability of a boolean expression expressed in clausal form. Boolean variables can be conveniently described as integers in the interval $[0, 1]$. It is shown that the Davis-Putnam method for determining satisfiability can be improved by the use of interval variables.

In the job-shop scheduling problem, a set of jobs are to be processed using a set of machines. Each job consists of an ordered sequence of operations. The problem is to find the minimum time span to process all operations in the given order. The 3-SAT and job-shop scheduling problems are described in Section 4.4. We also show in Section 4.4.3 that the class of constraint satisfaction problems (CSP) is a subset of the problems that the proposed language can handle.

Finally, Chapter 9 describes a graphical interface designed using the Java programming language. The interface consists of rolldown menus, text boxes, and button actions. These components aid the user in constructing a program in the proposed language with minimal effort. Debugging and profiling support is also provided.

1.5 Historical Background and Related Work

In this section I briefly discuss the historical background behind the various themes that motivated this dissertation, namely: the origins of non-determinism, interval arithmetic, logic programming, and constraint logic programming.

1.5.1 Non-Determinism

Non-deterministic algorithms were originally referred to as backtrack programs in early literature. The word “backtrack” was named by D. H. Lehmer of the University of California at Berkeley. Backtrack has been discovered independently by many researchers. Golumb’s was the first attempt to formulate the scope and methods of backtrack pro-
gramming. He formulates the backtrack problem as follows: Determine the vector $(x_1, x_2, \ldots, x_n)$ from the cartesian product space $X_1 \times X_2 \times \cdots \times X_n$, which maximizes the criterion function $f(x_1, x_2, \ldots, x_n)$. Let $M_i$ be the number of distinct values in $X_i$, and let $M = M_1 \times M_2 \times \cdots \times M_n$. The brute-force approach is to evaluate the criterion function with each of the $M$ possible sample vectors, and determine the vector which produces the desired value.

The backtrack algorithm is designed to yield the same answer as the brute-force approach in less than $M$ trials. In the backtrack approach, the sample vector is built one component at a time and tested whether the partial vector has a chance of success. If the partial vector $(x_1, x_2, \ldots, \omega)$ is already determined to be suboptimal, then $M/(M_1 \times M_2)$ possible test vectors are immediately ruled out.

In 1967, Floyd proposed language features to express non-deterministic algorithms. Floyd views non-deterministic algorithms as conceptual devices to simplify the design of backtracking algorithms. In Floyd’s approach, non-deterministic algorithms resemble conventional algorithms, except that

1. one may use a multiple-valued function, $\text{choice}(X)$, whose values are positive integers less than or equal to $X$, and
2. all points of termination are labeled either as success or failure.

A survey of non-deterministic programming features can be found in the paper by Cohen (COHE 79). The basic non-deterministic construct is the following:

$$\text{choice among } x \leftarrow 1 \text{ to } n.$$ 

In Floyd’s sense, this can be viewed as the statement $x \leftarrow \text{choice}(n)$.

Apt and Schaefer (ApSc 97) proposed language constructs to augment the expressive power of imperative programming. Some of the constructs developed are the
non-deterministic choice (the OR statement), a parameterized non-deterministic choice (the SOME statement), as well as the indicators for failure and success.

As to the complexity of non-deterministic programs, Knuth (1975) provides the methodology for their analyses. Purdom and Brown (PuBr 85) explain in detail the average time calculation of a backtracking algorithm by estimating the size and the number of nodes of a backtrack tree using probabilistic analysis.

1.5.2 Interval Arithmetic

The present day interest in interval arithmetic stems from the work of R. E. Moore (MOOR 66). Two earlier papers by W. Warmus (1956) and T. Sunaga (1958) reported the uses of interval arithmetic. Cleary (CLEA 87) introduced a relational form of interval arithmetic in logic programming.

Computers use fixed length arithmetic called floating point arithmetic. Real numbers in this arithmetic are approximated by a subset of the real numbers called the machine representable numbers. Two types of errors occur as a result of this representation: 1) when real valued input data is approximated by the machine number, and 2) when the intermediate results of computation are approximated by the machine numbers.

The idea of carrying error bounds along with the underlying computation dates back to the fifties when Dwyer (DWYE 51) explains the motivation for the use of interval arithmetic and provides a good introduction to the issues raised in approximate computation. He used the term range number to denote intervals. Gibb (GIBB 61) describes procedures for interval (range) arithmetic operations. The procedures for range addition, subtraction, multiplication, division and square are described in detail in that
paper. A first introduction of interval arithmetic for automatic error analysis is provided by Collins (COLL 60). An operational interval arithmetic semantics was described by Boche (BOCH 63).

The essence of interval arithmetic is that: i) closed real intervals are substituted for computational data in which there is uncertainty, ii) each such interval is represented by a pair of floating-point numbers that are known to be lower and upper bounds for the true (unknown) values of its corresponding datum, and iii) in place of each arithmetic operation in a numerical algorithm, a corresponding interval arithmetic operation computes the interval containing all possible results.

Three significant barriers to the use of interval arithmetic have been: i) the difficulty of obtaining directed roundings for interval endpoint computations, ii) the increased overhead in both computation and storage required by interval arithmetic, and iii) the lack of high-level language support for interval data types.

The first of these obstacles was taken care of with the introduction and implementation of IEEE/ANSI binary floating point standard, which requires provision of directed rounding modes in addition to the round-to-nearest default mode. Problems of computation and storage are less severe with the present day personal computers. A unified framework for interval arithmetic and interval constraints is presented in (HiEmWu 98). Optimization methods using interval methods can be found in the works of Hansen (HANS 92) and in the language NUMERICA (HeMiDe 97). A substantial collection of references on interval arithmetic is available at the URL cs.dep.edu/interval-comp.
1.5.3 Constraint Programming Languages

Perhaps the earliest usage of constraints in computer languages dates back to the work of Borning in his language THINGLAB (BORN 81). Borning specified constraints in the layout of graphical displays. Sussman and Steele’s language (SuSt 80) also represents one of the first usages of constraints and they might have coined the term known as constant propagation. It consists in determining the values of a variable in a formula as function of the known (bound) values of other variables, regardless of their position in the left or right hand part of an equation. This operation can be cascaded to determine further values of variables in other equations, once one of them becomes known. More powerful techniques are required in the case of constraints involving cyclic interdependencies. Relaxation methods (BORN 81) and linear and nonlinear equation solvers are employed in constraint languages to solve complex constraints.

The design and implementation of a constraint imperative programming language, Kaleidoscope’90, has been discussed in the paper by Freeman-Benson and Borning. (FrBo 92). The language combines some of the concepts of imperative and constraint programming methods. However, their language is aimed for usage in graphical interfaces where priorities are specified among various constraints so that only the ones that are strictly required need to be satisfied, the others being only partially satisfied.

A constraint programming language based on non-deterministic LISP has been developed by Siskind and McAllester (SiMc 93). It is called Scream and it is an extension of COMMON LISP with nondeterministic primitives. The constraint system solves both numeric and non-numeric constraints using techniques such as local propagation, Boolean constraint propagation, forward checking, etc.
1.5.4 Logic Programming

The article by Cohen (COHE 88) presents a historical perspective of the origins of logic programming languages. The birth of logic programming can be viewed as a confluence from the works of Alain Colmerauer and Robert Kowalski (KOWA 74). Colmerauer’s contribution came from his work in language processing, whereas Kowalski’s originated from his work in logic and theorem proving. The logic programming language PROLOG is especially suited to applications involving pattern matching, backtrack searching, or incomplete information. Kowalski illustrates the logic programming paradigm by the dictum:

$$\text{Logic Program} = \text{Logic} + \text{Control}$$

In the above, Logic refers to the facts and rules specifying the inferences that the user wishes to be held. The Control part has to do with the ordering and annotations specified by the user to achieve an efficient processing of the inferences.

Non-determinism is an integral part of logic programming and it allows searching in trees whose leaves indicate failure or success of the user-specified inferences. The concept of unification is also central to the logic programming language Prolog. Unification is one of the ways in which constructs in various domains are compared. Unification in Prolog deals with equality of trees, called terms, whose leaves may contain variables or constants. When two trees can be matched, unification yields a list of the most general bindings for the variables satisfying the tree equality.

1.5.5 Constraint Logic Programming (CLP) Languages

The introduction of constraints into logic programming languages can be traced to the design and implementation of Prolog II by Colmerauer (COLM 82). The domain of the
language is that of infinite trees. The unification of such trees is performed by solving systems of equations involving tree-valued variables. Backtracking occurs whenever the equations (constraints) have no solution. Disequations were also introduced in the language using the predicate $\text{dif}(T_1, T_2)$, where $T_1$ and/or $T_2$ can be unbound variables. In the constraints terminology, $\text{dif}$ adds the disequation $T_1 \neq T_2$ to the current set of constraints. The predicate checks if two infinite trees are not equal.

The fundamental concept in CLP languages is to use constraint solving in addition to unification. In the CLP paradigm, Kowalski’s dictum is revised to read

$$\text{Constraint Logic Program} = \text{Constraints} + \text{Logic} + \text{Control}$$

where $\text{Constraints}$ are essentially built-in predicates whose satisfiability is, like unification, determined by efficient specialized programs.

Jaffar and Lassez provided a semantic foundation (JaLa 86) and developed the language CLP($\mathbb{R}$) (JMSY 92). A particularly interesting feature of this language is a $\text{delay/wake up}$ mechanism. The delay condition applies to nonlinear equations, and the wakeup condition occurs when the delayed equation becomes linear as a result of local propagations. CLP($\mathbb{R}$) handles constraints over real numbers and should be considered as an approximate implementation for reasoning in real arithmetic, because of its use of floating point representation for real numbers.

Colmerauer and his group developed an extension of Prolog called Prolog III (COLM 90). Prolog III solves constraints over Booleans, rational numbers, and lists. The built-in predicates encompass capabilities for solving systems of linear equations, inequations (greater, smaller, etc.) and disequalities (not-equal) within the domain of rationals.

Other CLP languages are briefly mentioned for the sake of completeness. The
language CHIP (DiHeSiAg 88) was developed and employed to solve many combinatorial search problems. The language also deals with integer constraints. CAL (SaAi 89) focuses on solving nonlinear equations over complex numbers. CLP(Σ*) deals with constraints about regular expressions over some alphabet and is focused on string manipulation problems. Another constraint language, 2lp, has been developed to bridge the gap between operations research problems and constraint logic programming (AtMcTr 93). This language provides explicit non-deterministic constructs as well as constructs for solving linear programming problems. Marriott and Stuckey (MaSt 98), in their book, provide a treatise on programming with constraints. The book covers topics ranging from constraint-solving techniques to programming methodologies for constraint programming languages.

The paradigm of concurrent CLP is based on don’t care non-determinism and involves selecting one particular branch of OR-parallel constructs and neglecting the others. In recent years, there has been significant research work done in this area. Since our parallel versions are based in don’t know non-determinism and in narrowing, we will not delve further into this matter. Good references on the topic can be found in (KeCo 94).

**Interval Constraints**

A major new extension — that of introducing the domain of Intervals — was incorporated into Prolog IV, again designed by Colmerauer and his group at the University of Marseilles in France. This extension opens new vistas for CLP languages manipulating floating-point variables, a domain that has been practically neglected by designers of symbolic languages such as Lisp and Prolog. By neglect it is meant disregarding the inherent lack of accuracy when checking the equality of two floating-point numbers. As
a result of approximations due to round-off errors, the equality of two floating-point variables only makes sense by specifying a small quantity within which the two variables may differ. This of course may be sufficient when dealing with certain engineering problems, but becomes dangerous when one wishes to achieve a rigorous, logic-based definition of equality. The constraint language CLP(BNR) (BeOl 92) was probably the first one to extend Prolog to cover intervals representing reals, integers, and Booleans.

We have already seen that interval variables are defined by specifying lower and upper bounds for the values of those variables. Any value between those bounds is considered to be a possible value for the variable. Interval variables allow their users to rigorously perform arithmetic operations and comparisons using the now widely spread IEEE standards for handling floating-point numbers. Actually, the lower and upper bounds of each variable are considered as very large integers. We have also seen that there are symbolic representations for the bounds specifying minus or plus infinity, and those bounds are considered when arithmetic operations are performed.

Therefore the introduction of interval variables extends in a very elegant manner the domains dealt by the first CLP languages. Basically, real numbers can be specified by interval variables whose bounds are finite numbers (large integers specified by floating-point representations) Finite domains are also treated as integer interval variables and Booleans are a special case of integer interval variables in which the bounds are 0 and 1.

This spectrum of types of variables is one of the novel features of Prolog IV and a great deal of thought has been given by its designer to provide a flexible and logical interaction between various types of variables. For example, interval variables specify reals, and finite domain variables, including Booleans, are also specified using integer interval variables. Systems of linear equations, inequations and disequalities involve
rational variables; equalities, inequalities and disequalities are also applicable to interval variables; finally, as in Prolog II, equalities and disequalities are applicable to the infinite tree-type variables. It is a challenge to a language designer to harmoniously integrate the various domains and predicates, as it is done in Prolog IV.

It should also be mentioned that the incorporation of intervals further extended the capabilities of the language to solve Operations Research problems. It has enabled the practical solution of scheduling problems that previously required hours of computations. In addition, interval variables allow proving propositions asserting that there exist no solutions to systems of equations or inequations in which variables are required to have values within certain ranges. When confronted with such situation, a processor for an interval constraint language simply replies "no", thus providing evidence that no solution exists (i.e., assuming that the interpreter is proven correct.)

1.5.6 Previous Brandeis Research in Parallel Versions of Logic Programming Languages

Parallel systems implementing logic programming languages have been explored from two angles: explicit parallelism and implicit parallelism. In explicit parallelism, annotations are provided by the user to specify tasks to be performed in parallel (e.g., STRAND, GHC, KL1, etc.) Implicit parallel systems attempt to parallelize the logic programs without annotations.

There are two underlying sources of implicit parallelism in logic programming languages, OR-parallelism and AND-parallelism. OR-parallelism consists of the simultaneous exploration of several choices that would be computed successively in a sequential execution. AND-parallelism consists of the simultaneous computation of several goals.
The Aurora system and the Muse model are OR-parallel implementations (AlKa 90). The Aurora system is based on Warren's model (WARR 86). An Aurora prototype, based on SICStus Prolog, was implemented on several shared-memory multiprocessors. Mudambi's thesis (MUDA 92) investigates the performance of Aurora and MUSE OR-parallel systems in non-uniform memory access architectures. In his thesis, Smith (SMIT 94) presents a system called MultiLog which explores data Or-parallel logic programs. The disj operator is used to partially replace backtracking with multiple environments which are explored in parallel.

In the context of this dissertation, there are two modes of parallelism that are exploited: OR-parallelism using shared-memory architectures, and narrowing-parallelism using data parallel architectures. The parallel choice statement in this work bears close resemblance to the OR-parallel model in logic programming. However, choice parallelism becomes much simpler in the case where unification is completely replaced by equality constraints involving single interval variables.

In constraint logic programming, constraint satisfiability is the counterpart of unification in logic programming. There have been few efforts to parallelize unification, and, to our knowledge, there has been no known effort to parallelize constraint satisfiability in various domains as it exists in Prolog IV. The work by Ju (JU 98) represents an effort to parallelize constraint satisfiability in the case of intervals, using shared memory architectures. Our data parallel approach also aims at dealing with interval constraint satisfiability.

A summary of the work in parallelism done at Brandeis on logic programming and constraint logic programming can be summarized by the table shown in Figure 1.5.
<table>
<thead>
<tr>
<th></th>
<th>Shared-memory</th>
<th>Data parallel</th>
<th>Constraint satisfiability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Tree Unification</td>
</tr>
<tr>
<td>Mudambi</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Smith</td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>Ju</td>
<td>*1</td>
<td>*1</td>
<td>*</td>
</tr>
<tr>
<td>Kalathur</td>
<td>*2</td>
<td>*1</td>
<td>*</td>
</tr>
</tbody>
</table>

1 Parallelization of narrowing.

2 Parallelization of choice.

Figure 1.5: Brandeis research in parallel logic programming and constraint logic programming.

1.5.7 Final Remarks

It is appropriate to end this section by placing the proposed language in the context of other language paradigms, and their dictums. In the specification of the Pascal programming language, Wirth summarizes its paradigm as:

*Imperative Program = Data Structures + Algorithms.*

By combining the imperative language paradigm with that of the proposed language leads to:

*Programs = Intervals + Constraints + Control*

In the above, *Intervals* are equated to *Data Structures*, *Constraints* to Constraint Logic Programming, and the non-deterministic *Control* to Logic Programming.
Chapter 2

Basic Operations on Intervals

In this chapter, we rigorously define the concept of intervals, the intersection operation, and narrowing involving primitive constraints. The various number systems (integers, floating points, reals, and intervals including booleans) are presented in Section 2.1. Interval arithmetic is explained in detail in Sections 2.2 and 2.3. Open and closed intervals are discussed in Section 2.4. Finally, the narrowing operation and its properties are presented in Section 2.5.

2.1 Numbers

In this section, we examine some of the different number systems before studying the interval number system in detail in Section 2.2. The following summarizes the various domains for which interval variables can be used.

- **Integers**

  The set of all integers is denoted by $\mathbb{Z}$, where $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. 
• **Rational Numbers**

Rational numbers are an extension of the integer numbers. The set of all rationals is denoted by \( \mathbb{Q} \), where \( \mathbb{Q} = \{ \ldots, -1, -\frac{1}{2}, -\frac{1}{3}, \ldots, 0, \ldots, \frac{1}{3}, \frac{1}{2}, 1, \ldots \} \).

• **Real Numbers**

Real numbers are an extension of the rational numbers. The set of real numbers is denoted by \( \mathbb{R} \). Each real number can be specified by a converging infinite sequence of rational numbers. For example, \( \sqrt{2} = (\frac{1}{1}, \frac{14}{10}, \frac{141}{100}, \frac{1414}{1000}, \frac{14142}{10000}, \ldots) \).

• **Floating Point Numbers**

Floating point numbers (machine representable numbers) are useful in approximating real numbers. The set of floating point numbers is denoted by \( \mathcal{F} \). A floating point number can be written as \( a * b^c \), where \( a, b, c \) are in \( \mathbb{Z} \), the set of integers.

Formally, the system \( \mathcal{F}[b, A, m..M] \) represents all floating point numbers

\[
d_0d_1d_2\ldots d_{A-1} * b^c, \text{ where } 0 \leq d_k < b, m \leq c \leq M.
\]

The set of floating point numbers with \( b = 2 \) is used for computer implementations; \( a \) and \( c \) are usually represented using a fixed number of bits. In the IEEE 754 64-bit double precision representation, \( a \) is stored in 53 bits (one bit for the sign and fifty-two bits for the magnitude) and \( c \) is stored in 11 bits.

**Infinity**

The floating point number system is augmented with two special symbols: \( \infty \) and \( -\infty \). The symbol \( \infty \) (\( -\infty \)) represents a real number which is too large (small) to be described with \( \mathcal{F} \). A check is performed to test for these symbols prior to performing an arithmetic operation.
Rounding

The IEEE floating point operations approximate the corresponding operations with real numbers. One can differentiate three types of rounding; when the result of a floating point operation is not a floating point number, the result is rounded to the nearest floating point number. Consider the following addition operation:

\[ 1 \times 10^0 + 1 \times 10^3 = 1001 \times 10^0. \]

Both \( 1 \times 10^0 \) and \( 1 \times 10^3 \) are members of \( \mathcal{F}[10, 3, -9 \ldots 9] \), but \( 1001 \times 10^0 \) is not. In the “rounding to nearest” mode, the result will be \( 1 \times 10^3 \).

Another form of rounding is called “upward rounding”, where the result is rounded up to a larger floating point number: if the result is positive, it is rounded away from zero. If the result is negative, it is rounded towards zero. In the above example, the result will be \( 101 \times 10^1 \).

Another variation of rounding is called “downward rounding”, where the result is rounded down to a smaller floating point number. If the result is positive, it is rounded towards zero. If the result is negative, it is rounded away from zero. The result for the above example in this case will be \( 1 \times 10^3 \). IEEE 754 requires that the operators \(+, -, \ast, /\), and \( \sqrt{\cdot} \) are rounded to the nearest floating point number.

2.1.1 Interval Arithmetic

Although the floating point computations are simple and efficient, rounding can result in a sequence of floating point computations to diverge from the correct answer. Interval arithmetic guarantees the correct results though it is built from floating point arithmetic by doing the appropriate rounding operations as specified by the IEEE 754 standard.

An interval is specified by two floating point numbers, a lower and upper bound.
The interval \([a, b]\) represents any real number between \(a\) and \(b\). Rather than returning a single floating point number for each operation, the result for each operation is an interval in which the real result is guaranteed to be. For example, \(\pi\) can be represented as the interval \([0.314, 0.315]\). The set of intervals is denoted by \(\mathcal{I}\). The upper bound of an interval \(x\) is denoted by \(u_x\) while \(l_x\) denotes the lower bound. The width of the interval is the difference between the upper and lower bound, \(w_x = u_x - l_x\). Every interval has a non-negative width, i.e., for every \(x\) in \(\mathcal{I}\), \(w_x \geq 0\). An interval has a zero width if it represents a particular real number which happens to coincide with a floating point number. A real number is contained in an interval if for every \(x\) in \(\mathcal{R}\), for every \([a, b]\) in \(\mathcal{I}\), \(x \in [a, b]\) iff \(a \leq x \leq b\).

The essence of interval arithmetic is that:

- Closed, real intervals are substituted for computational data in which there is uncertainty.

- Each such interval is represented by a pair of floating-point numbers that are known to be the lower and upper bounds for the true value of its corresponding data.

- In place of each arithmetic operation in a numerical algorithm, a corresponding interval arithmetic operation computes the interval containing all possible results of performing the original operation on any values taken from the interval operands.

### 2.2 Operations using Intervals

**Definition:** An interval number is an ordered pair of real numbers, \([a, b]\), \(a \leq b\), \(a, b \in \mathcal{R}\), denoting the set \(\{x|a \leq x \leq b\}\). All intervals are closed intervals. The arithmetic operations on the interval numbers, or simply intervals, are presented below.
(MOOR 66):

\[ [a, b] + [c, d] = [a + c, b + d] \]
\[ [a, b] - [c, d] = [a - d, b - c] \]
\[ [a, b] \times [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)] \]
\[ [a, b] \div [c, d] = [a, b] \times [1/d, 1/c], \text{ if } 0 \notin [c, d] \]

The reader should refer to (JU 98) for a detailed treatment of different boundary conditions involving intervals.

### 2.3 The Intersection operator \( \cap \)

**Definition:** The intersection of two intervals, \([a, b] \cap [c, d]\), is the largest interval which is a subset of the two given intervals. If such an interval exists, the operation succeeds and yields the intersecting interval. Otherwise, the operation fails. Figure 2.1 shows the different cases that arise when performing the intersection operation. From the figure, we can conclude that

\[ [a, b] \cap [c, d] = \begin{cases} 
  [\max(a, c), \min(b, d)], & \text{if } \max(a, c) \leq \min(b, d) \\
  \bot, & \text{otherwise}
\end{cases} \]

#### 2.4 Open and Closed Intervals

For every \(a, b \in \mathbb{R}, a \leq b\), the following notations are used for intervals ([ and ] represent closed interval bounds; ( and ) represent open interval bounds):

\[ [a, b] = \{ x \in \mathbb{R} \mid a \leq x \leq b \} \]
\[ (a, b] = \{ x \in \mathbb{R} \mid a < x \leq b \} \]
\[
\begin{align*}
\text{Case} & & \text{Result} \\
[l_x = l_y, u_x = u_y] & & [l_x, u_x] \\
[l_x \leq l_y, u_x \geq u_y] & & [l_y, u_y] \\
[l_x \geq l_y, u_x \leq u_y] & & [l_x, u_x] \\
[l_x \leq l_y, u_x \leq u_y] & & [l_y, u_x] \\
[l_x \geq l_y, u_x \geq u_y] & & [l_x, u_y] \\
u_x < l_y & & \text{fail} \\
l_x > u_y & & \text{fail}
\end{align*}
\]

Figure 2.1: The Intersection operation.
\[ [a, b) = \{ x \in \mathbb{R} \mid a \leq x < b \} \]
\[ (a, b) = \{ x \in \mathbb{R} \mid a < x < b \}. \]

In the constraint language presented, the user can only express closed intervals. Nevertheless, open intervals can be transformed into closed intervals with the introduction of disequality (\( \neq \)) constraints. For example, the constraint \( x \in (a, b] \) can be rewritten as the constraints \( x \in [a, b] \) and \( x \neq a \). In the case where an interval is split in the enumeration phase, open intervals are implicitly used:
\[ [a, b] = [a, m] \cap (m, b], \text{ where } m = \frac{a+b}{2}, \]

where \( m \) represents a floating-point number which is the mid-point between the floating-point numbers \( a \) and \( b \).

### 2.5 Narrowing

At the heart of the constraint solver is the operation called **narrowing**. The narrowing operation takes as input an \( n \)-ary constraint, say, \( f(x_1, x_2, \ldots, x_n) \), and the intervals, \( i_{x_1}, i_{x_2}, \ldots, i_{x_n} \), and computes the new interval bounds, \( i'_{x_1}, i'_{x_2}, \ldots, i'_{x_n} \), of the variables \( x_1, x_2, \ldots, x_n \). For each variable in the constraint, this operation is (usually) the intersection of the current interval with the interval obtained by performing the basic interval arithmetic operation on the constraint rewritten with this variable as the result. For example, consider the constraint \( x + y = z \) and the variable \( y \). The rewritten constraint with \( y \) as the result is \( y = z - x \). The new interval bounds for \( y \) would be \( [l_y, u_y] \cap ([l_z - u_x, u_x] - [l_x, u_x]) \). The resulting intervals could be smaller or remain the same as those before the narrowing operation. It is known that the result of successive narrowing operations will either converge or fail (OlVe 93). A narrow operation can fail because the intersection fails, or in some cases (e.g., the \( \neq \) constraint, which does not
<table>
<thead>
<tr>
<th>Constraint</th>
<th>Result</th>
</tr>
</thead>
</table>
| $x + y = z$ | \[ [l_x, u_x] = [l_x, u_x] \cap \{ [l_z, u_z] - [l_y, u_y] \} \]  
\[ [l_y, u_y] = [l_y, u_y] \cap \{ [l_z, u_z] - [l_x, u_x] \} \]  
\[ [l_z, u_z] = [l_z, u_z] \cap \{ [l_x, u_x] + [l_y, u_y] \} \] |
| $x - y = z$ | equivalent to $y + z = x$ |
| $x \cdot y = z$ | \[ [l_x, u_x] = [l_x, u_x] \cap \{ [l_z, u_z]/[l_y, u_y] \} \]  
\[ [l_y, u_y] = [l_y, u_y] \cap \{ [l_z, u_z]/[l_x, u_x] \} \]  
\[ [l_z, u_z] = [l_z, u_z] \cap \{ [l_x, u_x] * [l_y, u_y] \} \] |
| $x/y = z$ | equivalent to $y * z = x$ |
| $x = y$ | \[ [l_x, u_x] = [l_x, u_x] \cap [l_y, u_y] \]  
\[ [l_y, u_y] = [l_x, u_x] \] |
| $x \neq y$ | fail if $l_x = l_y = u_x = u_y$  
\[ l_y = l_y + 1 \text{ if } l_x = u_x = l_y \]  
\[ u_y = u_y - 1 \text{ if } l_x = u_x = u_y \]  
\[ l_x = l_x + 1 \text{ if } l_y = u_y = u_x \]  
\[ u_x = u_x - 1 \text{ if } l_y = u_y = u_x \] |
| $x \leq y$ | equivalent to $x - y = [-\infty, 0]$ |
| $x \geq y$ | equivalent to $x - y = [0, \infty]$ |
| $x = int \ y$ | $u_x = u_y$ if $u_y = \infty$  
\[ u_x = [u_y], \text{ otherwise;} \]  
\[ l_x = l_y \text{ if } l_y = -\infty \]  
\[ l_x = [l_y], \text{ otherwise;} \] |
| $x = bool \ y$ | $x = int \ y \land [l_x, u_y] = [l_x, u_y] \cap [0, 1]$ |

Figure 2.2: The Narrow Operations.
use the intersection operation) the narrow operation fails because the intervals do not satisfy the constraint. Figure 2.2 shows some of the narrowing operation results. Similar narrow operations can be defined for $x^y = z$, $y = \sin(x)$, $y = \cos(x)$, $y = \tan(x)$, etc.

**Example:** Consider the constraint $x \ast y = z$ with the initial intervals $[1,10]$, $[5,15]$, and $[4,8]$ for the variables $x$, $y$, and $z$, respectively. Narrowing yields the following intervals:

$$x = [1, 10] \cap [4,8]/[5, 15] = [1, 10] \cap [0.26667, 1.6] = [1, 1.6]$$

$$y = [5, 15] \cap [4,8]/[1, 1.6] = [5, 15] \cap [2.5, 8] = [5, 8]$$

$$z = [4, 8] \cap [1, 1.6] \ast [5, 8] = [4, 8] \cap [5, 12.8] = [5, 8]$$

A major property of the narrowing algorithm is stated below (OLVe 93):

- If a solution exists, it will be within the narrowed interval.

- There are **no** solutions outside the interval.

The properties of the narrowing operation are described in Section 2.5.1. The correctness of the narrowing operations ensure that no solutions are lost in the narrowing phase.

**Unions of Intervals:** In some cases, the result of the narrowing operation is a union of intervals. When the narrowing operation results in unions of intervals, the union $[x_1, y_1] \cup [x_2, y_2] \cup \ldots \cup [x_n, y_n]$ is approximated by the interval $[x_i, y_j]$, where

$$x_i = \min(x_1, x_2, \ldots, x_n) \quad \text{and} \quad y_j = \max(x_1, x_2, \ldots, y_n).$$

It will be seen that this reduction of union of intervals into a single interval does not alter the capabilities of finding solutions by non-deterministic searches. This situation is illustrated in the following two cases:

1. $x \ast y = 1$, $x \in [-1, 1]$.

   Ideally, narrowing should yield $y \in [-\infty, -1] \cup [1, \infty]$. (Figure 2.4)
\begin{center}
\begin{tabular}{c c c c c}
  \hline
  $x \cdot y = z$ & $x = z/y$ & $y = z/x$ & $x \in$ & $y \in$ & $z \in$\\
  \hline
  $\star$ & $[0.27, 1.6]$ & $[5, 15]$ & $[4, 8]$ & & \\
  $\star$ & $[1, 1.6]$ & $[2.5, 8]$ & $[4, 8]$ & & \\
  $\star$ & $[1, 1.6]$ & $[5, 8]$ & $[4, 8]$ & & \\
  $\star$ & $[1, 1.6]$ & $[5, 8]$ & $[5, 12.8]$ & & \\
  $\star$ & $[1, 1.6]$ & $[5, 8]$ & $[5, 8]$ & & \\
  \hline
\end{tabular}
\end{center}

Figure 2.3: Narrowing steps for the constraint $x \cdot y = z$

\begin{center}
\includegraphics[width=0.5\textwidth]{figure23.png}
\end{center}

Figure 2.4: Intervals for variables in $x \cdot y = 1$
2. $x = y^2, x \in [4, 9]$.

In this case, narrowing should yield $y \in [-3, -2] \cup [2, 3]$. (Figure 2.5)

The examples in Figures 2.4 and 2.5 illustrate that narrowing can result in an union of intervals. If this occurs repeatedly, there is a danger of yielding combinatorially explosive computation. Interval operations applicable to say, two sets of unions, will have to consider elements in the cartesian product of the two sets. In the above examples, it is practical to retain the intervals $[-\infty, \infty]$ and $[-3, 3]$ respectively for the variable $y$. Space can be saved by keeping the largest interval enclosing the interval unions. The narrowing of succeeding operations may reduce the size of the interval, thus eliminating the possibility for a disjoint union.

The narrowing algorithm determines the largest interval for each variable ap-
pearing in a set of constraints if the result of the narrowing operation is a disjoint union of intervals. In reality, each member of the union of intervals can be a candidate for a solution. Even if the union of intervals remain at the conclusion of the narrow operations, the intervals are split during the enumeration procedure. The splitting algorithm ensures that no solutions are lost during this process. The variable $y$ can be split into $[-\infty, 0]$ and $(0, \infty)$ for the constraints shown in Figure 2.4. In this case, the narrowing operation reduces the interval for $y$ to $[-\infty, -1]$ and $[1, \infty]$. Similarly, for the constraints shown in Figure 2.5, the variable $y$ can be split into $[-3, 0]$ and $(0, 3]$. In this case, narrowing reduces the interval for $y$ to $[-3, -2]$ and $[2, 3]$.

### 2.5.1 Properties of the Narrowing Function

Let $\eta$ be any $n$-ary relation (e.g., $+, -, \ast, \min, \max, \ldots$) on $\mathbb{R}$. Note that $\mathbb{R}$ includes the symbols $\{-\infty, \infty\}$. Let $\mathcal{I}$ be the set of all intervals. Let $\vec{i}$ denote the interval vector $(i_{x_1}, i_{x_2}, \ldots, i_{x_n})$. Before defining the narrowing function $N_\eta$ of the $n$-ary relation $\eta$, let us define the approximation function of $\eta$, $\varphi$, as the smallest interval vector containing $\eta$. The approximation function has the following properties.

- Let $\eta$ and $\eta'$ be two $n$-ary relations on $\mathbb{R}$. Then, $\varphi(\eta') \subset \varphi(\eta)$ if $\eta' \subset \eta$. *(monotonicity)*

- $\varphi(\varphi(\eta)) = \varphi(\eta)$. *(idempotence)*

**Narrowing:** Given an $n$-ary relation $\eta$ and an interval vector $\vec{i}$, the narrowing function of $\eta$,

$$ N_\eta : \mathcal{I}^n \rightarrow \mathcal{I}^n $$

applied to $\vec{i}$ is the smallest interval vector containing $\vec{i} \cap \eta$, i.e.,

$$ N_\eta(\vec{i}) = \varphi(\vec{i} \cap \eta). $$
The narrowing function has the following properties:

1. \( \tilde{i} \cap \eta = N_\eta(\tilde{i}) \cap \eta \). (correctness)

2. \( N_\eta(\tilde{i}) \subseteq \tilde{i} \) (contractance)

3. \( N_\eta(\tilde{i}) \subseteq N_\eta(\tilde{j}) \) if \( \tilde{i} \subseteq \tilde{j} \). (monotonicity)

4. \( N_\eta(N_\eta(\tilde{i})) = N_\eta(\tilde{i}) \). (idempotence)

The contractance and idempotence properties ensure that the narrowing operation always converges to a fixed point.

Since floating-point numbers only can be represented on a computer, the arithmetic operations involving intervals over \( \mathbb{R} \) are not sound. The IEEE standards for floating-point arithmetic are employed to achieve sound arithmetic operations (HICK 94; JU 98). These standards ensure that the intervals are rounded off in the appropriate way so that no solutions are missed (MULL 97).

### 2.5.2 Increasing the Efficiency of Narrowing

The narrowing algorithm over a set of constraints may cause a cyclic behavior which leads to a slow convergence (LhGoRu 94). The following observations can be drawn by studying the cyclic behavior:

1. A large number of narrowing operations carried out during a cycle are unnecessary.

2. Many other narrowing operations can be removed from cycles and performed only once after the cycles have been processed.

An interval narrowing algorithm is proposed in (LhGoRu 94) for identifying and simplifying such cyclic behavior dynamically. Consider the following set of constraints:
1. \( y = x \),
2. \( y = 2 \times x \),
3. \( y = 10 \times x \),
4. \( z = 3 \times y \)

Let the initial domains for the variables \( x, y, \) and \( z \) be \([0, 10], [-\infty, \infty]\), and \([-\infty, \infty]\).

The narrowing operations are listed in Figure 2.6. In that figure, the constraint \( y = 2 \times x \) results in a weaker reduction for the variable \( x \) than the constraint \( y = 10 \times x \). Narrowing the constraint \( z = 3 \times y \) only affects the interval for variable \( z \). The algorithm proposed in (LhGoRu 94) replaces \( n \) iterations of constraints \((1, 2, 3, 4)\) by \( n \) iterations of \((1, 3)\) followed by one iteration of \((4)\).
<table>
<thead>
<tr>
<th>$y = x$</th>
<th>$y = 2 \times x$</th>
<th>$y = 10 \times x$</th>
<th>$z = 3 \times y$</th>
<th>$x$</th>
<th>$y$</th>
<th>$z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>[0, 10]</td>
<td>$[-\infty, \infty]$</td>
<td>$[-\infty, \infty]$</td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td></td>
<td></td>
<td>[0, 10]</td>
<td>[0, 10]</td>
<td>$[-\infty, \infty]$</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td>[0, 5]</td>
<td>[0, 10]</td>
<td>$[-\infty, \infty]$</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td></td>
<td></td>
<td>[0, 1]</td>
<td>[0, 10]</td>
<td>$[-\infty, \infty]$</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>*</td>
<td></td>
<td>[0, 1]</td>
<td>[0, 10]</td>
<td>[0, 30]</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td></td>
<td></td>
<td>[0, 1]</td>
<td>[0, 1]</td>
<td>[0, 30]</td>
</tr>
<tr>
<td>6</td>
<td>*</td>
<td></td>
<td></td>
<td>[0, 0.5]</td>
<td>[0, 1]</td>
<td>[0, 30]</td>
</tr>
<tr>
<td>7</td>
<td>*</td>
<td></td>
<td></td>
<td>[0, 0.1]</td>
<td>[0, 1]</td>
<td>[0, 30]</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>*</td>
<td></td>
<td>[0, 0.1]</td>
<td>[0, 1]</td>
<td>[0, 3]</td>
</tr>
<tr>
<td>9</td>
<td>*</td>
<td></td>
<td></td>
<td>[0, 0.1]</td>
<td>[0, 0.1]</td>
<td>[0, 3]</td>
</tr>
<tr>
<td>10</td>
<td>*</td>
<td></td>
<td></td>
<td>[0, 0.05]</td>
<td>[0, 0.1]</td>
<td>[0, 3]</td>
</tr>
<tr>
<td>11</td>
<td>*</td>
<td></td>
<td></td>
<td>[0, 0.01]</td>
<td>[0, 0.1]</td>
<td>[0, 3]</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>*</td>
<td></td>
<td>[0, 0.01]</td>
<td>[0, 0.1]</td>
<td>[0, 0.3]</td>
</tr>
</tbody>
</table>

Figure 2.6: An example for illustrating cyclic phenomena in narrowing.
Chapter 3

The Language

This chapter describes the proposed programming language and its imperative and non-deterministic constructs. The nucleus of the constraint solver is described in detail in Sections 3.4 and 3.11. Several examples written in the proposed language will be presented in Chapter 4; they demonstrate the language’s expressiveness in solving deterministic and non-deterministic programs.

Basically, the statements in the source program are translated into a series of the equality, inequality, and disequality constraints. As mentioned previously, a solution is found if the lower and upper bounds of all the output variables in the program are considered equal within the required precision. If any of the narrowing operation fails, then there is no solution within the current execution path of the program. In the case of non-deterministic programs, alternate execution paths are traversed in case of failure, or if all the solutions are desired. At the end of each narrowing cycle, the intervals of the output variables may not be close enough to provide an adequate solution. In that case, the domain of the intervals is split into two and each alternative is explored separately or
jointly depending on sequential or parallel implementations. Several heuristic strategies
for interval domain splitting are presented in Section 3.9.

3.1 Grammar

A subset of the grammar describing the constraint language is shown in Figure 3.1. This
subset is used to explain the various algorithms presented in the following sections. The
BNF specification for the entire language is presented in the Appendix A.1.

3.2 Data Structures for Description of Algorithms

For illustration purposes, programs in the language are represented by lists of statements.
In the description of algorithms in the following sections, we will consider three types of
statements from the language:

- \textit{CHOICE} statement, with access to its \textit{head} and \textit{tail} components,

- \textit{SPLIT} statement, with access to its \textit{first} and \textit{second} splitting components,

- \textit{CONSTRAINT}, of the form \( x \ op \ y = z \), or \( op \ x = y \), or \( x \ op y \).

The parameters of the \textit{solve} interpreter (described in Section 3.4) which traverses the
source tree and solves the constraints are:

\( p \) : The input \textit{program}, comprising the sequence of statements, with access to the \textit{head}
and \textit{tail} of the of the sequence.

\( cl \) : The accumulated \textit{constraint-list}, as the \textit{solve} process traverses through the \textit{program},
with access to the \textit{head} and \textit{tail} of the constraint list.

42
program → statement_list

statement_list → statement ; statement_list | e

statement → CHOICE <two or more statements>
   | SPLIT variable ( c_exp , c_exp ) ( c_exp , c_exp )
   | constraint
   | { statement_list }

constraint → c_exp comp_op c_exp
   | INT variable_list
   | BOOL variable_list
   | BIND variable ( constant , constant )

c_exp → c_exp op_2 c_exp
   | op_1 c_exp
   | variable
   | constant
   | ( c_exp )

comp_op → = | <> | ≤ | ≥

op_2 → PLUS | MINUS | OR | TIMES | DIV | AND | POWER

op_1 → NOT | UMINUS | SQRT | INTEGER | SIN | COS | ...

c_exp is a constraint arithmetic expression. L and U are the lower and upper bounds of a variable.

Figure 3.1: A subset grammar for the language
BIND y (1, 3);

CHOICE {
    {x = 1};
    {x = 2};
    {x = 3}
};

SPLIT y (l(y), l(y)) (l(y) + 1, u(y));

x & y;

write x, y

Figure 3.2: An example

*iv*: The interval-vector of the variables appearing in the constraint_list.

In the following figures, lists are depicted as trees. The list structure for the program in Figure 3.2 is shown in Figure 3.3.

### 3.3 Nucleus of the Constraint Solver

The flowchart describing the constraint solver is shown in Figure 3.4. The flowchart is useful in developing the Markovian analysis for analyzing the time to solve the constraints. The major steps of the interval based constraint solver are as shown below:

1. Consider the list of intervals for the variables appearing in the input constraints.

2. Apply the narrowing operation for all the constraints and recompute lower and upper bounds.
3. Test for convergence, i.e., the list of intervals stabilize.

   If no convergence is attained, repeat Steps 2 and 3.

4. Check if results are satisfactory, i.e., they yield the desired results.

   If not, split and resume at Step 2.

5. Output results.

### 3.4 The Procedure Solve

The sequential procedure `solve` which traverses the syntax-tree representing a source program and interprets the constraints is shown in Figure 3.5. The main call is `solve(p, NIL, iv)`, where `p` points to the root of the syntax-tree, and `iv` is the interval-vector with all the variables initialized to $(-\infty, \infty)$. The second parameter `cl` (initialized to `NIL`) contains the list of accumulated constraints.

The procedure `narrow` is described in detail in Section 3.4.1. After all the constraints are processed, the resulting interval vector might not be a solution. An interval vector is a solution if the lower and upper bounds are equal or within a desired precision for all the intervals in the interval vector. It is worthwhile recalling that in
Figure 3.4: Nucleus of the constraint solver
CLP(Intervals), a final resulting interval vector has the following interpretation: “If there exist one or more solutions, these lie within the bounds of the resulting interval vector.” The properties of the narrowing function ensure that no solutions are lost in the narrowing phase. The procedure `enumerate` shown in Figure 3.6 is used to actually determine if these solutions exist. The function `strategy` returns which variable to be split. The variable is chosen based upon the domain splitting strategies as explained in Section 3.9. The procedure `splitintervals` shown in Figure 3.7 splits the given interval vector into two interval vectors. If the split variable is constrained to be boolean, the interval for that variable is split into [0, 0] and [1, 1]. If the split variable is constrained as integer, the interval \([l_{var}, u_{var}]\) is split into \([l_{var}, mid]\) and \([mid + 1, u_{var}]\), where \(mid\) is the integer mid-point of \(l_{var}\) and \(u_{var}\). Otherwise, the interval is split into \([l_{var}, mid]\) and \([mid, u_{var}]\).

3.4.1 The procedure Narrow

Each constraint in the constraint-list has a unique index (e.g., the constraint’s position in the list). In what follows the notion of a constraint and its index are used interchangeably. The following data structures are useful in implementing the narrow operation efficiently:

**dependency-list**: For each variable occurring in the constraint-list, the dependency-list is a linked list of the constraints in which the variable appears.

**constraint-queue**: The constraints yet to be narrowed are processed using the FIFO queue.

**is-queued**: A Boolean vector to determine if a constraint is already in the queue.

The queue operations are:
\[\text{Sequential procedure } \text{solve}\]

\begin{verbatim}
procedure solve(p, cl, iv)

/* p — the source program */
/* cl — the constraint list */
/* iv — the interval vector */

begin
    if p \neq \text{NIL} then
        if narrow(cons(head(p), cl), iv, iv') then
            solve(tail(p), cons(head(p), cl), iv')
        end
    else begin
        if enumeration_desired then enumerate(cl, iv)
        else print(iv)
    end
end solve
\end{verbatim}

Figure 3.5: The procedure solve.
procedure enumerate(cl, iv)

/* cl — the constraint list */

/* iv — the interval vector */

begin

   var ← strategy(iv) % determine the variable to be split%

   split_intervals(var, iv, iv₁, iv₂)

   if narrow(cl, iv₁, iv₁') then

      begin

         if solution(iv₁') then print(iv₁') else enumerate(cl, iv₁')

      end

   if narrow(cl, iv₂, iv₂') then

      begin

         if solution(iv₂') then print(iv₂') else enumerate(cl, iv₂')

      end

end enumerate

Figure 3.6: The procedure enumerate.
procedure split_intervals(var, iv, iv₁, iv₂)

/* var — the variable to be split */
/* iv — the input interval vector */
/* iv₁, iv₂ — the two output interval vectors */

begin

iv₁ ← iv; iv₂ ← iv

if bool_var(var) then /* variable constrained as boolean */

begin

iv₁^ar = (0,0); iv₂^ar = (1,1)

end

else if integer_var(var) then begin /* variable constrained as integer */

mid = integer_mid_point(iv₁^ar₁, iv₂^ar₁)

iv₁^ar₁ = mid; iv₂^ar₁ = mid + 1

end

else begin /* variable not constrained, split at the mid-point */

iv₁^ar₁ = iv₂^ar₁ = mid_point(iv₁^ar₁, iv₂^ar₁)

end

end split_intervals

Figure 3.7: The procedure split_intervals.
\textit{init\_queue(\textit{queue})}: Initialize the empty queue.

\textit{put\_queue(\textit{queue, c})}: Add the constraint \textit{c} to the tail of the queue.

\textit{get\_queue(\textit{queue, op(x,y,z)})}: The constraint \textit{op(x,y,z)} is deleted from the front of the queue. The constraint index \textit{c} is returned.

When a new constraint is encountered while traversing the syntax tree of the source program, the constraint is initially placed in the constraint-queue. The narrow procedure (Figure 3.8) obtains a constraint from the queue, performs the appropriate narrow operation, and checks if the interval bounds for any of the variables appearing in the constraint have changed. In such a case, the constraints in which the variables appear are added to the queue, if they are not already in the queue. The auxiliary procedure \textit{add\_constraints} (Figure 3.9) is used to add the constraints to the queue. The interval vector is said to converge when the constraint-queue becomes empty.

A revised interval narrowing algorithm is proposed in (LhGoRu 94) which speeds up the convergence process. The algorithm presented in that paper identifies cycles in which a lot of narrowing operations are carried out unnecessarily.
procedure narrow(cl, iv, iv')
/* cl — the input constraint list */
/* iv — the input interval vector */
/* iv' — the (output) narrowed interval vector */
begin
  init_queue(constraint-queue); is-queued ← false
  put_queue(constraint-queue, head(cl)); is-queued_{head(cl)} ← true
  iv' ← iv
  while not empty_queue(constraint-queue) do
    c ← get_queue(constraint-queue, op(x,y,z)); is-queued_{c} ← false
    if narrow_op((x,y,z), iv', (x',y',z')) then
      if change(iv_x,x') then
        iv_x ← x'; add_constraints(x, constraint-queue, is-queued) end if
      if change(iv_y,y') then
        iv_y ← y'; add_constraints(y, constraint-queue, is-queued) end if
      if change(iv_z,z') then
        iv_z ← z'; add_constraints(z, constraint-queue, is-queued) end if
      else return false
    end if
  end while
  return true
end narrow

Figure 3.8: The procedure narrow.
procedure add_constraints(var, constraint-queue, is-queued)

/* add the constraints in which var appears to the constraint-queue */
/* is-queued is true if the constraint c is already present in the constraint-queue */

begin

constraint-list-ptr ← dependency-list_{var}

while constraint-list-ptr ≠ Nil do

c ← head(constraint-list-ptr); constraint-list-ptr ← tail(constraint-list-ptr);

if not is-queued then

put_queue(constraint-queue, c); is-queued ← true end if

end while

end add_constraints

Figure 3.9: The procedure add_constraints.
3.5 Passive Constraints

In the narrowing phase, while processing the constraints, if the interval of any variable changes, all the constraints in which the variable appears are added to the constraint queue. In particular cases, some constraints need not be added. We term such constraints as passive. Here are some cases where the constraints are passive:

\[ x \leq y, \quad \text{if } U(x) \leq L(y). \]

\[ x \neq y, \quad \text{if } (L(x), U(x)) \cap (L(y), U(y)) = \perp. \]

\[ x = y, \quad \text{if } L(x) = U(x) \text{ or } L(y) = U(y). \]

3.6 Data Types

The following basic data types are used in the language:

- **real**, to store floating point numbers in double precision,
- **integer**, to store an integral number, and
- **interval**, specified by two floating point numbers, a lower and upper bound.

Vectors can also be defined in the language as follows:

\[ \text{array } [DIM_1, DIM_2, \ldots, DIM_n] \text{ of basic data type}, \text{where the } i^{th} \text{ dimension is indexed 1 to } DIM_i. \text{ A detailed description of the variables and their declarations is provided in Section 3.6.1. An example illustrating arrays is presented in Section 3.6.2.} \]

3.6.1 Variable Types

Variable declarations in the language can be classified into two categories:
1. **logic variables**—these resemble the Prolog logic variables. These variables cannot be reinstantiated to a different value once they are given a value. However, since we are dealing with interval logic variables, the interval of the variable itself changes (narrows). As long as the interval does not represent a value, any assignment of a value (or interval) within that interval succeeds. The declaration of the logic variables is as follows:

```plaintext
var X, Y : interval;
```

2. **ordinary (real and integer) variables**—these inherit the properties of the imperative language variables. The use of these variables must be preceded by their definition. Reassignment holds without restrictions. These variables cannot be intervals and have to defined using fixed values. If these variables appear as the *l-value* of an assignment, the *r-value* has to be a fixed value. Another characteristic of these variables is that, on backtracking, they retain the current value. The declaration of these type of variables is as shown below:

```plaintext
var Z, W : real;
var I, J : integer;
```

The example in Figure 3.10 illustrates the difference between logic and ordinary variables. The declarations of these variables are exemplified by the keywords *interval* and *integer* respectively. The output of the program is shown in Figure 3.11.
main()
{
    var X, Y : interval;
    var J : integer;
    {
        choice {
            { X = 1 };
            { X = 2 }
        };
        J = 2;
        write X, J;
        choice {
            { Y = 3 };
            { Y = 4 }
        };
        J = 3;  /* J remains 3 even on backtracking */
        write Y, J;
        fail
    }
}

Figure 3.10: Example illustrating the types in the language.

X = 1; J = 2; Y = 3; J = 3;
    Y = 4; J = 3;
X = 2; J = 3; Y = 3; J = 3;
    Y = 4; J = 3

Figure 3.11: Output of the program shown in Figure 3.10

56
main()
{
    var X : interval;
    var N : integer;
    var A: array[10] of interval;
    {
        N = 10;
        member(X,1,A);
        write X;
        fail
    }
}

proc member(X: interval; L: integer; S: array)
{
    {
        choice {
            { L <= N; X = S[L] };
            { L <= N; member(X,L+1,S) }
        }
    }
}

Figure 3.12: The member function.

3.6.2 List Processing with Arrays

In the present implementation, arrays of fixed sizes can only be declared in the main procedure (i.e., arrays cannot be declared as local variables within procedures). An array name can be passed as an argument to a procedure call. This restricts the ability to use arrays in applications which rely heavily on list structures. However, a partial support for procedures involving lists can be accomplished. One such is the member procedure shown below:
3.7 Imperative Statements

The imperative statements used in the language are the if-then, if-then-else, for, while-do, and do-while statements. The code in Figure 3.13 shows the use of each of these statements to compute the sum of first $n$ numbers.

```haskell
var i, sum, n: integer;

/* using the for statement */
sum = 0;
for (i = 1; i <= n; i = i+1)
    sum = sum + i

/* using the while-do statement */
sum = 0; i = 1;
while (i <= n) do
    { sum = sum + i; i = i + 1}
```

Figure 3.13: Some imperative statements

3.8 Procedures

A procedure declaration in the language has the following syntax:

```haskell
proc procedure_name (formal_parameter_list)
{
    local_declarations
    {
        statements
    }
}
```
The entry point of the program is the procedure \texttt{main}. A procedure call is a statement of the form: \texttt{procedure\_name(actual\_parameters)}). The statements within a procedure can make procedure calls to the same procedure or any other procedure, thus supporting recursion.

### 3.9 Domain Splitting and Tree Traversal Strategies

The narrowing algorithm produces as output an interval vector which might not be the desired solution (i.e., the lower and upper bounds are not close enough to the desired precision). In order to obtain the actual solutions, the intervals of the source variables are split (CLEA 87) and passed on to the narrowing algorithm.

**Example:** Consider the set of constraints \{X \neq Y, X \neq Z, Y \neq Z\}. Let the initial interval vector for the variables \(X, Y, Z\) be \([1, 4], [1, 4], [1, 4]\). Narrowing the constraint set does not change the interval vector (See Figure 3.14).

<table>
<thead>
<tr>
<th>(x \neq y)</th>
<th>(x \neq z)</th>
<th>(y \neq z)</th>
<th>(x \in)</th>
<th>(y \in)</th>
<th>(z \in)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*)</td>
<td>([1, 4])</td>
<td>([1, 4])</td>
<td>([1, 4])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(*)</td>
<td>([1, 4])</td>
<td>([1, 4])</td>
<td>([1, 4])</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(*)</td>
<td>([1, 4])</td>
<td>([1, 4])</td>
<td>([1, 4])</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.14: An example for illustrating splitting strategies.

The language provides the following traversal strategies for exploring all solutions in the search tree:

- **depth-first traversal**, the first source variable with different lower and upper
bounds is split. This corresponds to a depth-first traversal of the solution space.

The narrowing steps leading to the first solution for the example in Figure 3.14 are shown in Figure 3.15.

When a variable is split, there are two sets of interval vectors. The narrowing continues with the first set. The second set is pushed onto a stack. When narrowing fails, or if more solutions are desired, an interval vector is popped from the stack and narrowing continues.
<table>
<thead>
<tr>
<th>Split</th>
<th>(x \neq y)</th>
<th>(x \neq z)</th>
<th>(y \neq z)</th>
<th>(x \in)</th>
<th>(y \in)</th>
<th>(z \in)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>([1, 2])</td>
<td>([1, 4])</td>
<td>([1, 4])</td>
<td>([1, 2])</td>
<td>([1, 4])</td>
<td>([1, 4])</td>
</tr>
<tr>
<td>(*)</td>
<td>([1, 2])</td>
<td>([1, 4])</td>
<td>([1, 4])</td>
<td>([1, 2])</td>
<td>([1, 4])</td>
<td>([1, 4])</td>
</tr>
<tr>
<td>(y)</td>
<td>([1, 1])</td>
<td>([1, 4])</td>
<td>([1, 4])</td>
<td>([1, 1])</td>
<td>([2, 4])</td>
<td>([1, 4])</td>
</tr>
<tr>
<td>(*)</td>
<td>([1, 1])</td>
<td>([2, 4])</td>
<td>([2, 4])</td>
<td>([1, 1])</td>
<td>([2, 4])</td>
<td>([2, 4])</td>
</tr>
<tr>
<td>(*)</td>
<td>([1, 1])</td>
<td>([2, 4])</td>
<td>([2, 4])</td>
<td>([1, 1])</td>
<td>([2, 4])</td>
<td>([2, 4])</td>
</tr>
<tr>
<td>(*)</td>
<td>([1, 1])</td>
<td>([2, 3])</td>
<td>([2, 4])</td>
<td>([1, 1])</td>
<td>([2, 3])</td>
<td>([2, 4])</td>
</tr>
<tr>
<td>(*)</td>
<td>([1, 1])</td>
<td>([2, 2])</td>
<td>([2, 4])</td>
<td>([1, 1])</td>
<td>([2, 2])</td>
<td>([2, 4])</td>
</tr>
<tr>
<td>(*)</td>
<td>([1, 1])</td>
<td>([2, 2])</td>
<td>([2, 4])</td>
<td>([1, 1])</td>
<td>([2, 2])</td>
<td>([3, 4])</td>
</tr>
<tr>
<td>(*)</td>
<td>([1, 1])</td>
<td>([2, 2])</td>
<td>([3, 3])</td>
<td>([1, 1])</td>
<td>([2, 2])</td>
<td>([3, 3])</td>
</tr>
</tbody>
</table>

Figure 3.15: Depth-first traversal strategy.
- **round-robin traversal**, where the variable to be split is chosen in a cycle. Assuming the source variables to be split are ordered $1 \ldots n$, the $i_{th}$ variable is chosen in the $i_{th}$ step. This corresponds to a breadth-first traversal of the solution space. The narrowing steps leading to the first solution for the example in Figure 3.14 are shown in Figure 3.16.

- **first-fail traversal**, the variable with the smallest domain, i.e., the most constrained variable, is the one chosen for splitting. In the above example, first-fail traversal follows the same pattern as that for depth-first strategy.

- **fair traversal**, assuming the source variables to be split are ordered $1 \ldots n$, in the $k_{th}$ step, the $i_{th}$ variable, $1 \leq i \leq n$, is split $j$ times such that $i + j = k$. This corresponds to a snake pattern as shown in Figure 3.17 (HoUl 69). Let the pair $(i, j)$ denote the fact that the $i_{th}$ variable is split $j$ times. The first few pairs are $(1, 1), (1, 2), (2, 1), (1, 3), (2, 2), (3, 1)$, etc. Given any pair $(i, j)$, the pair will eventually appear in the list. In the above example, fair traversal follows the same pattern as that for depth-first strategy.

The following directives can be specified in the language to enforce a particular traversal strategy: **DFMODE**, **RRMODE**, **FFMODE**, and **FAIRMODE**. If no splitting strategy is specified, depth-first traversal is used.

When a variable is selected for splitting, the domain of the variable can be subdivided in several ways. Two special constructs, **FRSPLIT** and **MIDSPLIT**, allow the domain to be split into the first and rest components in the former case, and into two equal ranges in the latter case.
<table>
<thead>
<tr>
<th>Split</th>
<th>$x \neq y$</th>
<th>$x \neq z$</th>
<th>$y \neq z$</th>
<th>$x \in$</th>
<th>$y \in$</th>
<th>$z \in$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td></td>
<td></td>
<td></td>
<td>[1, 2]</td>
<td>[1, 4]</td>
<td>[1, 4]</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>[1, 2]</td>
<td>[1, 4]</td>
<td>[1, 4]</td>
</tr>
<tr>
<td>$y$</td>
<td></td>
<td></td>
<td></td>
<td>[1, 2]</td>
<td>[1, 2]</td>
<td>[1, 4]</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>[1, 2]</td>
<td>[1, 2]</td>
<td>[1, 4]</td>
</tr>
<tr>
<td>$z$</td>
<td></td>
<td></td>
<td></td>
<td>[1, 2]</td>
<td>[1, 2]</td>
<td>[1, 2]</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>[1, 2]</td>
<td>[1, 2]</td>
<td>[1, 2]</td>
</tr>
<tr>
<td>$x$</td>
<td></td>
<td></td>
<td></td>
<td>[1, 1]</td>
<td>[1, 2]</td>
<td>[1, 2]</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td>[1, 1]</td>
<td>[2, 2]</td>
<td>[1, 2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td>[1, 1]</td>
<td>[2, 2]</td>
<td>[2, 2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td>[1, 1]</td>
<td>fail</td>
<td>fail</td>
</tr>
<tr>
<td>Pop ($x$)</td>
<td></td>
<td></td>
<td></td>
<td>[2, 2]</td>
<td>[1, 2]</td>
<td>[1, 2]</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td>[2, 2]</td>
<td>[1, 1]</td>
<td>[1, 2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td>[2, 2]</td>
<td>[1, 1]</td>
<td>[1, 1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td>[2, 2]</td>
<td>fail</td>
<td>fail</td>
</tr>
<tr>
<td>Pop ($z$)</td>
<td></td>
<td></td>
<td></td>
<td>[1, 2]</td>
<td>[1, 2]</td>
<td>[3, 4]</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>[1, 2]</td>
<td>[1, 2]</td>
<td>[3, 4]</td>
</tr>
<tr>
<td>$x$</td>
<td></td>
<td></td>
<td></td>
<td>[1, 1]</td>
<td>[1, 2]</td>
<td>[3, 4]</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td>[1, 1]</td>
<td>[2, 2]</td>
<td>[3, 4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td>[1, 1]</td>
<td>[2, 2]</td>
<td>[3, 4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>[1, 1]</td>
<td>[2, 2]</td>
<td>[3, 4]</td>
</tr>
<tr>
<td>$z$</td>
<td></td>
<td></td>
<td></td>
<td>[1, 1]</td>
<td>[2, 2]</td>
<td>[3, 3]</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>[1, 1]</td>
<td>[2, 2]</td>
<td>[3, 3]</td>
</tr>
</tbody>
</table>

Figure 3.16: Round-robin traversal strategy.
3.10 Non-deterministic Constructs

The basic non-deterministic primitives supported in the proposed language are the \texttt{CHOICE}, the \texttt{SPLIT}, and the \texttt{FAIL} constructs. The \texttt{EITHER ... OR, SOME}, and \texttt{REDO} constructs are variations of the \texttt{CHOICE} statement (ApSc 97).

3.10.1 \textit{Choice}

The syntax of the \texttt{CHOICE} statement is as shown below:

\begin{verbatim}
CHOICE \textit{S}_1; \textit{S}_2; \ldots; \textit{S}_n
\end{verbatim}

where each \textit{S}_i is a statement. Operationally, the statement \textit{S}_1 is first explored and the rest of the alternatives are pushed onto the choice stack. If an explicit \textit{fail} statement is encountered, or if the current set of constraints is infeasible, backtracking takes place. The state corresponding to the previous choice point is restored. Execution resumes with the next alternate choice from the top of the choice stack.
3.10.2 Split

The **SPLIT** statement has the following syntax:

```
SPLIT var (exp1, exp1_) (exp2, exp2_)
```

The statement specifies how the interval for the variable `var` has to be split. The interval vector is split into two components. One component has the interval `(exp1, exp1_)` for the variable `var`, while the other component has the interval `(exp2, exp2_)` for the variable `var`. One of the components is first explored. The other is pushed onto the choice stack.

The **Split** statement should not be confused with the domain splitting strategies described in Section 3.9. The strategies described in that section control the order in which the variables are split. The **Split** statement specifies how a variable is split.

3.10.3 Fail

Explicit backtracking is triggered by the **FAIL** statement. If there is an alternate choice, execution resumes with that choice. On the other hand, if no alternatives exist, the program ends in a failure. The syntax of the **FAIL** statement is:

```
FAIL
```

To prune the set of choices, the **CUT** statement could be used. This removes the sibling choices. However, the **CUT** and the **FAIL** statements should not be used together to achieve the *cut-fail* mechanism of Prolog. This is because the **FAIL** statement in the language is not local to the **CHOICE** statement and pops the top choice of the choice stack.
3.10.4 Either ... Or

The `either ... or` statement is a restricted form of the `choice` statement; it has only two alternatives. The syntax of the statement is as shown below:

```
     either S1 or S2
```

where $S_1$ and $S_2$ are any statements in the language. The `either ... or` statement is equivalent to:

```
     choice $S_1$; $S_2$
```

3.10.5 Some

The `some` statement is an iterative form of the `choice` statement. The number of alternatives in the body of the `some` statement is evaluated dynamically. The `some` statement has the following syntax:

```
     some(v = expr1; conditional_expr; v = expr2) S
```

$v$ is the variable (of type `integer`) whose value controls the non-deterministic iterations of the statement $S$; $expr_1$ is the initial value for the variable $v$; $conditional_expr$ is a conditional expression involving the variable $v$, if the conditional expression evaluates to true, a choice point is pushed onto the choice stack and the statement $S$ is executed; $expr_2$ affects the value of the variable $v$ through each non-deterministic iteration of the statement $S$.

Example: The statement

```
     some{i = 1; i <= 5; i = i + 1} {x = i}
```

is equivalent to

```
     choice { {x = 1}; {x = 2}; {x = 3}; {x = 4}; {x = 5} }
```
var i, counter: integer;
var x: interval;
{
    counter = 0;
    REDO {
        SOME(i=1; i <= 5; i = i + 1)
        {
            x = i;
        }
    }
    AFTER
    {
        counter = counter + 1;
    }
    write counter
}

Figure 3.18: The Redo example

3.10.6 Redo

The statement REDO $S_1$ AFTER $S_2$ specifies that the statements $S_1$ and $S_2$ are first executed in sequence; if there is a choice point left within $S_1$, control returns to that choice point and $S_2$ is once again executed. The process continues until there are no more choice points within $S_1$. The computation then succeeds and proceeds with the statement after the redo statement. For example, the program in Figure 3.18 succeeds and outputs the value 5.

The non-deterministic constructs explained in this section are variants of the Choice, Split, and Fail statements. Therefore, only these forms of the non-deterministic constructs will be used in algorithms described in subsequent sections.
3.11 Modified Procedure Solve

The sequential procedure solve which traverses the syntax-tree representing a source program and interprets the constraints (including the choice and the split statements) is shown in Figure 3.19. The main call is solve(p, NIL, iv), where p points to the root of the syntax-tree, and iv is the interval-vector with all the variables initialized to (-\infty, \infty). The second parameter cl (initialized to NIL) contains the list of accumulated constraints.

When a choice is encountered while executing a sequence of statements, the program continues execution with the first choice and pushes the rest of the choices onto a stack (the procedure push in Figure 3.19). When a failure occurs, the alternate statement list is popped from the stack and the program continues execution (the procedure pop in Figure 3.19).
procedure solve(p, cl, iv) /* p: program, cl: constraint list, iv: interval vector */
begin
  if p ≠ NIL then
    switch head(p)
    case CHOICE:
      push(cons(tail(head(p)), tail(p)), cl, iv);
      solve(cons(head(head(p)), tail(p)), cl, iv)
    case SPLIT:
      split_intervals(head(p), iv, iv1, iv2);
      push(tail(p), cons(head(p), cl), iv2);
      solve(tail(p), cons(head(p), cl), iv1)
    case FAIL:
      if pop(p, cl, iv) then solve(p, cl, iv)
    otherwise: /* a constraint */
      if narrow(cons(head(p), cl), iv, iv') then
        solve(tail(p), cons(head(p), cl), iv')
      else if pop(p, cl, iv) then solve(p, cl, iv)
    end switch
  else begin
    if enumeration_desired then enumerate(cl, iv) else print(iv)
    if pop(p, cl, iv) then solve(p, cl, iv)
  end
end solve

Figure 3.19: The Revised procedure solve
Chapter 4

Examples

In the following sections we present some typical examples of the usage of interval variables, constraints, and non-determinism. They demonstrate that our approach can achieve a versatility comparable to that of languages like Prolog III and CLP(ℜ), but with a considerably reduced overhead, and in many cases, with less programming effort. The deterministic programs will be presented in section 4.1, where as the non-deterministic ones appear in section 4.2. Some of the programs are juxtaposed with their corresponding triplet representation. The variety of examples presented in this chapter demonstrate the versatility of intervals. They are taken from areas as varied as engineering, numerical analysis, number theory, and roots of constraints in problems from solving polynomial equations.

Section 4.3 shows with an example how Hidden Markov Models can be modeled using interval constraints. Two computationally difficult problems are examined in Section 4.4. Section 4.4.3 narrates how constraint satisfaction problems can be transformed into constraints in the proposed language. An example of partially evaluating a program
main()
{ var N: integer; var x: interval;
  var A: array [3] of interval;
  N = 3;
  installment(3,1000);
  write x
}
}

proc installment(i: integer; c: interval)
{ var c1: interval;
  { if i == 0 then c = 0
   else
   { c1 + A[N - i + 1] = 1.1 * c;
     installment(i - 1, c1)
   }
  }
}

(a)eña

(b)

Figure 4.1: (a) Installment program. (b) Generated constraints.

in the language is presented in Section 4.5.2.

4.1 Deterministic Programs

In this section, we show several examples using deterministic constructs of the language.

The constraints, after the narrowing process, may not yield a solution. In such cases, a
depth-first enumeration strategy is used in obtaining a solution, if one exists.
4.1.1 Installment Problem

The following example from (COLM 90) is frequently used to illustrate the capabilities of Prolog III. The problem is to determine the relationship between logic variables representing a given amount of money $C$, an interest rate $R$, and a list of $n$ monthly installments $X_1, X_2, \ldots, X_n$. The list is implemented as an array whose contents are the logic variables $X_i$'s. The problem can be stated by specifying $n$ constraints:

$$C_{i+1} + X_i = (1 + R) \times C_i,$$

with $i = 1, \ldots, n$, $C_1 = C$, and $C_{n+1} = 0$.

As stated above, the problem is non-linear due to the product $R \times C_i$. However, when $R$ is a known constant, one can determine the $X_i$'s given the value of $C$, or the resulting $C$ given the values of the $X_i$'s. If $R$ is an interval, the non-linear case can be handled using enumeration.

The procedure `installment` in Figure 4.1 has two parameters: the argument $i$ denoting the remaining $i$ installments to repay an amount $c$. The main program calls `installment` to determine the value $x$ for the case $c = 1,000$ such that the monthly installments are $x$, $2 \times x$, and $3 \times x$ considering an interest rate 0.1. An edited form of the generated constraints is presented in infix notation in Figure 4.1.

4.1.2 Circuit Analysis-1

Constraints have often been used in the design of electric circuits (JMSY 92). The circuit in Figure 4.2 can be analyzed by solving constraints which specify the application of Ohm's law $V = I \times R$, where $V$ stands for the voltage, $I$ for the current, and $R$ for the resistance. The circuit being considered has two parallel branches which are connected in series.
The procedure `resistor` embodies Ohm's law. In a parallel circuit, Kirchhoff's law is applicable (the sum of the currents into a node equals the sum of the outgoing currents). In a serial circuit, Kirchhoff's voltage law is applicable (the sum of the voltages along a closed circuit is zero). The `main` program corresponds to determining the voltage \(v\) when the current \(i\) has a value 10 amps and the resistors having the values 10, 20, 30 and 40 ohms. (Figure 4.3)

### 4.1.3 Circuit Analysis-2

The example in this section shows the familiar full-adder circuit (Figure 4.4). The gates in the circuit are labeled 1 \ldots 5. Each gate \(i\) is associated with a Boolean logic variable \(g_i\). \(g_i\) is true if the corresponding gate is faulty. The constraint that at most \(n\) gates are faulty is represented as:

\[
g_1 + g_2 + g_3 + g_4 + g_5 \leq n.
\]
main()
{ var v: interval;
    { par_series(v,10,10,20,30,40);
        write V
    }
}
proc par_series(v,i,r1,r2,r3,r4: interval)
{ var v1,v2: interval;
    {v1+v2 = v;
        par_circuit(v1,i,r1,r2); par_circuit(v2,i,r3,r4)
    }
}
proc par_circuit(v,i,r1,r2: interval)
{ var i1,i2: interval;
    { i1+i2 = i;
        resistor(v,i1,r1); resistor(v,i2,r2)
    }
}
proc resistor(v,i,r: interval)
{ v = i * r
}
Output:

v = 238.09

Figure 4.3: Circuit program and the result of its evaluation.
The truth table of the *full-adder* circuit can be verified by specifying that none of the gates are faulty, i.e., $n = 0$. The program is shown in Figure 4.5.
main()
{
  var x,y,z,s,c: interval;
  var g1, g2, g3, g4,g5: interval;
  {
    bool x, y,z,s,c;
    bool g1, g2, g3, g4,g5;

    g1 + g2 + g3 + g4 + g5 <= 0;
    adder(x,y,z,s,c,g1,g2,g3,g4,g5);
    write x, y, z, s, c, g1, g2, g3, g4, g5
  }
}

proc adder(x,y,z,s,c, g1, g2, g3, g4, g5: interval)
{
  var u1, u2, u3: interval;
  {
    bool u1, u2, u3;

    (*g4) <-> (u3 <-> "(x <-> z)") = 1;
    (¬g5) <-> (s <-> "(y <-> u3)") = 1;
    (¬g1) <-> (u1 <-> (x & z)) = 1;
    (*g2) <-> (u2 <-> (y & u3)) = 1;
    (*g3) <-> (c <-> (u1 # u2)) = 1
  }
}

Figure 4.5: Program Verifying the Full-adder Circuit.
4.1.4 Polynomial Interpolation

The following example won a prize in a competition among Prolog III students at the University of Marseilles, France. Let us assume we have a sequence of \( n \) points specified by their coordinates \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\). Consider the polynomial:

\[
P(x) = a_0 + a_1 * x + \ldots + a_{n-2} * x^{n-2} + a_{n-1} * x^{n-1}
\]

\[
= a_0 + (a_1 + (\ldots (a_{n-2} + a_{n-1} * x) * x) \ldots) * x
\]

We wish to determine the values of the coefficients \( a_i \) such that:

\[
P(x_1) = y_1, \ P(x_2) = y_2, \ldots, \ P(x_n) = y_n.
\]

Since the values of the pairs \((x_i, y_i)\) are known, the program in Figure 4.6 can generate \( n \) linear simultaneous equations corresponding to \( y_i = P(x_i) \). Their solution yields the values for the \( a_i \)'s. The procedure \texttt{lagrange} invokes the procedure \texttt{horner} \( n \) times to execute \( y_i = P(x_i) \), for \( i = 1, \ldots, n \). The computation of \( P(x_i) \) is done using Horner's scheme and therefore generates the simultaneous equations having the \( a_i \)'s as unknowns. The output \( a_i \)'s for the particular example yield the polynomial \( 1 + 2x + x^2 \).
main()
{ var N: integer;
    N = 3;
    lagrange(3);
}
}
proc lagrange(n: integer)
{
    if n > 0 then
    { horner(N, X[N - n + 1], Y[N - n + 1]);
        lagrange(n - 1)
    }
}
proc horner(n: integer; x,y: interval)
{ var s: interval;
    if n == 0 then y = 0
    else
    { y = A[N - n + 1] + x * s;
        horner(n - 1, x, s)
    }
}
}

Figure 4.6: Polynomial Interpolation using Horner’s method.
4.1.5 Diophantine Equations and a Problem Suggested by Fermat

main()
{
    var x, y, w, z: interval;
{
        int x, y, w, z;

        BIND x (1,100);
        BIND y (1,100);
        BIND z (1,100);
        BIND w (1,100);

        x^2 + y^2 + w^2 = z^2;
        x <= y;
        y <= w;
        write x, y, w, z
    }
}

Results:

Figure 4.7: Diophantine Equations

The program is shown in Figure 4.7 finds positive integer values between, say, 1 and 100, for the variables x, y, w, z satisfying the equation

\[ x^2 + y^2 + w^2 = z^2. \]
A Problem Suggested by Fermat

Consider the sequence of consecutive integers from 1 to $\infty$. The problem is to find three consecutive integers such that the first number is a square number and the third number is a cubic number (SING 98).

Example: The sequence 25, 26, 27 is the only sequence satisfying the constraints. The left number 25 is a square of 5 and the right number 27 is a cube of 3.

The above conditions can be expressed formally as follows: Consider the sequence $x-1, x, x+1$, such that $y^2 = x - 1$ and $z^3 = x + 1$. Eliminating the variable $x$, the constraint becomes $y^2 + 2 = z^3$, $y, z \in [1, \infty]$. The values of the intervals for the variables $y$ and $z$ during the narrow and split operations are as follows:

- **Narrow:** $y \in [5, \infty], z \in [3, \infty]
- **Split y:** $y \in [5, 5], z \in [3, \infty]
- **Narrow:** $y \in [5, 5], z \in [3, 3]$

Solution

The details of the above operations are shown in Figure 4.8. As seen above, the program can determine the existing solution. However, the task of proving that this is the only solution is a considerably harder problem that cannot be determined using intervals, since at the limit, i.e., when the upper bounds is $\infty$, the equation is satisfied. Therefore, one can extend the upper bound to the largest existing floating point number in a computer and prove that there are no solutions in that range.
<table>
<thead>
<tr>
<th>$y^2 = t_1$</th>
<th>$t_1 + 2 = t_2$</th>
<th>$z^3 = t_2$</th>
<th>$\text{int } y$</th>
<th>$\text{int } z$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$y$</th>
<th>$z$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[-\infty, \infty]$</td>
<td>$[-\infty, \infty]$</td>
<td>$[1, \infty]$</td>
<td>$[1, \infty]$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[1, \infty]$</td>
<td>$[-\infty, \infty]$</td>
<td>$[1, \infty]$</td>
<td>$[1, \infty]$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[1, \infty]$</td>
<td>$[3, \infty]$</td>
<td>$[1, \infty]$</td>
<td>$[1, \infty]$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[1, \infty]$</td>
<td>$[3, \infty]$</td>
<td>$[1, \infty]$</td>
<td>$1.442, \infty$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[1, \infty]$</td>
<td>$[3, \infty]$</td>
<td>$[1, \infty]$</td>
<td>$2, \infty$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[1, \infty]$</td>
<td>$[8, \infty]$</td>
<td>$[1, \infty]$</td>
<td>$2, \infty$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[6, \infty]$</td>
<td>$[8, \infty]$</td>
<td>$[1, \infty]$</td>
<td>$2, \infty$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[6, \infty]$</td>
<td>$[8, \infty]$</td>
<td>$[2.449, \infty]$</td>
<td>$2, \infty$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[6, \infty]$</td>
<td>$[8, \infty]$</td>
<td>$[3, \infty]$</td>
<td>$2, \infty$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[9, \infty]$</td>
<td>$[8, \infty]$</td>
<td>$[3, \infty]$</td>
<td>$2, \infty$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[9, \infty]$</td>
<td>$[11, \infty]$</td>
<td>$[3, \infty]$</td>
<td>$2, \infty$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[9, \infty]$</td>
<td>$[11, \infty]$</td>
<td>$[3, \infty]$</td>
<td>$2.223, \infty$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[9, \infty]$</td>
<td>$[11, \infty]$</td>
<td>$[3, \infty]$</td>
<td>$3, \infty$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[9, \infty]$</td>
<td>$[27, \infty]$</td>
<td>$[3, \infty]$</td>
<td>$3, \infty$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[25, \infty]$</td>
<td>$[27, \infty]$</td>
<td>$[3, \infty]$</td>
<td>$3, \infty$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[25, \infty]$</td>
<td>$[27, \infty]$</td>
<td>$[5, \infty]$</td>
<td>$3, \infty$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[25, \infty]$</td>
<td>$[27, \infty]$</td>
<td>$[5, 5]$</td>
<td>$3, \infty$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$[25, 25]$</td>
<td>$[27, 27]$</td>
<td>$[5, 5]$</td>
<td>$3, 3$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.8: Details of the narrowing operations for the Fermat’s example.
4.1.6 Geometric Examples

In this subsection, we demonstrate the usefulness of the proposed language by developing small primitive procedures in the geometric domain. Interesting theorems such as a unique circle passes through three given points can be verified using these procedures (AlldTr 93). We restrict to integer domains in the examples that follow.

1. **Points on a circle:** The procedure `point_on_circle(a, b, x, y, r)` states the fact that the point $(a, b)$ lies on the circle with center $(x, y)$ and radius $r$. The `main` procedure finds the radius and the center of the circle passing through three given points.

```plaintext
main()
{
  var x, y, r : interval;
  {
    0 <= x; x <= 100;
    0 <= y; y <= 100;
    0 <= r; r <= 100;
    int x, y, r;
    point_on_circle(10, 0, x, y, r);
    point_on_circle(5, 5, x, y, r);
    point_on_circle(15, 5, x, y, r);
    write x, y, r
  }
}

proc point_on_circle(a, b, x, y, r: interval)
{
{
  (((a - x) * (a - x)) + ((b - y) * (b - y)) = r * r
}
}
```

2. **Points on a perpendicular line:** The procedure `perpendicular_to_line(x_1, y_1, x_2, y_2, a_1, b_1, a_2, b_2)` ensures that the line segment joining the points $(x_1, y_1)$ and $(x_2, y_2)$ is perpendicular to the line segment joining the points $(a_1, b_1)$ and $(a_2, b_2)$. Given three coordinates of a triangle, the procedure `main` finds the intersecting point of the perpendicular lines from the vertices to the corresponding faces of the triangle.
main()
{
var a,b : interval;
{
    0 <= a; a <= 100;
    0 <= b; b <= 100;
    perpendicular_to_line(a,b,5,10,0,0,10,0);
    perpendicular_to_line(a,b,10,0,0,0,5,10);
    perpendicular_to_line(a,b,0,0,10,0,5,10);
    write a, b
}
}

proc perpendicular_to_line(x1,y1, x2,y2, a1,b1, a2,b2: interval)
{
{
    ((x2 - x1) * (a2 - a1)) + ((y2 - y1) * (b2 - b1)) = 0
}
}
4.1.7 Scheduling Problems

Figure 4.9: The Precedence Diagram for the Scheduling Problem.

Interval constraints are very useful in solving scheduling problems. In this section, we will examine a simple scheduling problem in which a set of jobs (activities) are to be processed. For each job, the following information is given:

1) The processing time (duration) for each job, and

2) The activities that have to completed before the job is processed.

Let $a_1, a_2, \ldots, a_n$ denote the $n$ activities. The processing time for activity $a_i$ is denoted by $p_i$. Let $s_i$ denote the start time for activity $a_i$. If $a_i$ precedes $a_j$, then $s_j \geq s_i + p_i$.

Let $f_i$ denote the finish time for $a_i$, i.e., $f_i = s_i + p_i$. The following constraints need to be satisfied in order to meet the precedence requirements shown in Figure 4.9:

\[
\begin{align*}
    s_2 &\geq f_1 \\
    s_3 &\geq f_1 \\
    s_4 &\geq f_2 \\
    s_5 &\geq f_3, \, s_5 \geq f_4 \\
    s_6 &\geq f_5 \\
    s_7 &\geq f_3, \, s_7 \geq f_4 \\
    s_8 &\geq f_3, \, s_8 \geq f_4 \\
    s_9 &\geq f_6, \, s_9 \geq f_7, \, s_9 \geq f_8
\end{align*}
\]
The goal is to find the start times for the activities to achieve the minimum time span. The minimum time span, $M$, to schedule and complete all the activities is at most $V = \sum_{i=1}^{n} p_i$. The initial intervals for $M$ and $s_i$, $i = 1 \ldots n$, then become $[0, V]$. Narrowing the above constraints with the duration times, $p_1 = 4$, $p_2 = 1$, $p_3 = 6$, $p_4 = 2$, $p_5 = 1, p_6 = 4$, $p_7 = 2$, $p_8 = 3$, and $p_9 = 2$, the intervals for the start times and the time span are as follows:

$$s_1 = [0, 10] \quad s_2 = [4, 17] \quad s_3 = [4, 14] \quad s_4 = [5, 18] \quad s_5 = [10, 20]$$

$$s_6 = [11, 21] \quad s_7 = [10, 23] \quad s_8 = [10, 22] \quad s_9 = [15, 25] \quad M = [15, 25]$$

Splitting $M$ to get the minimum time span, the intervals narrow to:

$$s_1 = 0 \quad s_2 = [4, 7] \quad s_3 = 4 \quad s_4 = [5, 8] \quad s_5 = 10$$

$$s_6 = 11 \quad s_7 = [10, 13] \quad s_8 = [10, 12] \quad s_9 = 15 \quad M = 15$$

The start times for the activities $a_2, a_4, a_7,$ and $a_8$ are intervals indicating that these are slack variables and can start at any time within their respective interval.
4.2 Non-deterministic Programs

The following example illustrates the functionality of the basic non-deterministic primitives in the language. In contrast with the deterministic programs, the following examples use the `choice`, `split`, and the `fail` statements. The tree in Figure 4.11 illustrates the depth-first nature in which the tree is traversed by the program shown in Figure 4.10. The choices are explored left to right.

```plaintext
main()
{
  var X,Y : interval;
  {
    f(X,Y);
    write X, Y ;
    fail
  }
}

proc f(X,Y: interval)
{
  {
    choice {
      {X = 0}; {X = 1} ; {X = 2}
    };
    choice {
      {Y = 3}; {Y = 4} ; {Y = 5}
    }
  }
}
```

Figure 4.10: A Non-deterministic Program
4.2.1 Fibonacci Numbers

Another example illustrating the non-deterministic features of the language is the computation of the Fibonacci numbers given by the following recurrence equation:

\[ Fib(0) = Fib(1) = 1 \]
\[ Fib(n) = Fib(n - 1) + Fib(n - 2) \]

Expressing the above in a relational form, the procedure \( fib(N,Y) \) shown in Figure 4.12 computes \( Y \) to be the \( N_{th} \) Fibonacci number (and vice versa), where \( Y = Fib(N) \).

An alternate approach of computing the Fibonacci by eliminating the double recursion is shown in the program in Figure 4.13. The computation of the Fibonacci can also be sped up by using memoization (WARR 92).
main()
{
    var N : interval;
    {
        fib(N, 5);
        write N
    }
}

proc fib(n,y : interval)
{
    var y1, y2 : interval;
    {
        choice {
            {n = 0 ; y = 1};
            {n = 1 ; y = 1};
            { 2 <= n; y = y1 + y2; fib(n - 1,y1); fib(n - 2, y2) }
        }
    }
}

Result:

Figure 4.12: Doubly-recursive Fibonacci Program
proc fib(n,y: interval)
{
    var y1: interval;
    {
        choice {
            { n = 0 ; y = 1};
            { 1 <= n;  fib(n,y1,y) }
        }
    }
}

proc fib1(n,y1, y2: interval)
{
    var y3: interval;
    {
        choice {
            { n = 1; y1 = 1; y2 = 1};
            { 2 <= n; y2 = y1 + y3; fib1(n-1,y3, y1) }
        }
    }
}

Figure 4.13: Singly-recursive Fibonacci Program
4.2.2 N Queens

The N queens problem involves placing N queens on a chessboard such that no two queens attack each other. Let $A_i$ denote the column of the $i_{th}$ row (assuming the rows and columns are numbered 1 to $N$ appropriately). The following constraints should be satisfied:

1. $1 \leq A_i \leq N, 1 \leq i \leq N$.

2. $A_i \neq A_j, 1 \leq i < j \leq N$.

3. $A_i \neq A_j + (j - i), 1 \leq i < j \leq N$.

4. $A_i \neq A_j - (j - i), 1 \leq i < j \leq N$.

The program shown in Figure 4.14 generates and solves the above constraints and produces all solutions to the 8-Queens problem.
main()
{ 
    var N : integer;
    var A : array [8] of interval;
    { 
        N = 8;
        bounds(1);
        no_attack(1)
    }
}

proc bounds(i: integer)
{ 
{ 
    choice 
    { (N+1) <= i };
    { i <= N; 1 <= A[i]; A[i] <= N; int A[i]; bounds(i+1) }
}
}

proc no_attack(i: integer)
{ 
{ 
    choice 
    { N <= i };
    { (i+1) <= N; no_attack_1(A[i],i+1,1); no_attack(i+1) }
}
}

proc no_attack_1(x: interval; i,i1: integer)
{ 
{ 
    choice 
    { (N + 1) <= i };
    { i <= N; x <> A[i]; x <> (A[i] + i1); (x + i1) <> A[i];
        no_attack_1(x,i+1, i1 + 1) }
}
}

Figure 4.14: The N-Queens program
4.3 Hidden Markov Models

Consider the example of tossing a coin where the two possible outcomes are head and tail. With a fair coin, there is an equal probability of tossing either a head or a tail. Suppose two coins are used, one being a fair coin and the other a biased one. Assume that the unfair coin is more biased towards having tosses yielding tails. Given:

1. a sequence of heads and tails,

2. the probabilities of a head or a tail outcome for each of the two coins, and

3. the probabilities with which a transition occurs from one coin to the other,

the problem can be modeled using Hidden Markov Models (HMMs) (DURB 98). An HMM can be described using the finite state automaton (FSA) shown in Figure 4.15. In the figure, f and b represent the fair and the biased coin states, respectively. Let h and t denote the head and tail outcomes. The various probabilities are labeled as $P_{ijk}$, $i, k \in \{b, f\}$ and $j \in \{h, t\}$. For example, $P_{fth}$ specifies that the coin is initially in the fair state, the outcome is a tail when the coin is tossed, and the next coin to be
considered is the biased one. In a non-Bayesian approach, the probability of tossing a coin or switching coins are independent of each other.

The problem can be stated as follows. Given a sequence of outcomes (e.g., \( h, t, h, t, t, t, t, h, t, h, t, \ldots \)), determine the sequence of probable states (e.g., \( f, f, f, b, b, \ldots \)) that represent the transition between fair and biased coin tosses. Determining when a state changed occurred is trivial once the sequence of the end states is known. Let \( n \) denote the length of the input sequence \( S \). The probability of being in a (fair or biased) state at the end of the \( s^{th} \) outcome, \( Q_{f_s} \) or \( Q_{b_s} \), is calculated by the following equations:

\[
Q_{f_s} = \max(Q_{f_{s-1}} * P_{f_s|f}, Q_{b_{s-1}} * P_{b_s|f}), \quad \text{and} \\
Q_{b_s} = \max(Q_{f_{s-1}} * P_{f_s|b}, Q_{b_{s-1}} * P_{b_s|b}), \quad s = 2 \ldots n.
\]

\( Q_{f_1} = P_{f_s|f} \) and \( Q_{b_1} = P_{b_s|f} \) where \( x \) is the initial state, \( x \in \{f, b\} \).

In the above equation, \( S \) is the \( s^{th} \) outcome (\( h \) or \( t \)). The first argument of the \( \max \) function is the probability if the previous state is the fair state, and the second argument is the probability if the previous state is the biased state. \( Q_{f_s} \) and \( Q_{b_s} \) are the probabilities of being in the fair state and the biased state at the end of the \( n \) outcomes: the coin is in the fair state if \( Q_{f_s} \) is the maximum of the two values. Otherwise, the coin is in the biased state. To trace back from the final state to the initial state, we need to keep track of the state from which \( Q_{f_s} \) and \( Q_{b_s} \) had their maximum value. This is done using the \textit{pointer} values \( \Gamma_{f_s} \) and \( \Gamma_{b_s} \), \( s = 1 \ldots n \), \( \Gamma_{f_s} \) (\( \Gamma_{b_s} \)) denoting the state \((f \ or \ b)\) from which \( Q_{f_s} \) (\( Q_{b_s} \)) has the maximum value. \( \Gamma_{f_1} = \Gamma_{b_1} = \text{initial state} \), and

\[
\Gamma_{f_s} = \begin{cases} 
  f & \text{if } Q_{f_{s-1}} * P_{f_s|f} > Q_{b_{s-1}} * P_{b_s|f}, \quad s = 2 \ldots n, \\
  b & \text{otherwise}
\end{cases}
\]

93
\[ \Gamma_{b_s} = \begin{cases} 
\text{f} & \text{if } Q_{f_{s-1}} \cdot P_{f_s \cdot b} > Q_{b_{s-1}} \cdot P_{b_s \cdot b} \quad s = 2 \ldots n. \\
\text{b} & \text{otherwise} 
\end{cases} \]

The states, \( \mathcal{E}_s, s = 1 \ldots n \), in which the coin is during the \( n \) steps can then be calculated backwards using the following equations,

\[ \mathcal{E}_n = \begin{cases} 
\text{f} & \text{if } Q_{f_n} > Q_{b_n} \\
\text{b} & \text{otherwise} 
\end{cases} \]

\[ \mathcal{E}_s = \begin{cases} 
\Gamma_{f_s} & \text{if } \mathcal{E}_{s+1} = \text{f} \\
\Gamma_{b_s} & \text{if } \mathcal{E}_{s+1} = \text{b} 
\end{cases}, s = n - 1 \ldots .1. \]

The computing of \( Q_{f_s} \) and \( Q_{b_s} \) requires the cumulative multiplication of probabilities. (e.g., multiplying 0.45 by itself successively results in the following sequence, 0.2025, 0.091125, 0.041006, 0.018453, 0.008304, 0.003737, 0.001682, 0.000757, 0.000341, ... ) The above equations for computing probabilities will only be useful for sequences of trivial lengths. To get accurate results for long sequences, the multiplications can be replaced with summation of logarithms of the probabilities. In the HMM example, we are not interested in calculating the probabilities at the end of each step, but rather in the relative values of the probabilities to determine the greater value. The equations for \( \mathcal{Q}'s \) can be replaced with the following equations:
\[ Q_{f_s} = \max(Q_{f_{s-1}} + \log(P_{j_s} s_j), Q_{b_{s-1}} + \log(P_{k_s} s_j)), \quad \text{and} \]
\[ Q_{b_s} = \max(Q_{f_{s-1}} + \log(P_{j_s} s_j), Q_{b_{s-1}} + \log(P_{k_s} s_j)), \quad s = 2...n. \]
\[ Q_{f_1} = \log(P_{x s_1} f), \quad \text{and} \quad Q_{b_1} = \log(P_{x s_1} b), \]
where \( x \) is the initial state, \( x \in \{ f, b \} \).

The program to compute the states for a given input sequence of head’s and tail’s in the coin tossing example is shown in Figure 4.17. The input sequence of head’s and tail’s is represented by the array \textbf{Sequence} of 1’s (head’s) and 2’s (tail’s). The fair state and the biased state are depicted with 1 and 2. The three dimensional array \textbf{P} represents the probabilities shown in Figure 4.15.

The purpose of using interval constraints is to study the various outcomes of the HMM program taking into account that,

1. the probability of remaining in the fair state represented by the variable \( q \). (\( q = 0.9, 0.7, 0.5, \text{etc.} \)) The probabilities for the fair state are expressed in terms of the variable \( q \).
   \[ P_{f_{n+1} f} = P_{f_1 f} = q, \quad P_{f_{n+1} b} = P_{f_1 b} = \frac{(1-q)}{2}. \]

2. the probability of remaining in the biased state is expressed by the pair of variables \( < r, p > \) in which \( r \) is the bias-“weight” of the biased coin and \( p \) is the transition probability of returning to a fair state. The probabilities for the biased state are expressed in terms of the interval variables \( r \) and \( p \): \( P_{b_{n+1} b} = r * p, P_{b_{n+1} f} = (1-r) * p, \)
   \[ P_{b_{n+1} f} = P_{b_1 f} = \frac{(1-p)}{2}. \]

The input sequence studied in this example has the form, \((ht)^+ t^n (ht)^+\). The goal of the program is to establish the probability regions in which the coin remains always in the fair state irrespective of \( n \) (Region1), and switches from fair to biased state when encountered with a repeating sequence of \( t \)'s, and switches back to the fair state when the sequence becomes fair, i.e., alternating \( h \) and \( t \) (Region2).
<table>
<thead>
<tr>
<th>r</th>
<th>Region 1</th>
<th>Region 2</th>
<th>Region 1</th>
<th>Region 2</th>
<th>Region 1</th>
<th>Region 2</th>
<th>Region 1</th>
<th>Region 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>[0, 0.668]</td>
<td>(0.668, 1]</td>
<td>[0, 0.575]</td>
<td>(0.575, 1]</td>
<td>[0, 0.548]</td>
<td>(0.548, 1]</td>
<td>[0, 0.536]</td>
<td>(0.536, 1]</td>
</tr>
<tr>
<td>0.2</td>
<td>[0, 0.781]</td>
<td>(0.781, 1]</td>
<td>[0, 0.653]</td>
<td>(0.653, 1]</td>
<td>[0, 0.620]</td>
<td>(0.620, 1]</td>
<td>[0, 0.605]</td>
<td>(0.605, 1]</td>
</tr>
<tr>
<td>0.3</td>
<td>[0, 0.873]</td>
<td>(0.873, 1]</td>
<td>[0, 0.760]</td>
<td>(0.760, 1]</td>
<td>[0, 0.716]</td>
<td>(0.716, 1]</td>
<td>[0, 0.696]</td>
<td>(0.696, 1]</td>
</tr>
<tr>
<td>0.4</td>
<td>[0, 0.917]</td>
<td>(0.917, 1]</td>
<td>[0, 0.882]</td>
<td>(0.882, 1]</td>
<td>[0, 0.853]</td>
<td>(0.853, 1]</td>
<td>[0, 0.814]</td>
<td>(0.814, 1]</td>
</tr>
</tbody>
</table>

Table 4.1: Values of the probability \( p \) in the different regions for varying \( n \).

The interval for \( p \) is computed for the two regions described above, for given values of \( r \), (e.g., 0.1, 0.2, 0.3, and 0.4), and for varying lengths of successive \( t \)'s \( (n = 10, 20, 30, \) and \( 40 \)). The various intervals for \( p \) are shown in Table 4.1. The variable \( q \) for the fair state is 0.9.

The intervals for \( p \) can also be computed with a fixed length of the input string and varying \( q \), the probability of remaining in the fair state. Table 4.2 shows the values for \( p \) for various values of \( q \). The number of successive \( t \)'s is 30.

The graph in Figure 4.18 shows the regions in which the coin stays for different values of \( n, r, \) and \( p \). The region to the left of a line (e.g., \( n = 10 \)) is the region in which the coin remains in the fair state (Region 1). The region to the right of the line (e.g., \( n = 10 \)) is the region in which the coin makes a correct transition from the fair state to the biased state and then back to the fair state (Region 2). The graph in Figure 4.19 shows the regions for varying values of \( q, r, \) and \( p \).

This example shows the advantages of interval constraints in establishing regions (sub domains of the variables) for which a given behavior is “stable”, without having to
main() {
  var Sequence: array[200] of integer;
  var p, LogTable: array[2,2] of interval;
  var Q, Prev: array[2, 200] of interval;
  var EndState: array[200] of interval;
  var STEPS, STATES, EVENTS, STARTSTATE, i, j, k, s, v: integer;
  var r, p: interval;
  {
    STATES = 2; EVENTS = 2; read STEPS, STARTSTATE;
    p[1,1] = 0.45; P[1,2,1] = 0.45; P[1,1,2] = 0.05; P[1,2,2] = 0.05;
    P[2,2,1] = r*p; P[2,2,2] = (1-r)*p; P[2,1,1] = 0.5*(1-p); P[2,2,1] = 0.5*(1-p);
    read r;  bind p [0,1];
    for (i = 1; i <= STATES; i = i + 1)
      P[i,1,1] + P[i,2,1] + P[i,1,2] + P[i,2,2] = 1;
    for (i = 1; i <= STATES; i = i + 1)
      for (j = 1; j <= STATES; j = j + 1)
        for (k = 1; k <= EVENTS; k = k + 1)
          LogTable[i,k,j] = log(P[i,k,j]);
    for (s = 1; s <= STEPS; s = s + 1) read Sequence[s];
    for (i = 1; i <= STATES; i = i + 1)
      {
        Q[i,1] = LogTable[STARTSTATE, Sequence[1], i]; Prev[i,1] = STARTSTATE
      };
    for (s = 2; s <= STEPS; s = s + 1)
      {
        v = Sequence[s];
        for (i = 1; i <= STATES; i = i + 1)
          {
            Q[i,s] = max(Q[i,s-1] + LogTable[1,v,i], Q[2,s-1] + LogTable[2,v,i]);
            Prev[i,s] = max_index(Q[i,s-1] + LogTable[1,v,i], Q[2,s-1] + LogTable[2,v,i]);
          }
    };
    for (s = 1; s <= STEPS; s = s + 1)
      {
        bind EndState[s] [1,2]; int EndState[s]; write EndState[s]
      };
    EndState[STEPS] = max_index(Q[1,STEPS], Q[2,STEPS]);
    for (s = STEPS-1; s >= 1; s = s-1)
      EndState[s] = (Prev[1,s] * (2-EndState[s+1])) + (Prev[2,s] * (EndState[s+1]-1))
  }
}

Figure 4.17: Program to compute the states for the coin tossing example.
\begin{table}| r | Region\textsubscript{1} | Region\textsubscript{2} | Region\textsubscript{1} | Region\textsubscript{2} | Region\textsubscript{1} | Region\textsubscript{2} | Region\textsubscript{1} | Region\textsubscript{2} |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>[0, 0.336]</td>
<td>(0.336, 1]</td>
<td>[0, 0.401]</td>
<td>(0.401, 1]</td>
<td>[0, 0.471]</td>
<td>(0.471, 1]</td>
<td>[0, 0.548]</td>
<td>(0.548, 1]</td>
</tr>
<tr>
<td>0.2</td>
<td>[0, 0.379]</td>
<td>(0.379, 1]</td>
<td>[0, 0.453]</td>
<td>(0.453, 1]</td>
<td>[0, 0.531]</td>
<td>(0.531, 1]</td>
<td>[0, 0.620]</td>
<td>(0.620, 1]</td>
</tr>
<tr>
<td>0.3</td>
<td>[0, 0.435]</td>
<td>(0.435, 1]</td>
<td>[0, 0.520]</td>
<td>(0.520, 1]</td>
<td>[0, 0.611]</td>
<td>(0.611, 1]</td>
<td>[0, 0.716]</td>
<td>(0.716, 1]</td>
</tr>
<tr>
<td>0.4</td>
<td>[0, 0.509]</td>
<td>(0.509, 1]</td>
<td>[0, 0.610]</td>
<td>(0.610, 1]</td>
<td>[0, 0.721]</td>
<td>(0.721, 1]</td>
<td>[0, 0.853]</td>
<td>(0.853, 1]</td>
</tr>
</tbody>
</table>
\end{table}

Table 4.2: Values of the Probability $p$ in the different regions for varying $q$.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure4.18.png}
\caption{Plot for different regions of the Probabilities for varying $n$ and $q = 0.9$.}
\end{figure}
Figure 4.19: Plot for different regions of the probabilities for varying $q$ and $n = 30$. Recompute values corresponding to every single point in the interval.
4.4 Computationally Difficult Problems

In this section we describe two finite-domain problems which require considerable amount of computation. The 3-SAT problem involves determining the satisfiability of a set of boolean constraints in clausal form, each clause containing three disjuncts. As the ratio of the number of clauses \((m)\) to the number of variables \((n)\) increases, random instances of 3-SAT progress from formulæ which are generally satisfiable to formulæ which are generally not satisfiable, with an apparent sharp threshold being crossed when \(m/n = 4.25\) (DuBe 97). Using the right distribution of instances and appropriate parameter values, it is possible to generate random formulæ that are hard for which satisfiability is quite hard (ScMiLe 96). In (CrAu 96), the authors observe that the hardest satisfiability problems are those that are critically constrained, i.e., neither under- constrained that they have many solutions, nor so over- constrained that the search tree is small. The authors also prove that the sharp threshold is reached for \(m = 4.258n + 58.26n^{2/3}\). In (FREE 96), the author shows why the Davis and Putnam method (DaPu 60) for testing the satisfiability performs poorly on hard random 3-SAT problem, observing that the Davis and Putnam method fails to significantly reduce the problem size in the uppermost portion of the search tree. A detailed experimental investigation of the phase transition for different classes of satisfiability problems including k-SAT is presented in (GeWa 94).

The second problem is the job-shop sequencing problem in which a set of jobs are to be scheduled on a given number of machines in such a way that the processing times of any two jobs on a given machine are disjoint.
4.4.1 3-SAT

The Boolean constraints are assumed to be in the clausal form. Let $S$ be the set of clauses $\{C_1, C_2, \ldots, C_m\}$, where each clause $C_i$ is of the form $\{c_{i_1}, c_{i_2}, \ldots, \}$. The set $S = \{\{c_{i_1}, c_{i_2}, \ldots, \}, \{c_{2_1}, c_{2_2}, \ldots, \}, \ldots, \{c_{m_1}, c_{m_2}, \ldots, \}\}$ represents the Boolean constraint

$B_S = (c_{i_1} \lor c_{i_2} \lor \ldots) \land (c_{2_1} \lor c_{2_2} \lor \ldots) \land \ldots \land (c_{m_1} \lor c_{m_2} \lor \ldots)$. Let $V = \{x_1, x_2, \ldots, x_n\}$ be the set of variables appearing in the set $S$. The $c_{i_j}'s$ are either the positive or negative form of the variables, $x_k$ or $\neg x_k$.

It is known that an $n$-SAT problem can be transformed to a 3-SAT problem with an increased number of clauses. Furthermore, it is known that 2-SAT problem has polynomial complexity. Therefore 3-SAT problems correspond to the simplest regular form of boolean formulas that are in NP.

Let us restrict our attention to the 3-SAT problem. Each clause $C_i$ has at most three literals. Given a set of $m$ clauses ($S = \{C_1, C_2, \ldots, C_m\}$) with $n$ variables ($V = \{x_1, x_2, \ldots, x_n\}$), where each clause $C_i$ is of the form $C_i = \{c_{i_1}, c_{i_2}, c_{i_3}\}$. Further assume that there is an equal probability (0.5) that a variable in a given clause is positive or negative. It has been found recently that these 3-SAT problems enjoy very puzzling properties. It is intuitively clear that when $\frac{m}{n}$ is very small, the probability of satisfiability is very high (say 99%) and the satisfiability can be determined in a relatively short time. When $\frac{m}{n}$ is large, the probability of satisfiability is very low (say 1%) and again the determination of unsatisfiability is not time consuming. The amazing fact occurs when $\frac{m}{n} = 4.25$. For that value, the probability of satisfiability is 50% and the time for determining satisfiability attains a peak (Figure 4.20.)
Davis and Putnam Method

The Davis and Putnam method (DP-method) is useful in determining the satisfiability of Boolean constraints. The DP-method essentially consists of the following four rules.

1. **Tautology Rule:** Delete all clauses from $S$ that are tautologies, i.e., those clauses in which both the literals $x_k$ and $\neg x_k$ appear. The remaining set $S'$ is unsatisfiable if and only if $S$ is.

2. **One-Literal Rule:** If there is a unit clause $C_i = \{c_i\}$ in $S$, then obtain the set $S'$ by deleting the clauses in $S$ which contain $c_i$. If $S'$ is empty, then the set $S$ is satisfiable. Otherwise, obtain the set $S''$ from $S'$ by deleting $\neg c_i$ from $S'$. $S''$ is unsatisfiable if and only if $S$ is. However, if a clause $C_j$ contains the unit clause $\{\neg c_i\}$, then the set $S$ is unsatisfiable.

3. **Pure-Literal Rule:** Suppose the set $S$ contains a literal $c_{ij}$ such that the literal $\neg c_{ij}$ does not appear in any clause in $S$. In that case, obtain the set $S'$ by deleting all the clauses containing $c_{ij}$. The set set $S'$ is unsatisfiable if any only if $S$ is.

4. **Splitting Rule:** If the constraint $B_s$ can be rearranged into the form
\[(x_i \lor P_1) \land \ldots \land (x_i \lor P_r) \land (\neg x_i \lor Q_1) \land \ldots \land (\neg x_i \lor Q_s) \land R,\]

where \(x_i\) and \(\neg x_i\) do not appear in \(P_i, Q_i,\) and \(R,\) then obtain the sets \(S_1 = P_1 \land \ldots \land P_r \land R\) and \(S_2 = Q_1 \land \ldots \land Q_s \land R.\) \(S\) is unsatisfiable if and only if both \(S_1\) and \(S_2\) are unsatisfiable.

To test the satisfiability, the tautology rule is first applied. The one-literal rule is applied repeatedly until it is applicable. The same with the pure-literal rule. The splitting rule is then applied and the above procedure is repeated with the two clausal sets.

**Davis and Putnam with Interval Narrowing**

The one-literal rule combined with the interval narrowing technique can be efficiently used to test the satisfiability of the given set of Boolean constraints. Given the Boolean constraints \(\{C_1, C_2, \ldots, C_m\}\) in the 3-SAT form, we generate \(m\) constraints \(C_1 = 1; C_2 = 1; \ldots; C_m = 1.\) The \(m\) constraints are stored in a constraint table. Let \(x_1, x_2, \ldots, x_n\) be the variables appearing in the constraint set. The constraint table is list of triplets. If \(C_i = \{c_{i_1}, c_{i_2}, c_{i_3}\},\) then \(table_i = (v_1, v_2, v_3),\) where \(v_j = k\) if \(c_{i_j} = x_k,\) or \(v_j = -k\) if \(c_{i_j} = \neg x_k.\) The interval vector \((vec)\) for the variables \(x_1, x_2, \ldots, x_n\) is initialized with the interval \((0, 1).\) (Note that in this section, the notation \((x, y)\) is used to denote the boolean interval.)

In the proposed algorithm, instead of keeping track of the unit clauses, we keep a record of the variables that are bound to the constants 0 or 1. These variables are kept in a queue (or a stack). When processing a constraint, if the narrowing step narrows any of the variables, i.e., the interval \((0, 1)\) is narrowed to either 0 or 1, the corresponding variable is added to the queue (if it is not already there). For each variable in the queue,
each constraint in the constraint table is examined to determine if the variable occurs positively or negatively. The table entry is either made passive or is modified by deleting the variable as the case may be. The narrowing step is used to ascertain if any of the remaining variables in that constraint get narrowed.

If the queue is empty, a variable whose interval is \((0, 1)\) and still appears in the constraint table is chosen for splitting. The above steps are repeated with the appropriate interval vectors.

**Example:** Given the clauses

\[
\{ \{x_1, x_2, x_3\}, \{x_1, x_2, \neg x_3\}, \{x_1, \neg x_2, x_3\}, \{x_1, \neg x_2, \neg x_3\}, \\
\{-x_1, x_2, x_3\}, \{-x_1, x_2, \neg x_3\}, \{-x_1, \neg x_2, x_3\}, \{-x_1, \neg x_2, \neg x_3\}\},
\]

the constraint table representation is as follows:

\[
table = ((1, 2, 3), (1, 2, -3), (1-2, 3), (1, -2, -3), (-1, 2, 3), (-1, 2, -3), (-1, -2, 3), (-1, -2, -3)).
\]

The steps showing the unsatisfiability of the above constraint set are illustrated in Figure 4.21.
Figure 4.21: 3-SAT Example.
procedure \texttt{3-sat(table, vec, unit\_var\_queue)}

begin

while not empty(unit\_var\_queue) do

var $\leftarrow$ dequeue(unit\_var\_queue)

\textbf{switch} process(table, var, vec, unit\_var\_queue)

\hspace{1em} case SAT :

\hspace{2em} print(Satisfiable); exit()

\hspace{1em} case UNSAT :

\hspace{2em} print(UnSatisfiable); return UNSAT

\hspace{1em} otherwise:

\hspace{2em} break

\textbf{end switch}

\textbf{end while}

if empty(table) then print(Satisfiable); exit() else

var $\leftarrow$ branch(table, vec)

queue$_1$ $\leftarrow$ empty; enqueue(var, queue$_1$);

table$_1$ $\leftarrow$ table; vec$_1$ $\leftarrow$ vec; vec$_1$$_{var}$ $\leftarrow$ 0

queue$_2$ $\leftarrow$ empty; enqueue(var, queue$_2$);

table$_2$ $\leftarrow$ table; vec$_2$ $\leftarrow$ vec; s vec$_2$$_{var}$ $\leftarrow$ 1

\texttt{3-sat(table$_1$, vec$_1$, queue$_1$)};

\texttt{3-sat(table$_2$, vec$_2$, queue$_2$)};

\textbf{end if}

\textbf{end 3-sat}
procedure process(table, var, vec, queue)
begin
val ← vec[\text{var}]
passive.\text{ctr} ← m
for i ← 1 to m
    if not passive(table_i) then
        if val = 1 then
            if occurs(var, table_i) then
                passive(table_i) ← true; passive.\text{ctr} ← passive.\text{ctr} - 1
            else if occurs(\text{-}var, table_i) then
                j ← pos(\text{-}var, table_i); table_{ji} ← 0
                if narrow(table_i, vec, queue) = UNSAT then
                    return UNSAT
            end if
        end if
    end if
else if val = 0 then
    if occurs(\text{-}var, table_i) then
        passive(table_i) ← true; passive.\text{ctr} ← passive.\text{ctr} - 1
    else if occurs(var, table_i) then
        j ← pos(var, table_i); table_{ji} ← 0
        if narrow(table_i, vec, queue) = UNSAT then
            return UNSAT
    end if
end if
end if
end if

else passive_ctr ← passive_ctr − 1

end if

end for

if passive_ctr = 0 then return SAT else return UNKNOWN

end process
<table>
<thead>
<tr>
<th>Given</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>$vec_{F+}$</td>
</tr>
<tr>
<td>+ (0,1)</td>
<td>true</td>
</tr>
<tr>
<td>- (0,1)</td>
<td>true</td>
</tr>
<tr>
<td>- (1,1)</td>
<td>unsat</td>
</tr>
<tr>
<td>+ (0,0)</td>
<td>unsat</td>
</tr>
<tr>
<td>Other Cases</td>
<td>return true</td>
</tr>
</tbody>
</table>

Table 4.3: Narrow with two variables.

procedure narrow(table entry, vec, queue)
begin
    ($v_1$, $v_2$) ← rearrange(table entry)
    if $v_2 = 0$ then
        if $v_1 = 0$ then return UNSAT
        else Consult Table 4.3
    else Consult Table 4.4
end narrow
<table>
<thead>
<tr>
<th>$v_1$</th>
<th>$v_2$</th>
<th>$\vec{e}_C[v_1]$</th>
<th>$\vec{e}_C[v_2]$</th>
<th>Return</th>
<th>$\vec{e}_C[v_1]$</th>
<th>$\vec{e}_C[v_2]$</th>
<th>enqueue</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>(0,1)</td>
<td>(0,0)</td>
<td>true</td>
<td>(1,1)</td>
<td>(0,0)</td>
<td>$[v_1]$</td>
</tr>
<tr>
<td>-</td>
<td>+</td>
<td>(0,1)</td>
<td>(0,0)</td>
<td>true</td>
<td>(0,0)</td>
<td>(0,0)</td>
<td>$[v_1]$</td>
</tr>
<tr>
<td>+</td>
<td>-</td>
<td>(0,1)</td>
<td>(1,1)</td>
<td>true</td>
<td>(1,1)</td>
<td>(1,1)</td>
<td>$[v_1]$</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>(0,1)</td>
<td>(1,1)</td>
<td>true</td>
<td>(0,0)</td>
<td>(1,1)</td>
<td>$[v_1]$</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>(0,0)</td>
<td>(0,1)</td>
<td>true</td>
<td>(0,0)</td>
<td>(1,1)</td>
<td>$[v_2]$</td>
</tr>
<tr>
<td>+</td>
<td>-</td>
<td>(0,0)</td>
<td>(0,1)</td>
<td>true</td>
<td>(0,0)</td>
<td>(0,0)</td>
<td>$[v_2]$</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>(0,0)</td>
<td>(0,0)</td>
<td>unsat</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>-</td>
<td>(0,0)</td>
<td>(1,1)</td>
<td>unsat</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>+</td>
<td>(1,1)</td>
<td>(0,1)</td>
<td>true</td>
<td>(1,1)</td>
<td>(1,1)</td>
<td>$[v_2]$</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>(1,1)</td>
<td>(0,1)</td>
<td>true</td>
<td>(1,1)</td>
<td>(0,0)</td>
<td>$[v_2]$</td>
</tr>
<tr>
<td>-</td>
<td>+</td>
<td>(1,1)</td>
<td>(0,0)</td>
<td>unsat</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>(1,1)</td>
<td>(1,1)</td>
<td>unsat</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Other Cases: return true

Table 4.4: Narrow with two variables.
4.4.2 Job-Shop Scheduling

The scheduling problem described in Section 4.1.7 is often extended to include disjunctive constraints and it is known as the job-shop scheduling problem. The problem was proposed in 1963 (MuTh 63). An optimal solution for the job-shop scheduling problem was first solved in the year 1989 (CaPi 89). The job-shop problem is to schedule a set of jobs on a set of machines with the following constraints:

1. each machine can handle at most one job at a time,

2. each job has a specified processing order through the machines, i.e., the job precedences are to be obeyed.

More precisely, let $J$ be the number of jobs to be processed on a set of $M$ machines. The processing order for each job, $j \in J$, is represented by the sequence $(\sigma^j_1, \sigma^j_2, \ldots, \sigma^j_m)$, where $m = |M|$. The job $j$ is processed in sequence first on the machine $\sigma^j_1$, then on $\sigma^j_2$, etc. The processing time for each job $j$ on machine $\alpha$ is denoted by $p_{j\alpha}$. The goal is to find a schedule of $J$ on $M$ that minimizes the maximum of the completion times of the jobs $J$.

The constraints in the job-shop scheduling problem can be stated as follows. For each job $j$ and each machine $\alpha$, let $x_{j\alpha}$ be the starting time of $j$ on $\alpha$. (The finish time is $x_{j\alpha} + p_{j\alpha}$.) Then,

1. the start times are positive,
   
   $x_{j\alpha} \geq 0$ for all $j \in J, \alpha \in M$.

2. the start time of any job on a given machine should be greater than the finish time of the same job on the preceding machine in the job sequence,

   $x_{j\sigma^j_t} \geq x_{j\sigma^j_{t-1}} + p_{j\sigma^j_{t-1}}$ for all $j \in J$ and $t = 2, \ldots, m$. 

111
3. For any two jobs, the start time of a job on a given machine should be greater than
the finish time of any other job on the same machine,

\[ x_{ia} \geq x_{ja} + p_{ja} \quad \text{or} \quad x_{ja} \geq x_{ia} + p_{ia}, \quad \text{for all } i, j \in J, \alpha \in M. \]

The time span for processing all the jobs is

\[ T = \max(x_{j \sigma_m^j} + p_{j \sigma_m^j}). \]

The job-shop problem is to find the minimum time span.

The disjunctive constraint

\[ x_{ia} \geq x_{ja} + p_{ja} \quad \text{or} \quad x_{ja} \geq x_{ia} + p_{ia} \]

can be transformed into a conjunctive constraint by introducing a \((0, 1)\) boolean variable

\[ Y_{ij}^\alpha. \]

\[ x_{ia} \geq x_{ja} + p_{ja} - K \cdot Y_{ij}^\alpha \]

\[ x_{ja} \geq x_{ia} + p_{ia} - K \cdot (1 - Y_{ij}^\alpha) \]

for all \(i, j \in J, \alpha \in M\), where \(K\) is a large positive constant. In the above constraint,

\(Y_{ij}^\alpha = 1\) implies \(i\) is scheduled before \(j\) on machine \(\alpha\) and \(Y_{ij}^\alpha = 0\) implies \(j\) is scheduled

before \(i\).

\textbf{Example:} Consider the job-shop scheduling problem where 4 jobs are to be executed

on 3 machines (Figure 4.22). Each job consists of three operations. In the example shown

in Figure 4.22, the three operations in job 4 are to be executed in sequence on machines

1, 3, and 2 respectively. The pair \((a, b)\) within each node represents the processing of job

\(a\) on machine \(b\). The processing order for the four jobs is as follows:

\(\sigma^1 = (1, 2, 3), \sigma^2 = (2, 1, 3), \sigma^3 = (3, 2, 1), \sigma^4 = (1, 3, 2).\)

The above processing order enforces the following constraints (see the unidirectional arrows in Figure 4.22):

\[ x_{12} \geq x_{11} + p_{11}, \quad x_{13} \geq x_{12} + p_{12} \]
\[ x_{21} \geq x_{22} + p_{22}, \quad x_{23} \geq x_{21} + p_{21} \]
\[ x_{32} \geq x_{33} + p_{33}, \quad x_{31} \geq x_{32} + p_{32} \]
\[ x_{43} \geq x_{41} + p_{41}, \quad x_{42} \geq x_{43} + p_{43} \]

The disjunctive constraints for machine, say 1, are as follows (see the bidirectional arrows in Figure 4.22):

\[ x_{11} \geq x_{21} + p_{21} \text{ or } x_{21} \geq x_{11} + p_{11} \]
\[ x_{11} \geq x_{31} + p_{31} \text{ or } x_{31} \geq x_{11} + p_{11} \]
\[ x_{11} \geq x_{41} + p_{41} \text{ or } x_{41} \geq x_{11} + p_{11} \]
\[ x_{21} \geq x_{31} + p_{31} \text{ or } x_{31} \geq x_{21} + p_{21} \]
\[ x_{21} \geq x_{41} + p_{41} \text{ or } x_{41} \geq x_{21} + p_{21} \]
\[ x_{31} \geq x_{41} + p_{41} \text{ or } x_{41} \geq x_{31} + p_{31} \]

### 4.4.3 Constraint Satisfaction Problems (CSP)

To illustrate the expressiveness and granularity of the language, we consider how any CSP problem (MACK 77) can be easily transformed into a program in the language. We first review the properties of CSP and the algorithms used in solving the CSPs. With the help of an example, we show the translation of CSP constraints into constraints in the language.

A constraint satisfaction problem (CSP) is characterized as follows—given is a set \( V \) of \( n \) variables \( \{v_1, v_2, \ldots, v_n\} \), and associated with each variable \( v_i \) is a domain \( D_i \) of possible values. On some specified subsets of these variables, there are constraint relations. The set of solutions is the largest subset of the cartesian product of all the
Figure 4.22: A Job-Shop scheduling problem with 3 machines and 4 jobs.
given variable domains \(D = D_1 \times D_2 \times \ldots \times D_n\) such that each \(n\)-tuple in that set satisfies all the given constraint relations. If the set of solutions is empty, the CSP is unsatisfiable.

An example of CSP is the map coloring problem. The problem is to decide if three colors suffice to color a given planar map such that each region is a different color from each of its neighbors. This is formulated as a Boolean CSP by creating a variable for each region to be colored, associating with each variable the domain \{red, green, blue\}, and requiring that each pair of adjacent regions have different colors.

The satisfiability decision problem for CSP is equivalent to determining the truth of the following formula in first-order predicate logic:

\[
\exists x_1 \exists x_2 \ldots \exists x_n (x_1 \in D_1) \land (x_2 \in D_2) \land \ldots \land (x_n \in D_n) \land P_1(x_1) \land P_2(x_2) \land \ldots \land P_n(x_n) \land P_{12}(x_1, x_2) \land P_{13}(x_1, x_3) \land \ldots \land P_{n-1,n}(x_{n-1}, x_n).
\]

In the case of a crossword puzzle example, the unary constraints \(\{P_i\}\) specify the word length. The binary constraints arise when a word ACROSS intersects a word DOWN. For example, the constraint \(P_1\) can specify that the word starting at 1 ACROSS has 5 letters. \(P_{12}\) can be specifying that the third letter of word 1 ACROSS be the same as the first letter of word 2 DOWN. In general, \(p\)-ary predicates are required \((1 \leq p \leq n)\).

There are three primary ways to solve the constraint satisfaction problems, generate and test methods, backtrack algorithms and consistency algorithms.

**Generate and Test Algorithms**

In the generate and test methods, all the \(n\)-tuples in the cartesian product of the variable domains are successively generated and tested for satisfiability of the CSP problem. This is very inefficient since one may have to explore \(|D_1| \times |D_2| \times \ldots \times |D_n|\) possible
candidates.

**Backtrack Algorithms**

These algorithms explore the domain $D$ by sequentially instantiating the variables in some order. As soon as any predicate has all its variables instantiated, its truth value is determined. If that predicate is false, the partial assignment constructed so far cannot be part of any valid total assignment. Backtracking then selects the last variable with unassigned values remaining in its domain (if any) and instantiates it to its next value. The efficiency gain from backtracking arises from the fact that a potentially large subspace of $D$ is eliminated by a single predicate failure.

**Consistency Algorithms**

Complementary to the class of backtracking algorithms, the consistency algorithms are described by means of constraint networks. Each variable is represented by a vertex with its associated domain attached. An edge between the vertices corresponds to each pair of directly constrained variables. For binary constraints, each edge in the graph between vertices $i$ and $j$ is replaced by arc $(i, j)$ and arc $(j, i)$.

Node $i$, composed of vertex $i$ and the associated domain of variable $v_i$, is node consistent iff

$$\forall x(x \in D_i) \supset P_i(x).$$

Each node can trivially be made consistent by performing the domain restriction operation:

$$D_i \leftarrow D_i \cap \{x \mid P_i(x)\}.$$  

In the crossword puzzle, this corresponds to deleting from each variables domain any word of wrong length.
Similarly, arc \((i, j)\) is arc consistent iff

\[
\forall x (x \in D_i) \supset \exists y (y \in D_j) \cap P_{ij}(x, y).
\]

i.e., for every element in \(D_i\) there is at least one element in \(D_j\) such that the pair of elements satisfy the constraining predicate. Arc \((i, j)\) can be made arc consistent by removing from \(D_i\) all elements that have no corresponding element in \(D_j\) with the following arc consistency domain restriction operation:

\[
D_i \leftarrow D_i \cap \{x \mid \exists y (y \in D_j) \land P_{ij}(x, y)\}
\]

A network is node and arc consistent iff all its nodes and arcs are consistent.

### 4.5 Examples of Possible Preprocessing

The following two subsections explore the possibility of developing preprocessors to transform problems into the proposed language.

#### 4.5.1 Transforming CSP problems

In this section, we illustrate through a crypto-arithmetic example how a CSP problem can be translated into interval constraints. The problem is to assign distinct numerals (between 0 and 9) to the alphabets occurring in the equation,

\[
SEND + MORE = MONEY,
\]

such that the equality is satisfied. The variables in the CSP problem are \(V = \{s, e, n, d, m, o, r, y\}\), where each variable stands for the respective character. Since \(S\) and \(M\) are in the most significant positions, the domains for the variables \(s\) and \(m\) are restricted to the range 1..9, whereas all the other variables have the domain 0..9. The unary predicates specify the domain constraints for the variables:

\[
P_i(s) : 1 \leq s \leq 9, \quad P_i(e) : 0 \leq e \leq 9, \quad P_i(n) : 0 \leq n \leq 9, \quad P_i(d) : 0 \leq d \leq 9,
\]

117
\[ P_1(m) : 1 \leq m \leq 9, \ P_1(o) : 0 \leq o \leq 9, \ P_1(r) : 0 \leq r \leq 9, \ P_1(y) : 0 \leq y \leq 9. \]

The binary predicates specify that any two distinct variables cannot have the same value:

\[ P_2(s, e) : s \neq e, \ P_2(s, n) : s \neq n, \ P_2(s, d) : s \neq d, \text{ etc.} \]

Also, we need to introduce variables which represent the carry over values for the additions in each position of the equation to be solved. Let \( \{c_0, c_1, c_2, c_3, c_4\} \) be the carry values in the unit's, ten's, ..., positions, respectively. The predicate \( \text{adder}(x, y,\ carry_{in}, z,\ carry_{out}) \) is true if and only if \( x + y + carry_{in} = 10 \times carry_{out} + z \). The rest of the predicates for the CSP problem can then be specified as follows:

\[ P_3(d, e, c_0, y, c_1) : \text{adder}(d, e, c_0, y, c_1), \ P_4(n, r, c_1, e, c_2) : \text{adder}(n, r, c_1, e, c_2), \ P_5(e, o, c_2, n, c_3) : \text{adder}(e, o, c_2, n, c_3), \ P_6(s, m, c_3, o, c_4) : \text{adder}(s, m, c_3, o, c_4), \ P_7(z, z, c_4, m, z) : \text{adder}(z, z, c_4, m, z). \]

The variable \( z \) represents the value 0. \( c_0 \) has the domain \( \{0\} \), whereas the other carry variables have the domain \( \{0, 1\} \).

The above CSP problem can be translated into interval constraints as follows:

1. The unary predicates specifying the domains for the variables can be translated into \texttt{BIND} statements if the domains are continuous, e.g., \texttt{BIND s (1, 9), int s, BIND e(0, 9), int e}, etc. If the domains are not continuous, the \( \leq \) relation can be used to explicitly constrain the domains.

2. The binary predicates which state that the values of all variables be distinct are translated into \( \neq \) constraints, e.g, \( s \neq e, s \neq n, \text{ etc.} \)

3. The 5-ary predicates depicting the addition relationships are transformed into equality constraints, e.g., \( d + e + c_0 = 10 \times c_1 + y, \ n + r + c_1 = 10 \times c_2 + e, \text{ etc.} \)

Solving the constraints yields the solution \( < s, e, n, d, m, o, r, y > = < 9, 5, 6, 7, 1, 0, 8, 2 >. \)
The interval narrowing operations subsume the consistency algorithms for CSP problems. The splitting phase in enumerating the solution is similar to the technique employed in the backtrack algorithms.

4.5.2 Partial Evaluation

The example presented in this section shows that the partial evaluation techniques can be applied to programs in the proposed language. By partially evaluating the program through specialization with respect to a partial set of known input values, a much simpler program results thus improving the run-time performance.

The program shown in Figure 4.23 computes the dot product of two vectors \( x \) and \( y \). The variable \( N \) denotes the length of the vectors.

```plaintext
main()
{
    var N: integer;
    var x, y: array[100] of interval;
    var result: interval;
    {
        N = 4;
        dot_product(1, result);
        write result
    }
}

proc dot_product(i: integer; res: interval)
{
    var temp: interval;
    {
        choice {
            { i == N; res = x[i] * y[i] };
            { i < N; dot_product(i+1, temp); res = (x[i] * y[i]) + temp}
        }
    }
}
```

Figure 4.23: Program to compute the dot-product.
The program can be partially evaluated at compile-time when the size of the input vectors is known. Assuming \( N = 4 \), the specialized program shown in Figure 4.24 can be produced:

```plaintext
main()
{
    var N: integer;
    var x, y: array[100] of interval;
    var result, t1, t2, t3, t4: interval;
    {
        N = 4;
        t1 = x[4] * y[4];
        t4 = x[1] * y[1] + t3;
        result = t4;
        write result
    }
}
```

Figure 4.24: Specialized version of the dot-product program when \( N = 4 \).

If one of the input vectors is known to have, say, integer values, the program can be further specialized. Suppose the vector \( x \) has the values \( \{2, 7, 3, 5\} \). Partially evaluating the above program, we get a much simplified version as shown in Figure 4.25.

```plaintext
main()
{
    var N: integer;
    var x, y: array[100] of interval;
    var result, t1, t2, t3, t4: interval;
    {
        N = 4;
        t1 = 5 * y[4];
        t2 = 3 * y[3] + t1;
        t3 = 7 * y[2] + t2;
        t4 = 2 * y[1] + t3;
        result = t4;
        write result
    }
}
```

Figure 4.25: Specialized version of the dot-product program with a known input vector.
Chapter 5

Semantics

The formal semantics of a programming language is useful not only in building a mathematical model but also serves as a basis for understanding and reasoning about program behavior; furthermore, a formal definition provides accurate information to the compiler designer. We consider three approaches for specifying the semantics of the proposed language.

- Operational semantics, which describes the meaning of a programming language by specifying how the programs written in the language execute on an abstract machine. The operational approach considered in this work has been advocated by Gordon Plotkin (PLOT 81).

- Denotational semantics, which defines the meaning of a programming language using abstract mathematical concepts of complete partial orders, continuous functions and least fixed points. The denotational approach was pioneered by Christopher Strachey and Dana Scott (SCOT 82; STOY 77). Introductory presentations of denotational semantics are found in (GORD 88; LoSi 84; SCHM 86).
• Axiomatic semantics, which describes the meaning of a programming construct by providing proof rules for that construct within a programming logic. Extensive treatment of axiomatic semantics can be found in (APT 81).

The above approaches for defining the semantics are highly dependent on each other. Proving that the rules of an axiomatic semantics are correct relies on an underlying denotational or operational semantics. Proving an implementation correct requires the operational and denotational semantics be consistent. The denotational semantics has the advantage of abstracting the implementation details.

Only the operational semantics and denotational semantics (GUNT 92) of the constructs developed in this thesis are described in this chapter. The operational semantics is presented in Section 5.1. Section 5.2 describes the denotational semantics of the language. The axiomatic semantics of the proposed language is not developed in this work.

5.1 Operational Semantics

The syntax of a subset of the language described in this work is captured by the following grammar:

\[ P ::= S; \ldots; S \]

\[ S ::= E = E | E \leq E | E \neq E | BIND v (a, b) | CHOICE \{S; \ldots; S\} | SPLIT v (E, E)(E, E) \]

\[ E ::= c | v | L(v) | U(v) | E + E | E - E | E * E | E / E | \neg E | E \land E | E \lor E \]

\[ P \] denotes the program, \( S \) the statements, and \( E \) the expressions. \( x, y, z, v \) denote variables and \( a, b, c \) denote constants. The \textit{domain considered here} is the set of real numbers.
\[ R. \]

The results of the program are obtained by solving the constraints embodying the statements in the program. The constraints are solved using narrowing operations. A narrowing operation considers a constraint and attempts to reduce the interval domain of its variables. For any variable \( x \), let \( \sigma(x) \) denote the interval for \( x \), i.e., \( \sigma(x) = (L(x), U(x)) \), where \( L \) and \( U \) represent the numeric values of the lower and upper bounds respectively. Let \( \text{var}(S) \) denote the set of variables appearing in a (single) constraint \( S \). For each \( x_i \in \text{var}(S) \), let \( \text{N}(x_i, S) \) represent the narrowed interval domain after applying the narrowing operation for the constraint \( S \). Let \( \sigma \) denote the interval vector. The notation \( \sigma[i_x/x] \) represents the interval vector obtained by replacing the current interval for the variable \( x \) with the new interval \( i_x \).

**Definition:** A state of the constraint solver is defined as a pair \( \delta = \langle S, \sigma \rangle \), where \( S \) is the set of constraints to be processed, and \( \sigma \) is the interval-vector comprising the interval domains of the variables appearing in the source program \( P \).

**Notation:** Let \( P^x \) denote the set of constraints in \( P \) which contain the variable \( x \).

**Definition:** A transformation is defined as the change of the given state by a single step of the constraint solver. The transformation of the state \( \langle S, \sigma \rangle \) is represented in the following way:

\[ \langle S, \sigma \rangle \rightarrow \langle S', (b, \sigma') \rangle, \quad b \in \{\text{true, false}\}, \quad |S'| \leq |S|. \]

If \( b \) is \text{true}, the new state of the constraint solver is \( \delta' = \langle S', \sigma' \rangle \), otherwise a failure state is reached.

The constraint solver is described by an automaton whose behavior is emulated by the program shown in Figure 5.1.

The operational semantics of the language constructs is presented below. The **BIND** rule
Input: $< \mathcal{S}, \sigma >$ (Set of constraints $\mathcal{S}$ and interval vector $\sigma$)

\[ b \leftarrow \text{true} \]

while $b$ and $\mathcal{S} \neq \emptyset$ do

narrow:

\[ < \mathcal{S}, \sigma > \rightarrow < \mathcal{S}', (b', \sigma') > \]

if $b'$ then

\[ ( < \mathcal{S}, \sigma >, b ) \rightarrow ( < \mathcal{S}', \sigma' >, b' ) \]

else failure; exit

end while

Output: $\sigma$

Figure 5.1: The Solve Automaton.

specifies that binding a variable to an interval can only be done once, by narrowing the
$(-\infty, \infty)$ interval to the given bounds. The CHOICE statement specifies that one of the
alternatives is selected non-deterministically. The other alternatives are explored either
by parallel exploration of the choice branches or by backtracking. The SPLIT statement
specifies how the values of a variable can be split into two interval components and
explored in parallel.

The constraints are translated to the form $op(x, y, z)$, where $op$ is a ternary,
binary, or a unary operator, and the variables $x, y,$ and $z$ are the operands. (In the case
of binary and unary constraints, the extra operands are irrelevant.) Before performing
the narrowing operation, the constraints are checked to determine if they are passive or
not. A constraint is termed passive if the interval values of the operands are such that
the constraint is always satisfiable. If a constraint is determined to be passive, it is added
to the passive constraint set $P^{\text{passive}}$.

The narrow operation considers a constraint and determines the narrowed intervals for the operands. If the intervals of any of the operands differ from their original value, all the non-passive constraints in which the respective operands occur are added to the new set of constraints for which narrowing has to be redone.

**Notation:** The list $c_1 \cdot [c_2, \ldots, c_n]$ represents the set $\{c_1, c_2, \ldots, c_n\}$. $S_1 \cup S_2$ represents the union of the two sets.

**Initialize Variables:**

\[
< \text{BIND } x \ (a, b) \cdot S, \ \sigma > \rightarrow \begin{cases} 
< S, \ (\text{true, } \sigma[(a, b)/x]) > \\
\quad \quad \text{if } \sigma(x) = (-\infty, \infty), \\
< S, \ (\text{false, } \sigma) > \\
\quad \quad \text{otherwise.}
\end{cases}
\]

**Choice:**

\[
< \text{CHOICE } \{S_1; S_2; \ldots; S_n\} \cdot S, \ \sigma > \rightarrow \\
< S_1 \cdot S, \ \sigma > \quad \text{or}^1 \quad < \text{CHOICE } \{S_2; \ldots; S_n\} \cdot S, \ \sigma >
\]

**Split:**

$^1$or represents don't know non-determinism.
$\langle SPLIT \; x (E_{f_1}, E_{f_2}) (E_{s_1}, E_{s_2}) | S, \sigma \rangle \rightarrow \begin{cases} \langle S, (true, \sigma ((\sigma(E_{f_1}), \sigma(E_{f_2}))/x)] > \\ or \\ \langle S, (true, \sigma ((\sigma(E_{s_1}), \sigma(E_{s_2}))/x)] > \\ if \; L(x) \neq U(x), \\ \langle S, (true, \sigma) \rangle, \\ otherwise. \end{cases}$

Passive Check:
$\langle s \cdot S, \sigma \rangle \rightarrow \langle S, (true, \sigma) \rangle$
\[ \begin{cases} s \; is \; x \leq y \; and \; U(x) \leq L(y) \\ if \\ s \; is \; x \neq y \; and \; (L(x), U(x)) \cap (L(y), U(y)) = \perp \\ ... \end{cases} \]

Narrow:
$\langle S \cup S', (true, \sigma') \rangle$
where $s$ is $op(x, y, z)$ and $S' = \bigcup_{i \in \{x, y, z\}} S_i$,
\[ \begin{cases} p_i \setminus p^{\text{passive}} \; if \; \sigma(i) \neq W^{op}(i, s) \\ \phi \; otherwise \end{cases} \]
and $\sigma' = \sigma[W^{op}(i, s)/i], i \in \{x, y, z\}$,
\[ \langle S,(false, \sigma) \rangle \]
if $\exists i, i \in \{x, y, z\}$, such that $W^{op}(i, s) = \perp$

Example: Consider the statement $SPLIT \; x (L(x), L(x)) (L(x) + 1, U(x))$. Let $\sigma(x) = (1, 10)$. The variable $x$ in the two interval vectors resulting from the $SPLIT$ statement will have the intervals $(1, 1)$ and $(9, 10)$ respectively. Suppose the $SPLIT$ statement is specified as
$SPLIT \ y \ (L(y), (L(y) + U(y))/2) ((L(y) + U(y))/2, U(x))$. Let $\sigma(y) = (1,10)$. In this case, the variable $y$ in the two interval vectors will have the intervals $(1,5.5)$ and $(5.5,10)$. If the variable $y$ is constrained to be an integer variable, then the above intervals would be narrowed to $(1,5)$ and $(6,10)$ respectively.

**Example:** Consider the integer constraints

$$
(1) \ x \neq y \quad (2) \ x \neq z \quad (3) \ y \neq z
$$

We have $P^x = \{1,2\}$, $P^y = \{1,3\}$, and $P^z = \{2,3\}$. Let $\sigma(x) = (1,2)$, $\sigma(y) = (3,4)$, and $\sigma(z) = (3,3)$. The details in narrowing the constraint set $\{1,2,3\}$ are shown below:

$$\langle \{1,2,3\}, \sigma \rangle \rightarrow \langle \{2,3\}, (true, \sigma) \rangle,$$

since the constraint (1) is passive. Hence, $p_{\text{passive}} = \{1\}$.

$$\langle \{2,3\}, \sigma \rangle \rightarrow \langle \{3\}, (true, \sigma) \rangle,$$

since the constraint (2) is passive. Hence, $p_{\text{passive}} = \{1,2\}$.

$$\langle \{3\}, \sigma \rangle \rightarrow \langle \{\} \cup \{3\}, (true, \sigma') \rangle,$$

$\sigma'(x) = (1,2)$, $\sigma'(y) = (4,4)$, and $\sigma'(z) = (3,3)$. While narrowing the constraint (3), the interval for the variable $y$ changed. Hence the set $P^y \setminus p_{\text{passive}}$, i.e., the set $\{1,3\} \setminus \{1,2\} = \{3\}$ needs to be added to the set of constraints yet to be processed.

$$\langle \{3\}, \sigma' \rangle \rightarrow \langle \{\}, (true, \sigma') \rangle.$$

Since the resulting constraint set is empty, the constraint solver terminates successfully with the interval vector $\sigma'$.  

127
5.2 Denotational Semantics

The syntactic items appearing in the language are

- variables $V$,
- constants $C$,
- intervals (lower and upper bounds) $I$,
- arithmetic expressions $\text{ArithExp}$,
- boolean expressions $\text{BoolExp}$,
- statements $\text{Stat}$.

Conventions

The conventions used for the syntax in explaining the semantics are:

- $v$ ranges over the variables $V$,
- $c$ ranges over the constants $C$,
- $i$ ranges over the intervals $I$,
- $a$ ranges over the arithmetic expressions $\text{ArithExp}$,
- $b$ ranges over the boolean expressions $\text{BoolExp}$,
- $s$ ranges over the statements $\text{Stat}$.

The formation rules for the arithmetic expressions $\text{ArithExp}$ are:

$$a ::= c \mid v \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \mid a_1 / a_2$$
Similarly, the rules are boolean expressions  \textbf{BoolExp} are:

\[
b ::= 0 \mid 1 \mid \neg b \mid b_1 \land b_2 \mid b_1 \lor b_2
\]

and those for the statements \textbf{Stat} are \footnote{\(a \lor b\) denotes either \(a\) or \(b\)}:

\[
s ::= \{a_1, b_1\} = \{a_2, b_2\} \mid \{a_1, b_1\} \leq \{a_2, b_2\} \mid \text{bind } v (c_1, c_2) \mid \text{split } v (a_1, a_2) \mid s_1; s_2 \mid s_1||s_2
\]

Expressions compute a value (i.e., an interval) in a particular state. Let \(\Sigma\) denote the set of states. The statements compute a new state from the old state. In the initial state \(\sigma_0, \sigma_0(v) = (-\infty, \infty),\) for all the variables \(v\). The configuration \(<s, \sigma>\) denotes that the statement \(s\) is to be executed in state \(\sigma\). The relation \(<s, \sigma> \rightarrow \sigma'\) specifies that the state \(\sigma'\) is the resulting state when the statement \(s\) is executed in state \(\sigma\).

An arithmetic expression \(a \in \textbf{ArithExp}\) denotes the function \(A[a] : \Sigma \rightarrow I\). A boolean expression \(b \in \textbf{BoolExp}\) denotes the function \(B[b] : \Sigma \rightarrow B\), where \(B\) is the set \(\{(0, 0), (1, 1), (0, 1)\}\). A statement \(s\) denotes the function \(S[s] : \Sigma \rightarrow (T \times \Sigma)\), where \(T\) is the set of truth values \(\{\text{true}, \text{false}\}\).

The semantic functions

\[
A : \textbf{ArithExp} \rightarrow (\Sigma \rightarrow I) \\
B : \textbf{BoolExp} \rightarrow (\Sigma \rightarrow B) \\
S : \textbf{Stat} \rightarrow (\Sigma \rightarrow (T \times \Sigma))
\]

are defined by structural induction. The denotation of an arithmetic expression, as a relation between states and intervals, is defined as follows:

\[
\begin{align*}
A[c] &= \{(\sigma, (e, c)) \mid \sigma \in \Sigma\} \\
A[v] &= \{(\sigma, \sigma(v)) \mid \sigma \in \Sigma\} \\
A[a_1 + a_2] &= \{(\sigma, (l_{a_1} + l_{a_2}, u_{a_1} + u_{a_2})) \mid \sigma \in \Sigma\}
\end{align*}
\]
\[
\sigma \in \Sigma \& (\sigma, (l_{a_1}, u_{a_1})) \in \mathcal{A} [a_1 \& (\sigma, (l_{a_2}, u_{a_2})) \in \mathcal{A} [a_2]]
\]

\[
\mathcal{A} [a_1 - a_2] = \{(\sigma, (l_{a_1} - u_{a_2}, u_{a_1} - l_{a_2})) \mid \sigma \in \Sigma \& (\sigma, (l_{a_1}, u_{a_1})) \in \mathcal{A} [a_1 \& (\sigma, (l_{a_2}, u_{a_2})) \in \mathcal{A} [a_2]]
\]

\[
\mathcal{A} [a_1 \cdot a_2] = \{(\sigma, (\min(l_{a_1} \cdot l_{a_2}, l_{a_1} \cdot u_{a_2}, u_{a_1} \cdot l_{a_2}, u_{a_1} \cdot u_{a_2}))) \mid \sigma \in \Sigma \& (\sigma, (l_{a_1}, u_{a_1})) \in \mathcal{A} [a_1 \& (\sigma, (l_{a_2}, u_{a_2})) \in \mathcal{A} [a_2]]
\]

\[
\mathcal{A} [a_1 / a_2] = \{(\sigma, (\min(l_{a_1} / l_{a_2}, l_{a_1} / u_{a_2}, u_{a_1} / l_{a_2}, u_{a_1} / u_{a_2}))) \mid \sigma \in \Sigma \& (\sigma, (l_{a_1}, u_{a_1})) \in \mathcal{A} [a_1 \& (\sigma, (l_{a_2}, u_{a_2})) \in \mathcal{A} [a_2]],
\]

if \(0 \notin (l_{a_2}, u_{a_2})
\]

The denotation of a boolean expression, as a relation between states and the set \(B = \{(0, 0), (1, 1), (0, 1)\}\), is defined below. The logical operations \(\neg_B, \land_B, \lor_B\) are defined in Table 5.1.

\[
B [0] = \{(\sigma, (0, 0)) \mid \sigma \in \Sigma\}
\]

\[
B [1] = \{(\sigma, (1, 1)) \mid \sigma \in \Sigma\}
\]

\[
B [\neg b] = \{(\sigma, \neg_b t) \mid \sigma \in \Sigma \& (\sigma, t) \in B [b]\}
\]

\[
B [b_1 \land b_2] = \{(\sigma, t_1 \land_B t_2) \mid \sigma \in \Sigma \& (\sigma, t_1) \in B [b_1] \& (\sigma, t_2) \in B [b_2]\}
\]

\[
B [b_1 \lor b_2] = \{(\sigma, t_1 \lor_B t_2) \mid \sigma \in \Sigma \& (\sigma, t_1) \in B [b_1] \& (\sigma, t_2) \in B [b_2]\}
\]

### 5.2.1 The Statements

First, let us consider the constraints (statements) of the form \(v_1 \ op \ v_2 = v_3\). The semantics for some of the operations is described below:

\[
S [v_1 + v_2 = v_3] = \{(\sigma, (t_1 \land t_2 \land t_3, \sigma[i_{v_1}/v_1, i_{v_2}/v_2, i_{v_3}/v_3])) \mid \sigma \in \Sigma \&
\]

\[
(t_1, i_{v_1}) = \mathcal{A} [v_1] \sigma \cap \mathcal{A} [v_3 - v_2] \sigma \&
\]

130
<table>
<thead>
<tr>
<th></th>
<th>( \neg A )</th>
<th>( \land A )</th>
<th>( \lor A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(1,1)</td>
<td>(0,0)</td>
<td>(0,0)</td>
</tr>
<tr>
<td>(0,0)</td>
<td>(1,1)</td>
<td>(0,0)</td>
<td>(1,1)</td>
</tr>
<tr>
<td>(0,0)</td>
<td>(0,1)</td>
<td>(0,0)</td>
<td>(0,1)</td>
</tr>
<tr>
<td>(1,1)</td>
<td>(0,0)</td>
<td>(0,0)</td>
<td>(1,1)</td>
</tr>
<tr>
<td>(1,1)</td>
<td>(1,1)</td>
<td>(1,1)</td>
<td>(1,1)</td>
</tr>
<tr>
<td>(1,1)</td>
<td>(0,1)</td>
<td>(0,1)</td>
<td>(1,1)</td>
</tr>
<tr>
<td>(0,1)</td>
<td>(0,0)</td>
<td>(0,0)</td>
<td>(0,1)</td>
</tr>
<tr>
<td>(0,1)</td>
<td>(1,1)</td>
<td>(0,1)</td>
<td>(1,1)</td>
</tr>
<tr>
<td>(0,1)</td>
<td>(0,1)</td>
<td>(0,1)</td>
<td>(0,1)</td>
</tr>
</tbody>
</table>

Table 5.1: The logical operations.
\[(t_2, i_{v_2}) = A [v_2] \sigma \cap A [v_3 - v_1] \sigma[i_{v_1/v_1}] \&
\]

\[(t_3, i_{v_3}) = A [v_3] \sigma \cap A [v_1 + v_2] \sigma[i_{v_1/v_1}, i_{v_2/v_2}] \]

\[S \ [(v_1 - v_2 = v_3)] = \{(\sigma, (t_1 \land t_2 \land t_3, \sigma[i_{v_1/v_1}, i_{v_2/v_2}, i_{v_3/v_3}])) \mid \sigma \in \Sigma \&
\]

\[(t_1, i_{v_1}) = A [v_1] \sigma \cap A [v_2 + v_3] \sigma \&
\]

\[(t_2, i_{v_2}) = A [v_2] \sigma \cap A [v_1 - v_3] \sigma[i_{v_1/v_1}] \&
\]

\[(t_3, i_{v_3}) = A [v_3] \sigma \cap A [v_1 - v_2] \sigma[i_{v_1/v_1}, i_{v_2/v_2}] \]

\[S \ [(v_1 \ast v_2 = v_3)] = \{(\sigma, (t_1 \land t_2 \land t_3, \sigma[i_{v_1/v_1}, i_{v_2/v_2}, i_{v_3/v_3}])) \mid \sigma \in \Sigma \&
\]

\[(t_1, i_{v_1}) = A [v_1] \sigma \cap A [v_3/v_2] \sigma \&
\]

\[(t_2, i_{v_2}) = A [v_2] \sigma \cap A [v_3/v_1] \sigma[i_{v_1/v_1}] \&
\]

\[(t_3, i_{v_3}) = A [v_3] \sigma \cap A [v_1 + v_2] \sigma[i_{v_1/v_1}, i_{v_2/v_2}] \]

Now consider the constraints of the form \(x \ op \ y\).

\[S \ [(v_1 = v_2)] = \{(\sigma, (t_1, \sigma[i_{v_1/v_1}, i_{v_2/v_2}])) \mid \sigma \in \Sigma \&
\]

\[(t_1, i_{v_1}) = A [v_1] \sigma \cap A [v_2] \sigma \]

\[S \ [(v_1 \leq v_2)] = \{(\sigma, (\text{true}, \sigma[i_{v_1/v_1}, i_{v_2/v_2}])) \mid \sigma \in \Sigma \&
\]

\[i_{v_1} = (c_{1_1}, \text{min}(c_{1_2}, c_{2_2})) \& i_{v_2} = (\text{max}(c_{1_1}, c_{2_1}), c_{2_2}) \&
\]

\[(c_{1_1}, c_{1_2}) = A [v_1] \sigma \& (c_{2_1}, c_{2_2}) = A [v_2] \sigma \&
\]

\[c_{1_1} \leq \text{min}(c_{1_2}, c_{2_2}) \& \text{max}(c_{1_1}, c_{2_1}) \leq c_{2_2} \} \cup
\]

\[\{(\sigma, (\text{false}, \sigma)) \mid \sigma \in \Sigma \&
\]

\[(c_{1_1}, c_{1_2}) = A [v_1] \sigma \& (c_{2_1}, c_{2_2}) = A [v_2] \sigma \&
\]

\[(c_{1_1} > \text{min}(c_{1_2}, c_{2_2}) \lor \text{max}(c_{1_1}, c_{2_1}) > c_{2_2} \}
\]

Consider the sequence of statements \(s_1; s_2\).

\[S \ [(s_1; s_2)] = \{(\sigma, \sigma'') \mid (\sigma, (\text{true}, \sigma')) \in S [s_1] \& (\sigma', \sigma'') \in S \ [s_2] \} \cup
\]

\[\{(\sigma, (\text{false}, \sigma')) \mid (\sigma, (\text{false}, \sigma')) \in S [s_1] \}
\]

The semantics of the \textit{bind} statement is
$S \text{ [bind } v \ (c_1, c_2) \text{]} =$

$\{(\sigma, (\text{true, } \sigma([A[c_1]]\sigma, A[c_2] \sigma\sigma)/v))\}$

The denotation of the split statement is explained below:

$S \text{ [split } v \ (a_1, a_1') \ (a_2, a_2') \text{]} =$

$\{\{(\sigma, (\text{true, } \sigma([A[a_1] \sigma, A[a_1'] \sigma\sigma]/v))\} \cup$

$\{(\sigma, (\text{true, } \sigma([A[a_2] \sigma, A[a_2'] \sigma\sigma]/v))\} \mid \sigma \in \Sigma \text{ & split\_possible } \}$

$\{(\sigma, (\text{true, } \sigma)) \& \text{split\_not\_possible } \}$

The semantics of the (parallel) execution of statements $s_1 \parallel s_2$ is as shown below:

$S \text{ [s_1\parallel s_2] = } \{\{(\sigma, \sigma') \mid (\sigma, \sigma') \in S [s_1]\}\cup$

$\{(\sigma_1, \sigma_1') \mid (\sigma_1, \sigma_1') \in S [s_2] \mid \sigma_1 = \text{copy } (\sigma)\}$

5.2.2 The Narrow Operation

The narrow operator takes a sequence of statements and an input interval vector and applies the appropriate narrow operations until the resulting interval vector converges, or else results in a failure. Thus the narrow operation can be modeled as a while statement:

$\eta = \text{while } \text{non\_convergence and narrow\_succeeds do narrow}_s.$

This is equivalent to

$\eta \sim \text{if } b \text{ then narrow}_s; \eta.$

So, we have

$\mathcal{N} [\eta] = \{(\sigma, \sigma') \mid B [b] \sigma = \text{true } \& \{(\sigma, \sigma') \in \mathcal{N} [\text{narrow}_s; \eta] \} \cup$

$\{(\sigma, \sigma) \mid B [b] \sigma = \text{false}.}$

Making the substitutions

$\zeta = \mathcal{N} [\eta], \beta = B [b], \xi = \mathcal{N} [\text{narrow}_s],$

we have
\[ \zeta = \{(\sigma, \sigma') \mid \beta(\sigma) = \text{true} \land (\sigma, \sigma') \in \zeta \cup \xi \} \cup \\
\{(\sigma, \sigma') \mid \beta(\sigma) = \text{false} \}. \]

This definition is recursive since it involves \( \zeta \) on the left and right sides of the equation.

Consider the function \( \Gamma \) which computes \( \Gamma(\zeta) \) given \( \zeta \). So, we need a fixed point \( \zeta \) of \( \Gamma \) such that

\[ \zeta = \Gamma(\zeta). \]
Chapter 6

Sequential Implementation

This chapter describes the sequential implementation of the proposed language. Essentially, a sequential interpreter traverses the syntax tree and processes the statements according to the operational semantics of the language. The data structures used in the sequential implementation are described in Section 6.1. The details of the interpreter are presented in Section 6.2.

6.1 The Interpreter

The source program written in the mini-language is input to the YACC compiler, which checks the syntax and the actions associated with the YACC specifications build a tree representation of the source program. The interpreter takes this tree representation and processes the constraints in the statements. (Figure 6.1.) The constraints are transformed into quadruples of the form \((\text{operator}, \text{operand}_1, \text{operand}_2, \text{operand}_3)\). For binary and unary operators, a dummy operand is used to fill in the quadruple. The constraint solver examines the quadruples and calls the appropriate narrowing operations. A suc-
Figure 6.1: The modules of the sequential interpreter.

<table>
<thead>
<tr>
<th>Intervals</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>i</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bounds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.2: The Intervals data structure.

cessful or a failure result is returned to the interpreter.

### 6.1.1 Data Structures for the Constraint Solver

The most important data structure of the constraint solver is the interval vector. Each interval vector is represented as a structure

```c
typedef struct intervals {
    short *type;
    double *bounds;
} Intervals;
```
Each variable, either appearing in the source program, or those internally generated as a temporary variable, is assigned an unique identifier, \( i \), in the range 0..\( MAX\_VAR \). The identifier is used as an index into the \texttt{bounds} and \texttt{type} fields of the interval vector. The lower and upper bounds of the \( i_{th} \) variable are stored in the locations \texttt{bounds}[2 * i] and \texttt{bounds}[2 * i + 1] in the structure \texttt{Intervals} (Figure 6.2.) The type information, i.e., whether the variable is of type \textit{integer}, \textit{real}, or \textit{interval}, is stored in the location \texttt{type}[i].

The variable \texttt{MAX\_VAR} is initialized to a predefined constant. Whenever, the generated variables exceed this limit, \texttt{MAX\_VAR} is increased by a predefined constant and the \texttt{bounds} and \texttt{type} fields are reallocated to accommodate this new size.

The constraints themselves are stored in an array

\[
\text{int} \quad \ast \text{constraints_array};
\]

Each constraint is decomposed into a set of primitive constraints of the form

\[
\text{operator( operand1, operand2, operand3 ).}
\]

These four components of the \( i_{th} \) constraint are stored in the fields \texttt{constraints_array}[4 * i + j], \( j = 0 \ldots 3 \), of the constraint array (Figure 6.3.)

Initially, the constraint array is allocated a space to store a predefined number of constraints (\texttt{MAX\_CONSTRAINTS}). The constraint array is dynamically reallocated whenever the generated constraints exceed the current maximum constraint array limit.

### 6.1.2 Data Structures for the Interpreter

The interpreter parses and interprets the statements in the user program. The statements in the program are stored as a tree structure with a \texttt{value} field, and three children which are pointers to the same tree structure.

\[
\text{typedef struct node_table *Node_ptr;}
\]
constraints_array

<table>
<thead>
<tr>
<th>0</th>
<th>...</th>
<th>i</th>
<th>...</th>
<th>...</th>
</tr>
</thead>
</table>

- constraints_array[4*i] (operator)
- constraints_array[4*i+1]
- constraints_array[4*i+2] (operands)
- constraints_array[4*i+3]

![Image](image.png)

Figure 6.3: The constraints_array data structure.

![Image](image.png)

Figure 6.4: The Node_ptr data structure.

typedef struct node_table {
    char    value[MAX_NAME];
    Node_ptr node1;
    Node_ptr node2;
    Node_ptr node3;
} node_table;

In Figure 6.4, a square represents the pointer to a node of the tree and the oval depicts the value field.

The procedure sub-programs are stored by the interpreter as follows. Each procedure has an entry in the linked list of procedure entries. Each such entry has a field to store the procedure name, a pointer to the procedure code, and a link to the next...
procedure (Figure 6.5.)

typedef struct proc_table *Proc_ptr;
typedef struct proc_table {
    char        proc_name[MAX_NAME];
    Node_ptr    proc_code;
    Proc_ptr    next;
} Proc_table;

The following structure holds the necessary information to process the choice points occurring in the source program. When a choice statement is encountered, a choice point record is pushed onto the choice point stack. The number of constraints in the constraints array, just before the encountered choice is processed, is stored in the constr_ptr field. This field is useful when restoring the choice point record. The num_var field holds the identifier of the last created variable before the choice point. The Intervals structure field, interval_vec, of the Choice structure holds the current interval bounds and types of the variables 0..num_var. The field cp_next points to the next alternate choice among the current choices, if there is any. To keep track of the
remaining program to be executed once the alternate choices are explored, the `rest` field
points to the continuation of the program (Figure 6.6.)

typedef struct choice {
    int       constr_ptr;
    int       num_var;
    Intervals interval_vec;
    Node_ptr  cp_next;
    Node_ptr  rest;
} Choice;

6.2 Details of the Interpreter

The translation of procedures including recursive ones is briefly described as follows.
A sequence of statements is handled using the syntax-directed translation shown in
Appendix A.2. Procedure calls are handled by generating the sequence of equality con-
straints binding actual parameters to new variables representing the new parameters of

140
a procedure call. This strategy makes it worthwhile to perform iterations using recursive calls instead of standard \textit{while}- statements which would also have to introduce new interval variables. The translator recognizes interval, integer and real variables by a declaration list which is provided in the program being translated. The reader should have no difficulty in extending the YACC code provided in Appendix A.2 to implement the translation of the declaration of variables and recursive procedures.

The variables declared as \texttt{integer} and \texttt{real} in a procedure declaration are handled by call-by-value. Though interval variables could be initialized using a procedure call, integer and real variables cannot be initialized in this way. This is illustrated in the following program:

```
main()
{
    var X: interval;
    var Y: integer;
    {
        init(X,Y);
        write X,Y
    }
}

proc init(x: interval; y: integer)
{
    { x = 1; y = 2 }
}
```

The procedure call \texttt{init} generates the constraints:

1) \( x = X \)
2) \( y = Y \)
3) \( x = 1 \)
4) \( y = 2 \)
The second constraint yields an error as the integer variable \( y \) is being assigned the value of the variable \( Y \) which is not yet assigned any particular value.

Call-by-reference of the real and integer variables is provided by declaring the respective formal parameters as ref in the procedure declaration. In the above case, the second constraint does not yield an error. Moreover, the constraint

\[
5) \ Y = y
\]

is generated at the end of the procedure call. Let us reconsider the above program but with the variable \( Y \) assigned a value.

```plaintext
main()
{
    var X: interval;
    var Y: integer;
    {
        Y = 1;
        init(X,Y);
        write X,Y
    }
}

proc init(x: interval; y: ref)
{
    { x = 1; y = 2 }
}
```

The constraints generated are:

1) \( Y = 1 \)

2) \( x = X \)

3) \( y = Y \), i.e., \( y = 1 \)

4) \( x = 1 \)

5) \( y = 2 \) (since \( y \) is an integer variable which can be reassigned a value)

6) \( Y = y \)
The output is $X = 1$; $Y = 2$. However, if the formal parameter $y$ in the procedure $\text{init}$ is declared by call-by-value (i.e. $y : \text{integer}$), constraint (6) is not generated, and the output in this case is $X = 1$; $Y = 1$. 
Chapter 7

Data Parallel SIMD

Implementation

This chapter describes data-parallel algorithms designed to process interval constraints using SIMD computers. The approach consists in specifying a set of linear or non-linear constraints involving variables which are defined by integer (or floating point) intervals. The constraints are processed simultaneously using the narrowing operation. Since a variable could appear in different constraints residing in different processors, the resulting intervals for a given variable can be different. In such a case, the intersection operator is used to find the appropriate common value, if it exists. If the intersection fails then there are no valid values for the variable in the given initial intervals.

A solution to the set of constraints is found when the intersected values of every variable are such that the lower bound equals the upper bound. If this result is unachievable after the narrowing and intersection, then one has to resort to the splitting operation.
The novel aspects in this chapter are performing the needed operations using data-parallel algorithms (see for example (BLEL 94)). The algorithms make use of typical data-parallel operations (like prefix-sum, prefix-min) that can be performed in logarithmic time in SIMD machines.

A set of processors deals with a set of constraints and the intervals of their variables as explained in Section 7.1. An important component of the algorithm is the marking of unused processors, namely those which were utilized in narrowing and intersections that fail, or those which have found output solutions. New sets of processors and intervals are created by the split operation (Section 7.1.2). This consists of doubling the previous set of processors by considering one of the variables for splitting, the remaining ones being copied. The variable considered for splitting is that containing the smallest interval; this increases the chances of earlier success or failure. The memory and processor requirements are doubled by each splitting and are decreased whenever narrowing fails and solutions are found.

The Markovian analysis of the complexity of the proposed algorithms is presented in Section 7.2. The speed-up results obtained using a Maspar MP2 (MASP 93) with 4,096 processors are shown in Section 7.3. Section 7.4 discusses some of the enhancements that can be made to process the data-parallel algorithms more efficiently.
Figure 7.1: Flowchart of the data-parallel solver.
7.1 Proposed Algorithm

Given a set of $m$ constraints involving $n$ variables, let

$$I = \{(l_1, u_1), (l_2, u_2), \ldots, (l_n, u_n)\},$$

be the set of couples denoting the lower and upper bounds of the $n$ variables,

$$V = \{V_1, V_2, \ldots, V_n\}.$$

Let $C$ be the set of constraints involving these variables,

$$C = \{C_1, C_2, \ldots, C_m\}.$$

Each $C_i$ is of the form $(op_3, V_j, V_k, V_l)$ (e.g., $(+, V_3, V_10, V_k)$), or $(op_2, V_j, V_k)$ (e.g., $(\neq, V_1, V_k)$), or $(op_1, V_j)$ (e.g., $(integer, V_2)$), $1 \leq j, k, l \leq n$. Algebraic constraints can always be placed into this form by using auxiliary variables.

A set of processors $P$ operates on $m$ constraints and $n$ variables. A flowchart of the basic operations of the data-parallel algorithm is shown in Figure 7.1. Each processor $p_i \in P$ contains only the constraint $C_i$ and the $(l_i, u_i)$ interval pair of the variable $V_i$. The size of $P$ is the maximum of $m$ and $n$. It is therefore possible that a processor contains just a constraint or just a variable. Note that $C_i$ may contain the variables $V_j, V_k$, or $V_l$ but not necessarily $V_i$. The interval pairs for $V_j, V_k$, or $V_l$ are fetched by the processor $p_i$ from the processors $p_j, p_k$, or $p_l$, respectively.

Narrowing is done in parallel by set(s) of processors $P$. Each $p_i \in P$ operates on its corresponding $C_i$ and computes the new interval pairs for the variables $V_j, V_k$, or $V_l$. Since different processors use different constraints, the narrowed $I_j$ (the interval pair of the variable $V_j$) computed by one processor is not necessarily equal to the narrowed $I_j$ computed by a different processor. (Recall that variable $V_j$ is stored in a unique processor.)
Let \( I'_j, I''_j, \ldots \) be the new narrowed interval pairs of \( V_j \). The intersection operation reduces, in parallel, all of these to a new pair \( I_j \), i.e.,

\[
I_j = I'_j \cap I''_j \cap \ldots
\]

which is stored back in place of the previous \( I_j \) in processor \( j \). Once the new set \( I \) is computed, and it differs from the previous \( I \), the narrowing operation is repeated using the new \( I \) in lieu of the previous one. The process of narrowing and intersection is repeated until \( I \) converges. (See loop \((ABC)^+\) in Figure 7.1.) Whenever the narrow operation on any of the \( C_i \)'s fails, or if the intersection of any of the \( I_j \)'s fails, the current interval pair set \( I \) is abandoned by the set of processors \( P \) and can be re-used to store other \( I \)'s. (Note that the size of all the processor sets remains the same.)

Note that there are two types of intersection operations involved. One is local to the constraint’s narrowing operation (see Figure 2.1). The other is an intersection resulting from several processors having different local values for the variables specified by the constraint it stores. We now open a parenthesis to describe this latter type of intersection which corresponds to box B in Figure 7.1.

### 7.1.1 Parallel Intersection

To achieve a logarithmic parallel intersection of several values of each of the variables, we resort to temporarily copying the values of the variables into additional processors. Let us assume that the variable \( V_i \) appears in the constraints \( C_{i_1}, C_{i_2}, \ldots, C_{i_{\phi^{(i)}}} \), where \( \phi^{(i)} \) is the number of times the variable \( V_i \) appears in the constraint set \( C \). To perform the intersection in parallel, each variable requires \( \phi^{(i)} \) processors. Hence the size of the processor set is \( \max(n, m, \sum_{i=1}^{n} \phi^{(i)}) \).

Let us define the \textit{start} and \textit{end} positions of the variable \( V_i \) (i.e., \( s^{(i)} \) and \( e^{(i)} \)) as
follows:

- $e^{(i)} = \sum_{j=1}^{i} \phi^{(j)}$
- $s^{(i)} = e^{(i)} - \phi^{(i)} + 1$

After the narrowing operation, the computed interval pairs of the $\phi^{(i)}$ occurrences of the variable $V_i$ are copied into the processors $s^{(i)}, \ldots, e^{(i)}$. In a manner similar to the parallel reduce operations, the intersection of $\phi^{(i)}$ pairs can be done in $L$ steps, where $L = \log(max_{i=1}^{n} \phi^{(i)})$. The intersected interval pair of the $i^{th}$ variable is computed in the $s^{(i)}$ processor. If the intersection operation on all the variables is successful, the interval pair in the $s^{(i)}$ processor is copied into the $i^{th}$ processor.

Consider the $i^{th}$ processor of any processor set and let $(op, V_j, V_k, V_l)$ be the constraint assigned to the $i^{th}$ processor. In a first step, the intervals for the variables $V_j$, $V_k$, and $V_l$ (i.e., $I_j$, $I_k$, and $I_l$) are fetched from the corresponding processors (i.e., $j$, $k$, and $l$) and stored in the $i^{th}$ processor.

The configuration of the $i^{th}$ processor is as shown in Figure 7.2a. The narrowing operation, if successful, yields the configuration in Figure 7.2b with new intervals for the variables $V_j$, $V_k$, and $V_l$ (say, $I'_j$, $I'_k$, and $I'_l$). Then the $i^{th}$ processor performs the intersection of all the new intervals for the variable $V_i$. The configuration of the $i^{th}$ processor in this case is shown in Figure 7.2c. We can now resume the detailed description of convergence test represented by box C in Figure 7.1.

### 7.1.2 Creating New Sets of Processors

Whenever convergence is achieved, the new computed $I$ may not be a solution ($I$ is a solution if $I_j = u_j, 1 \leq j \leq n$). If the new $I$ is a solution, the set of processors $P$ can be reused after the solution output. Otherwise, a new set of processors is created by
(a) Original memory layout.  (b) Layout after narrowing.

(c) Layout during parallel intersection.

Figure 7.2: Processor configurations during narrow and intersection.
the split operation. The constraint set $C$ and the corresponding interval sets are copied over to the new set of processors. The old set of processors contain a split interval for $V$ (the variable one wishes to split) and the new set of processors acquire the complement of the split interval. Now we have a set of set of processors $SP$ operating in parallel on different $I$’s ¹. The $SP$ grows until all the solutions are found. So, at any instant, let us assume that there are $r$ set of processors, i.e.,

$$SP = \{ P^{(1)}, P^{(2)}, \ldots, P^{(r)} \},$$

operating on $r$ different $I$’s, i.e.,

$$SI = \{ I^{(1)}, I^{(2)}, \ldots, I^{(r)} \}.$$ 

The $P^{(i)}$’s in parallel do the narrowing and perform intersections until the $I^{(i)}$’s remain unchanged. If none of the intersections have failed and if none of $I^{(i)}$’s is a solution, the split operation creates $r$ new set of processors, i.e.,

$$SP = \{ P^{(1)}, P^{(2)}, \ldots, P^{(r)}, P^{(r+1)}, \ldots, P^{(2r)} \},$$

and $r$ new $I$’s, i.e.,

$$SI = \{ I^{(1)}, I^{(2)}, \ldots, I^{(r)}, I^{(r+1)}, \ldots, I^{(2r)} \}.$$ 

(This corresponds to the path $(ABC)^+DF$ in Figure 7.1) However, if some of the narrowed intersections fail, or if some of them are solutions, the corresponding set of processors are free and can be re-used by the split operation. (These correspond to the paths $AE$, $ABE$ and $ABCD$ in Figure 7.1) Let $r'$ be the cardinality of the set of set of processors which have to participate in the split operation ($r' < r$). Let this set of set of processors be:

¹Different in the sense that one of the variables has been split.
\( C = \{x_1 \neq x_2, x_1 \neq x_3, x_2 \neq x_3\} \)

Figure 7.3: Example illustrating the operations in the data-parallel solver.
\[ SP' = \{ P^{(k_1)}, P^{(k_2)}, \ldots, P^{(k_r)} \} \]

which operate on the corresponding interval sets:

\[ I' = \{ I^{(k_1)}, I^{(k_2)}, \ldots, I^{(k_r)} \}. \]

To maximize the use of the set of processors, the \( I^{(k_i)} \)'s are mapped onto the processors set's \( P^{(i)} \)'s:

\[ I^{(i)} = I^{(k_i)}, \quad 1 \leq i \leq r'. \]

We call this phase *compaction*. Now the \( r' \) interval sets reside within the set of processors \( \{ P^{(1)}, P^{(2)}, \ldots, P^{(r')} \} \). In this case, the *split* operation creates \( r' \) new set of processors:

\[ SP = \{ P^{(1)}, P^{(2)}, \ldots, P^{(r')}, P^{(r'+1)}, \ldots, P^{(2r')} \}, \]

and \( r' \) new \( I' \)'s:

\[ SI = \{ I^{(1)}, I^{(2)}, \ldots, I^{(r')}, I^{(r'+1)}, \ldots, I^{(2r')} \}. \]

The *split* operation terminates when \( r' \) becomes 0. An example illustrating the above operations is shown in Figure 7.3. (In a later section, we describe how to avoid compaction.)

Figure 7.4 shows the narrow and intersection operations for the interval set \( \{(1, 1), (1, 3), (1, 3)\} \). (See Step 3 in Figure 7.3) In the first pass, the narrowed interval set is \( \{(1, 1), (2, 3), (2, 3)\} \). Since it differs from the previous set, the narrow and intersection operations are repeated till the set converges. This corresponds to the loop \((ABC)^+\) in Figure 7.1 which in this cases converges \((ABCABCD)\).

### 7.1.3 The Split Operation

The algorithm illustrating the data-parallel solver is shown in Figure 7.5. Consider the \( k^{th} \) iteration of the \textbf{while} loop (in Figure 7.5); let \( r_k \) be the number of \textit{active} interval
Figure 7.4: Example illustrating convergence.

sets,

\[ I_k^{(1)}, I_k^{(2)}, \ldots, I_k^{(r_k)} \]

(We call an interval pair set active if it participates in the narrow and convergence operation.) Initially we have \( I_1^{(1)} = I \). The result of applying the narrow operation in parallel until convergence to the \( r_k \) active interval sets yields the following output:

- \( \beta^i \), \( i \) in \( 1 \ldots r_k \), where \( \beta^i \) is a boolean variable representing the outcome of narrowing of the constraint set \( C \) using \( I_k^{(i)} \). If narrowing fails, \( \beta^i = 0 \), otherwise, \( \beta^i = 1 \).

- \( I_{k+1}^{(i)} \), \( i \) in \( 1 \ldots r_k \), where \( I_{k+1}^{(i)} \) is the new interval set resulting from the narrowing of \( C \) using \( I_k^{(i)} \).

- \( \gamma^i \), \( i \) in \( 1 \ldots r_k \), where \( \gamma^i \) is a boolean variable indicating whether \( I_{k+1}^{(i)} \) is a solution \( (\gamma^i = 1) \) or not \( (\gamma^i = 0) \). If \( \beta^i = 0 \), \( \gamma^i \) is irrelevant.

For space and processor efficiency reasons, the data-parallel algorithm has to compact the \( I_{k+1}^{(i)} \) towards the left-most processor sets by eliminating the \( I_{k+1}^{(i)} \)'s whose
\{The break statement returns control to the exit of the repeat-until loop.\}
\{ $r_k$ is the number of active interval vectors in the $k^{th}$ iteration\}

$k \leftarrow 1; r_k \leftarrow 1; \text{work\_completed} \leftarrow \text{false}$

\textbf{while not} $\text{work\_completed}$ (i.e., $r_k \neq 0$) \textbf{do}

$\text{compaction\_possible} \leftarrow \text{false}$

\textbf{for all} active\_interval\_vectors \textbf{in parallel do}

\textbf{repeat}

\hspace{0.5cm} $\text{narrow}$

\hspace{1cm} \textbf{if} narrow\_fails then $\text{compaction\_possible} \leftarrow \text{true}; \text{break\ end\ if}$

\hspace{1cm} $\text{intersection}$

\hspace{2cm} \textbf{if} intersection\_fails then $\text{compaction\_possible} \leftarrow \text{true}; \text{break\ end\ if}$

\hspace{0.5cm} until convergence \textbf{end\ repeat}$

\textbf{if} narrow\_succeeds \textbf{and} intersection\_succeeds then

\hspace{1cm} \textbf{if} solution then output\_solution; $\text{compaction\_possible} \leftarrow \text{true}$ \textbf{end\ if}$

\hspace{1cm} \textbf{if} $\text{compaction\_possible}$ then $r'_k \leftarrow \text{compact}(r_k)$ \textbf{end\ if}$

\hspace{1cm} \textbf{if} $r'_k = 0$ (i.e., \text{splitting is not possible})

\hspace{2cm} \textbf{then} $\text{work\_completed} \leftarrow \text{true}; r_{k+1} \leftarrow 0$ \textbf{else} $r_{k+1} \leftarrow \text{split\ and\ copy}(r'_k)$

\hspace{1cm} \textbf{end\ if}$

\hspace{0.5cm} \textbf{end\ for}$

\hspace{1cm} $k \leftarrow k+1$

\textbf{end\ while}$

\hspace{1cm}

Figure 7.5: Algorithm for the data-parallel solver.
{ Input: \( r_k \) \hspace{1cm} \text{Output: } r'_k \}

{ \( r'_k \) - number of active interval vectors that participate in the split operation}

\textbf{function} \hspace{0.1cm} \textit{compact}(r_k) \hspace{1cm} \textbf{begin}

\hspace{1cm} \textbf{for all} \hspace{0.1cm} i \hspace{0.1cm} \text{in} \hspace{0.1cm} 1 \ldots r_k \hspace{0.1cm} \text{in parallel do}

\hspace{1.2cm} \textbf{if} \hspace{0.1cm} \beta^i = 1 \hspace{0.1cm} \textbf{then}

\hspace{1.5cm} \delta \leftarrow \text{prefix-sum}(\beta) \hspace{0.5cm} \text{i.e., } \delta^i \leftarrow \sum_{j=1}^i \beta^j

\hspace{1.5cm} I'(\delta') = I(\delta)

\hspace{1.2cm} \textbf{end if}

\hspace{1cm} \textbf{end for}

\hspace{1cm} r'_k \leftarrow \text{reduce_add}(\beta) \hspace{0.5cm} \text{i.e., } r'_k \leftarrow \sum_{i=1}^{r_k} \beta^i

\hspace{1cm} \textbf{return} \hspace{0.1cm} r'_k

\hspace{1cm} \textbf{end} \hspace{0.1cm} \textit{compact}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{algorithm}
\caption{Algorithm for compaction.}
\end{figure}

\( \beta^i = 0 \). This is done by a simple parallel prefix sum operation which is available in most SIMD computers. Once the solutions (i.e., \( I_k^{(i)} \)'s such that \( \gamma^i = 1 \) and \( \beta^i = 1 \)) are processed, the respective processor sets are made available by making the corresponding \( \beta^i \)'s equal to 0. Compaction is possible only if at least one of the \( \beta^i \)'s equals 1. If all the \( \beta^i \)'s are 1, no compaction is possible and we could proceed with the split and copy phase. Let \( \beta \) denote the vector \((\beta^1, \beta^2, \ldots, \beta^{r_k})\). The algorithm for compacting the interval sets is shown in Figure 7.6. The compacting algorithm utilizes the following parallel operations:

Given the vector \( \varsigma = (\varsigma_1, \varsigma_2, \ldots, \varsigma_p) \),

156
• \textit{prefix-sum}(\zeta) returns the vector whose \(i^{th}\) component is:

\[
\sum_{j=1}^{i} \zeta_j
\]

• \textit{reduce-add}(\zeta) returns the value:

\[
\sum_{j=1}^{p} \zeta_j
\]

• \textit{prefix-min}(\zeta) returns the \(i\) such that

\[\zeta_i \neq 0\] \(^2\) is the minimum value among \(\zeta_1, \zeta_2, \ldots, \zeta_p\).

For example, given \(\zeta = (2, 0, 3, 0, 1, 0, 4)\),

\[
\text{prefix-sum}(\zeta) = (2, 2, 5, 5, 6, 6, 10),
\]

\[
\text{reduce-add}(\zeta) = 10, \text{ and}
\]

\[
\text{prefix-min}(\zeta) = 5 \text{ (the index of 1).}
\]

Superimposing prefix-sum with a conditional such as \(\zeta_i \neq 0\) yields:

\[
\text{prefix-sum}(\zeta) = (2, _, 5, _, 6, _, 10),
\]

where the underscores indicate the values are irrelevant.

Note that the function \textit{compact} returns the value \(r'_k\) to be the number of the active interval sets that were successful. The split and copy phase essentially doubles the active interval sets. Let \(v(i), i \in 1 \ldots r'_k\), be the variable to split for the \(r'_k\) interval sets. One strategy is to choose the \(v(i)\) such that the difference between \(u_{u(i)}\) and \(l_{u(i)}\) is minimum. This is determined by the parallel \textit{prefix-min} operation. Let \(v\) denote the vector \((v(1), v(2), \ldots, v(r'_k))\). The algorithm \textit{split_and_copy} is shown in Figure 7.7.

That algorithm returns the new value of \(r\) for the \(k + 1\)st iteration of the while loop.

\(^2\)A value of zero corresponds to \(i = u\) and no further splitting is possible.
\{ \text{Input: } r'_k \quad \text{Output: } 2 \ast r'_k \} \\

\textbf{function} \text{split\_and\_copy}(r'_k) \\
\text{begin} \\
\text{for all } i \text{ in } 1 \ldots r'_k \text{ in parallel do} \\
\quad \rho^i \leftarrow I^i(u_j - l_j), \ j \in 1 \ldots n \\
\quad v^i \leftarrow \text{prefix-min}(\rho^i) \quad \text{i.e., } v^i \leftarrow j | \rho^i_j \text{ is minimum and is } \neq 0, \ j \in 1 \ldots n \\
\quad Temp^i \leftarrow I^i \\
\quad I^i \leftarrow \text{split\_strategy\_first}(v^i, I^i) \\
\quad I'^{r'_k+i} \leftarrow \text{split\_strategy\_second}(v^i, Temp^i) \\
\text{end for} \\
\text{return } 2 \ast r'_k \\
\text{end } \text{split\_and\_copy} \\

\text{Figure 7.7: Algorithm for split and copy.}

in Figure 7.5. Figure 7.8 shows the intermediate steps in the compact, split, and copy phases.

\subsection{Avoiding Compaction}

The compaction phase can be avoided by keeping track of the active processor sets (Figure 7.9). The revised \text{split\_and\_copy} procedure is shown in Figure 7.10.
7.2 Markovian Analysis

Given a transition probability matrix $T$ of the nodes in the flowgraph, and the vector $S$ denoting the time-variables of the nodes in the flowgraph (TRIV 82; CoWe 92), the time-formula for the average execution time is given by:

$$(\text{first\_row}((I - T)^{-1})) \cdot S$$

where $I$ is the identity matrix.

A Maple\(^3\) program computing the time-formula is presented in Figure 7.11. The probabilities considered are as follows:

- $p_n$, the probability that narrow succeeds,
- $p_i$, the probability that intersection succeeds,

\(^3\)A symbolic computing system developed in the Department of Computer Science at the University of Waterloo, Waterloo, Ontario, Canada.
{ Input: $r_k$  
Output : $r_k'$ } 

{ $r_k'$ - number of active interval vectors that participate in the split operation } 

function active-sets($r_k$) 

begin 

for all $i$ in $1 \ldots r_k$ in parallel do 

if $\beta^i = 1$ then 

$\delta \leftarrow \text{prefix-sum}(\beta)$  
i.e., $\delta^i \leftarrow \sum_{j=1}^{i} \beta^j$ 

$\text{pos}^{\delta^i} \leftarrow i$ 

end if 

end for 

$r_k' \leftarrow \text{reduce-add}(\beta)$  
i.e., $r_k' \leftarrow \sum_{j=1}^{r_k} \beta^j$ 

return $r_k'$ 

end active-sets 

Figure 7.9: Algorithm for active sets.
\begin{algorithm}
\small
\{ Input: \( r_k, r'_k \) \hspace{1cm} \text{Output: } r \}\}
\begin{algorithmic}
\Function{split_and_copy}{\( r_k, r'_k \)}
\State\begin{algorithmic}
\ForAll{\( i \) in \( 1 \ldots r_k \) in parallel do}
\If{\( \beta^i = 1 \)}
\State \( \rho^i \leftarrow I^{(i)}(u_j - l_j), j \in 1 \ldots n \)
\State \( v^i \leftarrow \prefix\text{-}\min(\rho^j) \) \hspace{0.5cm} \text{i.e., } v^i \leftarrow j|\rho^j_j \text{ is minimum and is } \neq 0, j \in 1 \ldots n \)
\State \( \Temp^{(i)} \leftarrow I^{(i)} \)
\State \( I^{(i)} \leftarrow \text{split\_strategy\_first}(v^i, I^{(i)}) \)
\State \( \Temp^{(i)} \leftarrow \text{split\_strategy\_second}(v^i, \Temp^{(i)}) \)
\EndIf
\EndFor
\State \( r \leftarrow \max(2 \ast r'_k, r_k) \)
\ForAll{\( i \) in \( 1 \ldots r \) in parallel do}
\If{\( \beta^i = 0 \)}
\State \( \kappa^i \leftarrow 1 \)
\State \( \eta \leftarrow \prefix\text{-}\sum(\kappa) \)
\If{\( \eta^i \leq r'_k \)}
\State \( I^{(i)} \leftarrow \Temp^{(pos^i)} \)
\State \( \beta^i \leftarrow 1 \)
\EndIf
\EndIf
\EndFor
\State\Return{} \( r \)
\EndFunction
\end{algorithmic}
\end{algorithm}

\caption{Revised algorithm for split and copy.}
\end{algorithm}

Figure 7.10: Revised algorithm for split and copy.
• $p_c$, the probability that *convergence* succeeds,

• $p_r$, the probability that *solutions* (results) are found,

• $p_s$, the probability that *split* is possible.

The computed time formula is:

$$t := \frac{\text{narrow}}{1-p_b P_1+P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9} + \frac{\text{intersection}}{1-p_b P_1+P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9} +$$

$$\frac{\text{convergence}}{1-p_b P_1+P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9} + \frac{\text{solution}}{1-p_b P_1+P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9} +$$

$$\frac{\text{compact}}{1-p_b P_1+P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9} + \frac{\text{split}}{1-p_s} + \frac{\text{copy}}{1-p_s}$$

For example, with the probability substitutions

$$p_b = 1, p_i = \frac{2}{3}, p_c = \frac{1}{2}, p_r = 0, \text{ and } p_s = \frac{1}{2},$$

corresponding to the path *ABCABCDFGABEFH* in Figure 7.1,

$$t := 3 \text{narrow} + 3 \text{intersection} + 2 \text{convergence} + \text{solution} + \text{compact} + 2 \text{split} + \text{copy}$$

We will now constrain the probabilities so that Kirchhoff’s laws are applicable. Let $C_j$ be the number of times branch $j$ in the flowchart of Figure 7.1 is executed in solving a problem. We differentiate between a success branch and a failure branch by using the counters $C_{js}$ and $C_{jf}$. For example, $C_{ns}$ and $C_{nf}$ are the number of times *narrow* succeeds and fails respectively. Then $p_b = \frac{C_{nf}}{C_{ns} + C_{nf}}$. Note that $C_{sf} = 1$. The following relations hold (with reference to Figure 7.1):

Node A :  
$$1 + C_{ss} + C_{sf} = C_{ns} + C_{nf},$$

Node B :  
$$C_{ns} = C_{is} + C_{if},$$

Node C :  
$$C_{is} = C_{es} + C_{ef},$$

162
% identify nodes
A := 1; B := 2; C := 3; D := 4; E := 5; F := 6; G := 7; H := 8;

% input of a sparse markovian matrix
T := array(1..8,1..8,
            [(A,B) = p_a, (A,E) = 1 - p_a, (B,C) = p_1, (B,E) = 1 - p_1,
            (C,D) = p_c, (C,A) = 1 - p_c, (D,E) = p_r, (D,F) = 1 - p_r,
            (E,F) = 1, (F,G) = p_s, (F,H) = 1 - p_s, (G,A) = 1],
            sparse);

% vector specifying identifiers for time variables
S := array(1..8, [narrow,intersection,convergence,solution,compact,split,
                  copy, 0] );

% the identity matrix
I := array(1..8,1..8, identity);

% evalm --- evaluate the matrix
m := evalm(I - T);

% linalg --- linear algebra package; inverse --- inverse of matrix
r := linalg[inverse](m);

% subvector --- appropriate subvector of matrix
v := linalg[subvector](r,1,1..8);

% dotprod --- dot product of two vectors
time := linalg[dotprod](v,S);

% Figure 7.11: Maple program to compute the time formula.
Node D: \[ C_{cs} = C_{rs} + C_{rf}, \]

Node F: \[ C_{nF} + C_{if} + C_{rs} + C_{rf} = C_{ss} + C_{sf}. \]

Let us study how the probabilities interact for some simpler cases:

- \( C_{ss} = 0 \) i.e., only one iteration.

  Assuming \( p_n = 1, \ p_i = 1, \ p_r = 1 \), we have \( p_c = \frac{1}{C_{ns}}, \ C_{ns} \geq 1 \).

- Assuming \( p_n = 1 \) and \( p_i = 1 \),

  \[ p_c = \frac{(1+C_{ss})}{C_{ns}}, \ C_{ns} \geq 1 + C_{ss}, \]
  \[ p_r = \frac{C_{rs}}{(1+C_{ss})}, \ 0 \leq C_{rs} \leq 1 + C_{ss} \]

Figure 7.12a shows how \( p_c \) varies with \( C_{ss} \) and \( C_{ns} \). Figure 7.12b shows how \( p_r \) varies with \( C_{ss} \) and \( C_{rs} \).

### 7.2.1 Analysis of Processor Sets

The number of processor sets in the \((k+1)\text{th}\) iteration of the algorithm in Figure 7.5 is given by the following finite-difference equation:
\[ r_{k+1} = 2 \cdot (r_k - Q(k) \cdot r_k) = 2 \cdot (1 - Q(k)) \cdot r_k \]
\[ r_1 = 1 \]

where \( Q(k) = Q_{\text{fails}}(k) + Q_{\text{solves}}(k) \), \( Q_{\text{fails}} \) being the fraction of processor sets whose interval sets fail, and \( Q_{\text{solves}} \) being the fraction of processor sets whose interval sets are solutions. These are precisely the processor sets reclaimed during compaction. The split and copy phase doubles the processor sets whose inputs are not yet solutions.

Solving the above finite difference equation, we have:

\[ r_k = 2^{k-1} \cdot (1 - Q(k))^{k-1}, \quad \text{where} \quad 0 \leq Q(k) \leq 1 \]

The maximum number of processor sets \( R \) that can operate in parallel is bound by the physical number of processors available. If the underlying machine has \( N \) processors \(^4\), then

\[ R = N \div \max(m, n) \]

where the problem has \( m \) constraints and \( n \) variables. As long as \( r_k \leq R \), all the \( r_k \) processor sets operate in parallel. However, if \( r_k > R \), the number of passes \(^5\) required \((NP)\) to process the \( r_k \) processors sets is given by:

\[
NP = \begin{cases} 
    r_k \div R & \text{if } r_k \mod R = 0 \\
    1 + (r_k \div R) & \text{otherwise}
\end{cases}
\]

In this case, each processor set holds more than one interval set. The \( i^{th} \) interval set is stored within the processor set \( i \mod R, 1 \leq i \leq r_k \). So each processor set has \( NP \) or \( NP - 1 \) interval sets.

\(^4\)In our case \( R = 4096 \)

\(^5\)A pass consists of processing 4096 constraints (or variable intervals)
Assume $Q$ to have a uniform distribution between 0 and 1. The number of processor sets $r_k$ in the $k^{th}$ iteration is shown in Figure 7.13a. The number of passes required to process the $r_k$ processor sets is shown in Figure 7.13b. For a fixed $R$, the number of passes required as the number of constraints increases is shown in Figure 7.13c. For a fixed problem size, the number of passes required as $R$ increases is shown in Figure 7.13d.
Figure 7.13: (a) Number of Processor Sets. (b) Number of Passes.
(c) Number of Passes vs. Number of Constraints.
(d) Number of Passes vs. Number of Processors.
<table>
<thead>
<tr>
<th>Problem</th>
<th>m</th>
<th>n</th>
<th># Solutions</th>
<th>Sequential Time</th>
<th>Parallel Time</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-Queens</td>
<td>140</td>
<td>72</td>
<td>92</td>
<td>6.1s</td>
<td>1.4s</td>
<td>4.4</td>
</tr>
<tr>
<td>9-Queens</td>
<td>180</td>
<td>90</td>
<td>352</td>
<td>29.2s</td>
<td>2.7s</td>
<td>10.8</td>
</tr>
<tr>
<td>Perm-6</td>
<td>15</td>
<td>6</td>
<td>720</td>
<td>1.0s</td>
<td>0.25s</td>
<td>4.0</td>
</tr>
<tr>
<td>Perm-7</td>
<td>21</td>
<td>7</td>
<td>5040</td>
<td>7.0s</td>
<td>0.43s</td>
<td>16.2</td>
</tr>
<tr>
<td>6x6 Job-Shop</td>
<td>793</td>
<td>617</td>
<td>1</td>
<td>0.85s</td>
<td>0.47s</td>
<td>1.8</td>
</tr>
</tbody>
</table>

Figure 7.14: Speed-up results.

7.3 Results

The table in Figure 7.14 compares the sequential and parallel execution times for various problems. The sequential time is obtained using the sequential implementation of the algorithm on the SGI Onyx (with one processor). The parallel time corresponds to the data-parallel implementation on the MASPAR MP2 with 4096 processors.
7.4 Enhancements

The following subsections describe how the data-parallel algorithms can be further improved.

7.4.1 Connected Components

Suppose the constraint set \( C = \{C_1, C_2, \ldots, C_m\} \) is partitioned into \( t \) disjoint sets \( \{S_1, S_2, \ldots, S_t\} \) such that for any constraints \( C_i \in S_r, C_j \in S_s, r \neq s, \var(C_i) \cap \var(C_j) = \emptyset \). The constraint sets \( S_1, S_2, \ldots, S_t \) are assigned to the processors \( p_1, p_2, \ldots, p_t \). The \( t \) processors constitute a processor set. Also, all the variables appearing in a constraint set are assigned to the respective processor. Each processor does the narrowing operations on its constraint set. No communication cost is involved in the narrowing phase as all the constraint sets are independent of each other.

\[
\text{for all } i \text{ in } 1 \ldots t \text{ in parallel do}
\]
\[
\text{repeat } \text{narrow}(S_i) \text{ until convergence}
\]

The above method is particularly useful if all the constraint sets are approximately the same size. Otherwise, the time for narrowing the constraints is dominated by the processor holding the maximum number of constraints.

7.4.2 Equal Partitioning of Constraints

In this model, the constraints and the variables are distributed uniformly over the required number of processors. These processors constitute a processor set. If the size of the processor set is less than the number of constraints, more than one constraint resides in each processor. In the extreme case where the size of the processor set is equal to the
number of constraints, only one constraint is assigned to each processor. On the other 
hand, if the size of the processor set is equal to one, all the constraints are assigned to 
a single processor. This is the case when the all the constraints form a single connected 
component.

Let us assume that the size of the processor set is \( t \). The \( m \) constraints are 
distributed equally among the \( t \) processors. The interval variables, \( V = \{ V_1, V_2, \ldots, V_n \} \), 
are dispersed equally among the \( t \) processors. Although each processor has (roughly) the 
same number of constraints, the number of distinct variables appearing in the constraints 
in each of the processors may not be the same. Each processor needs to keep a record 
of the variables appearing in its constraints, as well as the information whether the 
interval values of the variables are already fetched from the remote processors in which 
the variables reside.

**Notation:**

- \( C_i^j \) — the constraint \( C_i \) resides in processor \( p_j \).

- \( V_i^j \) — the variable \( V_i \) resides in processor \( p_j \).

- \( \text{fetch } c_{b_k}(V_i^j) \) — a Boolean function returning whether the interval for the variable 
  \( V_i \) residing in processor \( p_j \) is already fetched by the processor \( p_k \), or not.

- \( I^{h}_{V_i^j} \) — the interval for the variable \( V_i \) residing in processor \( p_j \) fetched by the 
  processor \( p_k \).

**Narrowing**

Narrowing is done in parallel by all the \( t \) processors. For each of the constraints, the 
algorithm checks whether the variables in the constraint reside in the same processor or 
in a different processor. If the variable resides in a different processor, and if the interval
repeat

for all \( k \) in \( 1 \ldots t \) in parallel do

for each constraint \( C_i^k \) do

for each \( V_j^r \in \text{var}(C_i^k) \) do

if not \( \text{fetch}_{ck}(V_j^r) \) then

\( L_{V_j^r}^k \leftarrow \text{copy}(V_j^r) \); \( \text{fetch}_{ck}(V_j^r) \leftarrow \text{true} \)

end for

\( \text{narrow}(C_i^k) \)

end for

for each variable \( V_j^i \) do

\( \text{intersection}(L_{V_j^i}^k) \)

if \( \text{changed} \) then \( \text{fetch}_{ck}(V_j^i) \leftarrow \text{false} \)

end for

end for

until convergence

Figure 7.15: Improved algorithm for narrowing.

value has not been fetched already, the processor fetches the interval for the variable into its local memory. Since each variable could appear in the constraints residing in different processors, the intersection operation is carried out in parallel. The narrowing and intersection operations are repeated until the interval set converges, or the operations fail (See Figure 7.15.)
Chapter 8

Shared Memory MIMD Implementation

This chapter outlines the parallel implementation of the constraint language on shared-memory multiprocessors. OR-parallelism is exploited through the choice and split statements. The narrowing of individual constraints making up a program is processed sequentially. Section 8.1 describes the data structures used by parallel algorithms. A solve interpreter for parallel processing of a program is specified in Section 8.2. Section 8.3 sketches the analysis of the parallel program using strings generated by a context-free grammar. An analytical study of the speed-up is shown in Section 8.4. Section 8.5 provides the implementation details of the solve interpreter. A detailed example on narrowing is shown in Section 8.6. Finally, the speed-ups achieved for various problems using a shared memory MIMD computer are reported in Section 8.7.
BIND y (1, 3);

CHOICE {
    {x = 1};
    {x = 2};
    {x = 3}
};

SPLIT y (l(y), l(y)) (l(y) + 1, u(y));

x <> y;

write x, y

Figure 8.1: An example program.

8.1 Data Structures

To describe the algorithms presented in this chapter, programs are represented by lists of statements. The statements can either be a choice, split, or a constraint. The choice
and the split statements have the following structure:

- the CHOICE statement, with access to its head and tail components, which repre-
sent the first and the rest of the alternatives,

- SPLIT, with access to its first and second splitting components, which specify how
  a variable will be split.

Pictorially, lists are depicted as trees. Figure 8.1 shows an example program with
split and choice constructs. The list structure for the program is shown in Figure 8.2.
8.2 The Parallel Procedure Solve

The parallel procedure \textit{solve} which traverses the syntax-tree representing a source program and interprets the \textit{choice}, \textit{split}, and \textit{narrow} operations is shown in Figure 8.3. The parameters of the \textit{solve} interpreter are:

\begin{itemize}
  \item \textbf{p :} The input \textit{program}, comprising the sequence of statements, with access to its \textit{head} and \textit{tail},
  \item \textbf{cl :} The accumulated \textit{constraint-list}, as the \textit{solve} process traverses through the \textit{program}, with access to the list’s \textit{head} and \textit{tail}, and
  \item \textbf{iv :} The \textit{interval-vector} of the variables appearing in the \textit{constraint-list}.
\end{itemize}

The \textit{solve} procedure assumes an unbounded number of available processors. The execution-tree corresponding to the traversal of the program in Figure 8.1 is shown in Figure 8.4. The main call is \textit{solve}(p, NIL, iv), where \textit{p} points to the root of the syntax-tree, and \textit{iv} is the interval-vector with all the variables initialized to (-\infty, \infty). The second parameter \textit{cl} (initialized to NIL) contains the list of accumulated constraints.
Once a leaf node of the execution-tree is reached, the resulting interval vector might not be a solution. It is worthwhile recalling that in CLP(Interval) a final resulting interval vector has the following interpretation: “If there exist one or more solutions, these lie within the bounds of the resulting interval vector.” The procedure \textit{enumerate} is used to actually determine if these solutions exist. The auxiliary boolean procedure \textit{last\_unexplored\_split\_var} finds the variable with different lower and upper bounds as specified by the last encountered SPLIT constraint.
{Parallel procedure \textit{solve} assuming an unbounded number of processors}

\textbf{procedure} solve\((p, \, cl, \, iv)\)
\begin{verbatim}
begin
  if \(p \neq \text{NIL}\) then
    switch head\((p)\)
    case \text{CHOICE}:
      parallel begin
        solve\(\text{cons(head(head(p)), \, tail(p)), \, cl, \, iv}\)
        solve\(\text{cons(tail(head(p)), \, tail(p)), \, cl, \, iv}\)
      parallel end
    case \text{SPLIT}:
      split\_intervals\(\text{head(p), \, iv, \, iv_1, \, iv_2}\)
      parallel begin
        solve\(\text{tail(p), \, cons(head(p), \, cl), \, iv_1}\)
        solve\(\text{tail(p), \, cons(head(p), \, cl), \, iv_2}\)
      parallel end
    otherwise: /* a constraint */
      if narrow\(\text{cons(head(p), \, cl, \, iv, \, iv')}\) then
        solve\(\text{tail(p), \, cons(head(p), \, cl, \, iv')}\)
      end switch
  else begin
    if \text{enumeration\_desired} then enumerate\(\text{cl, \, iv}\)
    else print\(\text{iv}\)
  end
end solve
\end{verbatim}

Figure 8.3: The Procedure \textit{solve}.\[176\]
Figure 8.4: Execution tree for the program in Figure 8.1.
procedure enumerate(cl, iv)
begin
  flag ← last_unexplored_split_var(cl, iv, var)
  if flag then
    split_intervals(var, iv, iv_1, iv_2)
    parallel begin
      if narrow(cl, iv_1, iv_1') then enumerate(cl, iv_1')
      if narrow(cl, iv_2, iv_2') then enumerate(cl, iv_2')
    parallel end
  end if
end enumerate

Figure 8.5: The procedure enumerate.
The statement

\[
\text{parallel begin} \\
S_1 \\
S_2 \\
\text{parallel end}
\]

specifies that the statements \( S_1 \) and \( S_2 \) are to be executed in parallel using copies of their current parameters. The procedure \textit{split-intervals} splits the given interval vector into two components as specified by the \textit{split} statement. The \textit{narrow} procedure determines the new interval vector such that the narrow operation on all constraints converges.

An interval vector is a solution if the lower and upper bounds are equal for all the variables which are constrained to be integers, or the bounds are within a desired precision for the variables not constrained as integers. The procedure \textit{enumerate}, shown in Figure 8.5 is useful in enumerating the solutions. The \textit{split} statement is also used as a directive in the enumeration phase.

### 8.3 Analysis

The calling sequence of the recursive procedure \textit{solve}, shown in Figure 8.3, can be expressed by a context-free grammar. Let the sequential execution of two processes \( P_1 \) and \( P_2 \) be denoted by \((P_1, P_2)\) and the parallel execution of \( P_1 \) and \( P_2 \) be represented by \( P_1 \parallel P_2 \). The context-free grammar capturing the \textit{solve} procedure is as follows:

\[
solve \rightarrow (\text{CHOICE } \text{solve}) \mid (\text{SPLIT solve}) \mid (\text{narrow solve}) \mid \text{enumerate} \mid \epsilon
\]

179
Figure 8.6: Parse tree for the execution tree in Figure 8.4.

\[ \text{enumerate} \rightarrow (\text{SPLIT}^{(\text{narrow \ enumerate})}) \mid \epsilon \]

To each program execution there is a corresponding syntax-tree which is obtained from the execution tree by disregarding the values of the variables. We can now consider an abstract model of the execution tree as the syntax-tree of the given context-free grammar. This abstract representation is convenient for attempting to provide an analytical study of speed-ups. A parse of the example shown in Figure 8.4 is outlined in Figure 8.6.

An estimate of the average execution time can be obtained by generating all strings of length \( n \) using the context-free grammar \( G \). Let \( m \) be the number of such strings (usually a very large number compared to \( n \)). Let \( E_i \) denote the execution time for string \( i \). The average execution time is \( \sum_{i=1}^{m} E_i / m \). When \( n \) is small, all the strings of length \( n \) can be generated by using DCG’s. However, for large \( n \), there are techniques that do not require the generation of all the strings. These techniques developed by (HiCo 88; FLAJ 87) just generate the so called uniformly random strings.
Such a uniform random generation can be thought of as one in which each proposed string is chosen according to an index of a random number between 1 and \( m \).

### 8.4 Processor Scheduling

For an analytical study of the number of processors required, the nodes of the execution-tree are assigned processor labels. Assuming an unbounded number of processors, a labeling of the execution-tree in Figure 8.4 is shown in Figure 8.7. When a SPLIT or a CHOICE is encountered, the node of the left branch is assigned the same label as its parent. The node of the right branch is labeled with the next available processor label. Let us assume that the processor labels are stored in a stack. When a leaf node is encountered, the processor label is free and pushed onto the stack. In the case shown in Figure 8.7, six processors are needed to complete the execution in five time steps.

In the case of a finite number of processors, some portions of the execution-tree are suspended until the processors become available. The table in Figure 8.8 shows the number of time steps needed with varying the number of processors for the execution tree in Figure 8.4. The context-free grammar is used in generating random strings of arbitrary length (ZIMM 92). The speed-up achieved with varying the number of processors can be analytically calculated (SHUB 95).

The Prolog program shown in Figure 8.9 labels the given execution tree. Constant (unit) time assumptions are made for choice, split, narrow, and enumerate. The execution tree is input as a list of triplets of the form \((Node, Level, Id)\), where \(Node\) is the node label, \(Level\) is the level of the node in the execution tree, and \(Id\) is the identifier for the node as mapped to a complete binary tree. A sample execution tree along with its input representation is shown in Figure 8.10.
Figure 8.7: Labeling the execution tree.

<table>
<thead>
<tr>
<th>#Processors</th>
<th>Time Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 8.8: Time Steps for the Execution Tree in Figure 8.4.
color([Node, Level, Id] | Rest], Labeled, Susp, Step, Colors, Temp_Colors, NStep, New_Colors, New_Temp_Colors, Final):-
    process([Node, Level, Id], Labeled, Step, Colors, Temp_Colors, Proc, Time, Aux_Step, Aux_Colors, Aux_Temp_Colors), !,
    append(Susp, Rest, NRest),
    color(NRest, [(Node, Level, Id, Proc, Time) | Labeled], [], Aux_Step,

color([Node, Level, Id] | Rest], Labeled, Susp, Step, Colors, Temp_Colors, NStep, New_Colors, New_Temp_Colors, Final):-
    color(Rest, Labeled, [(Node, Level, Id) | Susp], Step, Colors, Temp_Colors, NStep, New_Colors, New_Temp_Colors, Final).

color([], Labeled, [], Step, Colors, Temp_Colors, Step|Colors, Temp_Colors, Labeled) :- !.

color([], Labeled, Susp, Step, Colors, Temp_Colors, NStep, New_Colors, New_Temp_Colors, Final):-
    append(Colors, Temp_Colors, Aux_Colors),
    Aux_Step is Step + 1,
    color(Susp, Labeled, [], Aux_Step, Aux_Colors, [], NStep, New_Colors, New_Temp_Colors, Final).

process([Node, Level, Id], Labeled, Step, Colors, Temp_Colors, Proc, Time, NStep, New_Colors, New_Temp_Colors):-
    even(Id), parent(Id, PId), !, PLevel is Level - 1,
    member([_Node, PLevel, PId, Proc, PTime], Labeled),
    determine_time(PTime, Step, Colors, Temp_Colors, New_Colors, Aux_Colors, Time, NStep),
    leaf([Node, Level, Id], Proc, Aux_Colors, New_Temp_Colors).

process([Node, Level, Id], Labeled, Step, [Proc|Colors], Temp_Colors, Proc, Time, NStep, New_Colors, New_Temp_Colors):-
    parent(Id, PId), !, PLevel is Level - 1,
    member([_Node, PLevel, PId, _Proc, PTime], Labeled),!,
    determine_time(PTime, Step, Colors, Temp_Colors, New_Colors, Aux_Colors, Time, NStep),
    leaf([Node, Level, Id], Aux_Colors, New_Temp_Colors).

determine_time(Time, Time, Colors, Temp_Colors, New_Colors, [], NTime, NTime) :- !,
    NTime is Time+1,
    append(Colors, Temp_Colors, New_Colors).

determine_time(_, Time, Step, Colors, Temp_Colors, Colors, Temp_Colors, Step, Step).

leaf([_Node, Level, Id], _Proc, Temp_Colors, Temp_Colors) :-
    CLevel is Level + 1, CID is 2*Id, tree(TreeList),
    member([_Node, CLevel, CID, TreeList], !).

leaf([_Node, Level, _Id], Proc, Temp_Colors, [Proc|Temp_Colors]).

Figure 8.9: Program for labeling the execution tree.

183
Input Representation:

\[
[ (c,1,1), \\
  (s,2,2), (c,2,3), \\
  (c,3,4), (n,3,5), (n,3,6), (c,3,7), \\
  (n,4,8), (n,4,9), (e,4,10), (e,4,12), (n,4,14), (n,4,15), \\
  (e,5,16),(e,5,18), (e,5,28), (e,5,30) ]
\]

Figure 8.10: Sample execution tree with input representation.
Figure 8.11: Speed-up curves for varying length.

The program shown in Figure 8.9 can be extended to take into account varying times for choice, narrow, split, and enumerate. Random strings of arbitrary length can be drawn using the context-free grammar:

\[ S \rightarrow cSS \mid sSS \mid nS \mid e \]

(c: choice, s: split, n: narrow, e: enumerate)

The speed-ups with varying the number of processors can be analytically calculated. The speed-up curves are shown in Figures 8.11, 8.12, and 8.13. The notation used in the figures is as follows:

L.String-length.c-time.s-time.e-time.n-time

r.c-time.s-time.e-time.n-time
Figure 8.12: Speed-up curves for varying enumeration times.

Figure 8.13: Speed-up curves for varying narrow times.
8.5 Implementation Details

The parallel execution model supported on the shared memory MIMD machine $^1$ is the fork-join model shown in Figure 8.14 b. This model does not map directly to the parallelism exploited in the $solve$ procedure since it uses the parallel forking on the components of the CHOICE and SPLIT constraints assuming an unbounded number of processors. In the case of a finite number of processors, the parallel scheduling is achieved using a stack (or queue) which is shared by the various parallel processors (Figure 8.14 a). Each available processor acquires work from the shared stack. Whenever a processor acquires work, it is said to be busy.

$^1$SGI-Onyx
The arguments of the \textit{solve} procedure are stored in the shared stack. Whenever a processor encounters the \texttt{CHOICE} or the \texttt{SPLIT} statement, it pushes the arguments of the second parallel candidate onto the shared stack and continues execution with the first candidate. (Recall that \texttt{CHOICE} and \texttt{SPLIT} operate on binary trees.) A processor becomes available if it encounters a successful leaf node of the execution tree or if the narrow operation being executed fails. The \textit{solve} procedure terminates when the shared stack is empty and all the processors are free. Otherwise, a free processor attempts to pop the required arguments from the shared stack and continue the execution. The operations with the shared stack are done in a critical region in which only one processor can enter at a given time.

The three critical regions of the program in Figure 8.16 correspond to attempts of a processor to access the global stack (or queue). This occurs when:

1. a \texttt{choice} command is encountered and a new processor is needed; in this case the request for obtaining a new processor is pushed on the stack,

2. a \texttt{split} command is encountered and its execution requires a new processor; the behavior is analogous to that described for the choice command;

3. if a failure is determined or if a processor successfully completes its task then the critical region is entered to find if there are remaining tasks to be executed in the stack.

The parallel procedure \texttt{psolve} (Figure 8.15) forks the required number (say, $N$) of \texttt{psolve\_per\_proc} processes. The main call is \texttt{psolve}(\texttt{p,NIL,iv}), where \texttt{p} points to the root of the program tree, and \texttt{iv} is the interval-vector with all the variables initialized to \((−\infty, \infty)\). The second parameter \texttt{cl} (initialized to \textit{NIL}) contains the list of accumulated constraints. The procedure \texttt{fork} initiates $N$ executions of copies of \texttt{psolve\_per\_proc}
procedure \texttt{psolve}(p,cl,iv) \\
begin \\
\hspace{1em} \texttt{shared \ num\_busy\_procs} \gets 0 \\
\hspace{1em} \texttt{shared \ stack} \gets \texttt{empty} \\
\hspace{1em} \texttt{push}(p,cl,iv) \\
\hspace{1em} \texttt{fork( N, psolve\_per\_proc(\))} \\
end \texttt{psolve}

\hspace{1em} Figure 8.15: The procedure \texttt{psolve}.

procedure (Figure 8.16.)

\section*{Narrowing}

Each constraint in the constraint-list has a unique index (e.g., the constraint’s position in the list). In what follows the notion of a constraint and its index are used interchangeably. The following data structures are useful in implementing the narrow operation efficiently:

\texttt{dependency-list}: For each variable occurring in the constraint-list, the dependency-list is a linked list of the constraints in which the variable appears.

\texttt{constraint-queue}: The constraints yet to be narrowed are processed using the FIFO queue.

\texttt{is-queued}: A Boolean vector to determine if a constraint is already in the queue.

The queue operations are:

\texttt{init\_queue(queue)}: Initialize the empty queue.
procedure psolve_per_proc()

work_completed ← false

while not work_completed do

work_acquired ← false

critical region { if pop(p, cl, iv) then

    num_busy_procs ← num_busy_procs + 1; work_acquired ← true }

if work_acquired then

    while p ≠ NIL do

        switch head(p)

            case CHOICE:

                critical region { push(cons(tail(head(p)), tail(p)), cl, iv) }

                p ← cons(head(head(p)), tail(p))

            case SPLIT:

                split_intervals(head(p), iv, iv1, iv2)

                critical region { push(tail(p), cons(head(p), cl), iv2) }

                p ← tail(p); iv ← iv1

            otherwise: /* a constraint */

                if narrow(cons(head(p), cl), iv, iv') then

                    p ← tail(p); cl ← cons(head(p), cl); iv ← iv'

                else break

            end switch

        end while

    end while

end if

critical region { num_busy_procs ← num_busy_procs - 1 }

end if

critical region { if stack_empty and num_busy_procs = 0 then

    work_completed ← true }

end while
**put_queue(queue, c):** Add the constraint c to the tail of the queue.

**get_queue(queue, op(x,y,z)):** The constraint op(x,y,z) is deleted from the front of the queue. The constraint index c is returned.

When a new constraint is encountered while traversing the syntax tree of the source program, the constraint is initially placed in the constraint-queue. The narrow procedure obtains a constraint from the queue, performs the appropriate narrow operation, and checks if the interval bounds for any of the variables appearing in the constraint have changed. In such a case, the constraints in which the variables appear are added to the queue, if they are not already in the queue (Figure 8.17). The auxiliary procedure *add_constraints* (Figure 8.18) is used to add the constraints to the queue. The interval vector is said to converge when the constraint-queue becomes empty.
**procedure** narrow(cl, iv, iv')

begin

init_queue(constraint-queue) ; is-queued ← false

put_queue(constraint-queue, head(cl)); is-queued_{head(cl)} ← true

iv' ← iv

while not empty_queue(constraint-queue) do

c ← get_queue(constraint-queue, op(x,y,z)); is-queued_c ← false

if narrow_op((x,y,z), iv', (x',y',z')) then

if change(iv', x') then

iv'_x ← x'; add_constraints(x, constraint-queue, is-queued) end if

if change(iv', y') then

iv'_y ← y'; add_constraints(y, constraint-queue, is-queued) end if

if change(iv', z') then

iv'_z ← z'; add_constraints(z, constraint-queue, is-queued) end if

else return false

end while

return true

d  end narrow

---

Figure 8.17: The procedure `narrow`.  

192
\begin{figure}[h]
\begin{verbatim}
procedure add_constraints(var, constraint-queue, is-queued)
begin
    constraint-list-ptr ← dependency-list_{var}
    while constraint-list-ptr ≠ NIL do
        c ← head(constraint-list-ptr); constraint-list-ptr ← tail(constraint-list-ptr);
        if not is-queued_c then
            put_queue(constraint-queue, c); is-queued_c ← true end if
    end while
end add_constraints
\end{verbatim}
\end{figure}

\section{8.6 Example — N Queens}

The $N$ queens problem involves placing $N$ queens on a chess board such that no two
queens attack each other. Let $x_i$ denote the column position of the $i_{th}$ queen. The value
of each $x_i$ lies in the interval $(1, N)$. The following constraints should be satisfied:

- no two queens are in the same column (1),

- no two queens are in the same left or right diagonal (2 & 3)

1. $x_i \neq x_j, 1 \leq i < j \leq N,$

2. $x_i \neq x_j + (j - i), 1 \leq i < j \leq N,$

3. $x_i \neq x_j - (j - i), 1 \leq i < j \leq N.$

The program for the 5-Queens problem is shown in Figure 8.19. The variables
$x_1, x_2, \ldots, x_5$ represent the positions of the 5 Queens. Initially, all the variables are
The 5 - Queens Program

main()
|
| var N: logic;
| var x: array [MAX] of logic;
| |
| N = 5;
| safe(1)
|
| proc safe(i: logic)
|
| |
| CHOICE {
| |
| { N+1 <= i };
| { i =< N;
| BIND x[i] (1, N);
| SPLIT x[i] (L(x[i]), L(x[i]) + 1, U(x[i]));
| safe_1(x[i], i-1);
| safe(i+1)
| }
|
| proc safe_1(y, i, step: logic)
|
| |
| CHOICE {
| |
| { step =< 0 };
| { 1 =< step;
| y <> x[i]; y <> x[i] + step; y <> x[i] - step;
| safe_1(y, i+1, step-1)
| }
|
| The Unfolded Program

SPLIT x1 (L(x1), L(x1)) (L(x1) + 1, U(x1));
SPLIT x2 (L(x2), L(x2)) (L(x2) + 1, U(x2));
x2 <> x1; x2 <> x1 + 1; x2 <> x1 - 1;
SPLIT x3 (L(x3), L(x3)) (L(x3) + 1, U(x3));
x3 <> x1; x3 <> x1 + 2; x3 <> x1 - 2;
x3 <> x2; x3 <> x2 + 1; x3 <> x2 - 1;
SPLIT x4 (L(x4), L(x4)) (L(x4) + 1, U(x4));
x4 <> x1; x4 <> x1 + 3; x4 <> x1 - 3;
x4 <> x2; x4 <> x2 + 2; x4 <> x2 - 2;
x4 <> x3; x4 <> x3 + 1; x4 <> x3 - 1;
SPLIT x5 (L(x5), L(x5)) (L(x5) + 1, U(x5));
x5 <> x1; x5 <> x1 + 4; x5 <> x1 - 4;
x5 <> x2; x5 <> x2 + 3; x5 <> x2 - 3;
x5 <> x3; x5 <> x3 + 2; x5 <> x3 - 2;
x5 <> x4; x5 <> x4 + 1; x5 <> x4 - 1;

(Recall that L and U are lower and upper bounds)

Figure 8.19: Program for the 5 Queens.
in the interval \((1, 5)\). The procedure \textit{safe} places the \(i^{th}\) queen \((x_i, 1 \leq i \leq 5)\). The
BIND statement establishes the lower and upper bounds for the variable \(x_i\). The SPLIT
statement specifies the lower and upper bounds for the two components of the split
variable \(x_i\). The auxiliary procedure \textit{safe}_L generates the constraints such that the
variable \(x_i\) (i.e., the \(i^{th}\) queen) is safe with respect to the already placed queens \(x_j, 1 \leq
j < i\).

The constraints generated by the program in Figure 8.19 are shown in Figure
8.20. The dependency-list for each of the five variables is also shown. The dependency-
list is build as each constraint is encountered. For example, a snapshot of the dependency-
list before and after the constraint (7) is processed is shown in Figure 8.20. The
dependency-list when all the constraints are processed is also presented in Figure 8.20.

The execution-tree for the 5-Queens program is depicted in Figure 8.21. To
simplify the details, the positions of the first two queens are fixed with the values \(x_1 = 1
and x_2 = 3\). The narrowing phase when processing the constraints \((4) \rightarrow (9)\) is also
shown. The table indicates the processed constraint along with the value of the variable
\(x_3\) and the elements in the constraint-queue.
(Unfolded) Constraints

1) x2 <> x1
2) x2 <> x1 + 1
3) x2 <> x1 - 1
4) x3 <> x1
5) x3 <> x1 + 2
6) x3 <> x1 - 2
7) x3 <> x2
8) x3 <> x2 + 1
9) x3 <> x2 - 1
10) x4 <> x1
11) x4 <> x1 + 3
12) x4 <> x1 - 3
13) x4 <> x2
14) x4 <> x2 + 2
15) x4 <> x2 - 2
16) x4 <> x3
17) x4 <> x3 + 1
18) x4 <> x3 - 1
19) x5 <> x1
20) x5 <> x1 + 4
21) x5 <> x1 - 4
22) x5 <> x2
23) x5 <> x2 + 3
24) x5 <> x2 - 3
25) x5 <> x3
26) x5 <> x3 + 2
27) x5 <> x3 - 2
28) x5 <> x4
29) x5 <> x4 + 1
30) x5 <> x4 - 1

Dependency-list

Final Dependency-list

Solution:
(x1,x2,x3,x4,x5) = (1,3,5,2,4)

Figure 8.20: Constraints for the 5 Queens.
Figure 8.21: Details of the narrow operation.
<table>
<thead>
<tr>
<th>Problem</th>
<th>Seq. Time (sec)</th>
<th>Parallel Speed-Up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Processors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>i4</td>
<td>1.55</td>
<td>2.0</td>
</tr>
<tr>
<td>Combustion</td>
<td>63.4</td>
<td>2.0</td>
</tr>
<tr>
<td>9-Queens</td>
<td>10.3</td>
<td>2.0</td>
</tr>
<tr>
<td>13-Queens</td>
<td>160.4</td>
<td>2.0</td>
</tr>
<tr>
<td>Job-Shop (abz6)</td>
<td>42.5</td>
<td>2.0</td>
</tr>
</tbody>
</table>

**Figure 8.22: Speed-ups**

### 8.7 Results

The speed-up achieved on various problems is shown in Figure 8.22. The constraint solver is run on an SGI-Onyx with 16 processors. The problems i4 and combustion are from (HeMK 95). The data for the Job-Shop problems is taken from (ApCo 91).
Chapter 9

Graphical User Interface

In the age of multimedia and visual tools it is of paramount importance to develop user-friendly interfaces. The simplicity of the language facilitates the design of a graphical user interface (GUI) translating commands selected by user actions into statements of the language. The graphical interface expedites the trial and error process of developing and perfecting programs. In addition, tools are available in the GUI to study program behavior.

The GUI for the programming language consists of rolldown menus, text boxes, and buttons (Figure 9.1). The menus enable the user to select the various statements available in the language as well as the split strategies. By means of button clicks, the user can single step in the program execution, examine the values for a select list of variables, and profile the execution statistics.

The user interface is developed in Java and it generates a program that is processed using the Java Compiler Compiler from Sun Microsystems. The following sections describe in detail the various menus and button items that appear in the GUI. This de-
Figure 9.1: Graphical User Interface for the proposed language.
scription covers the two topics: program input and modification, and program execution.

9.1 Program Input and Modification

There are two ways in which statements can be specified in the language: i) the program can be read from a file using the File menu. In that case the program will be loaded into the GUI as a sequence of text boxes as shown in Figure 9.2, or ii) the program can also be constructed interactively with the help of the Statement menu and the button actions Add, Delete, and Edit which are explained below:

• Statement:

As shown in Figure 9.3, the Statement menu allows the user to select the various types of statements that are part of the language. (e.g., Bind, Constraint, Choice, Split, Fail, For, While, Write, etc.) The user selects an item from the menu and clicks the Add button to add the statement to the program.

The following buttons allow the user to interactively create and modify a program:

• Add:

Adds a statement to the existing program. If one or more statements already have been input, the new statement will be added after the last selected statement. Otherwise, the new statement will be appended to the end of the program. The user can then type in the corresponding text boxes to complete the statement. In the case of Figure 9.2, only three types of statements are utilized (bind, constraint, and write). In each case, the text box is filled manually after the corresponding statement is chosen. The number of text boxes that appear at the right side of the
Figure 9.2: The 5-Queens problem.

<table>
<thead>
<tr>
<th>Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>bind x1</td>
</tr>
<tr>
<td>bind x2</td>
</tr>
<tr>
<td>bind x3</td>
</tr>
<tr>
<td>bind x4</td>
</tr>
<tr>
<td>bind x5</td>
</tr>
<tr>
<td>constraint int x1, x2, x3, x4, x5</td>
</tr>
<tr>
<td>constraint x1 &lt;= x2; x1 &lt;= x2 + 1; x1 &lt;= x2 - 1</td>
</tr>
<tr>
<td>constraint x1 &lt;= x3; x1 &lt;= x3 + 2; x1 &lt;= x3 - 2</td>
</tr>
<tr>
<td>constraint x1 &lt;= x4; x1 &lt;= x4 + 3; x1 &lt;= x4 - 3</td>
</tr>
<tr>
<td>constraint x1 &lt;= x5; x1 &lt;= x5 + 4; x1 &lt;= x5 - 4</td>
</tr>
<tr>
<td>constraint x2 &lt;= x3; x2 &lt;= x3 + 1; x2 &lt;= x3 - 1</td>
</tr>
<tr>
<td>constraint x2 &lt;= x4; x2 &lt;= x4 + 2; x2 &lt;= x4 - 2</td>
</tr>
<tr>
<td>constraint x2 &lt;= x5; x2 &lt;= x5 + 3; x2 &lt;= x5 - 3</td>
</tr>
<tr>
<td>constraint x3 &lt;= x4; x3 &lt;= x4 + 1; x3 &lt;= x4 - 1</td>
</tr>
<tr>
<td>constraint x3 &lt;= x5; x3 &lt;= x5 + 2; x3 &lt;= x5 - 2</td>
</tr>
<tr>
<td>constraint x4 &lt;= x5; x4 &lt;= x5 + 1; x4 &lt;= x5 - 1</td>
</tr>
<tr>
<td>write x1, x2, x3, x4, x5</td>
</tr>
</tbody>
</table>
statement depends on its type. In the case of a *Bind* statement, two text boxes appear, the first to specify the variable, and the second to specify the bounds. If the value in the *Statement* menu is *Constraint*, a single text box appears for the user to specify the constraints.
• **Delete:**

Deletes the selected statements from the program. A statement is selected by clicking the checkbox associated with that statement.

• **Edit:**

The *Edit* button lets the user edit a selected statement. Also, the *edit* action pops an auxiliary window properly formatting the selected statement. The user can modify the statement in this window. At the present time, only the text box of a statement can be edited.
9.2 Program Execution

The menu items and button actions presented in this section control the execution of a program. The following three menus specify the traversal mode of the search tree, the manner in which a variable is split during enumeration, and whether one or all the solutions are required:

- **Traversal Strategy:**
  
The options in the *Traversal Strategy* menu determine the order in which variables are selected for interval splitting (e.g., *depth-first*, *round-robin*, *first-fail*, or *fair*). The corresponding menu entries are *DFMode*, *RRMode*, *FFMode*, and *FairMode*, respectively. (Figure 9.4).

- **Split:**
  
The options in the *Split* menu specify how the interval for the selected variable is split (e.g., *first and rest*, *middle*, or *no splitting*). The items in the *Split* menu are:
Figure 9.5: Splitting of a variable.

FRSplit, MidSplit, and NoSplit. (Figure 9.5).

- Solution:

Two options are available in the Solution menu, One, or All (Figure 9.6). If the value is One, the program execution stops when the first solution is found (or if none exist). If the value is All, execution continues until all the solutions are obtained.

The various buttons that assist the user in controlling the program execution and examining the results are explained below (The reader is referred back to Figure 9.1):

- Go:

Clicking the Go button starts the program execution ignoring any break points that are set in its text. If the selected item of the Split rolldown menu is NoSplit, execution stops when the narrowing operations converge for the constraints appearing in the program, or if any of the narrowing operations fail and no more
choices are left on the choice stack. In the former case, the values for the source
variables (those specified with the \texttt{write} statement) are output in the \textit{Execution}
Trace window. In the latter case, there is no solution to the program.

If the selected item of the \textit{Split} rolldown menu is other than \textit{NoSplit}, and if the
resulting interval is not a solution, enumeration of the interval variables takes place
according to the traversal mode selected in the \textit{Traversal Strategy} menu. One or
all of the solutions (if they exist) are output in the \textit{Execution Trace} window based
on the value of the \textit{Solution} rolldown menu.

The trace resulting from executing the program in Figure 9.2 is shown in Figure 9.7.

- \textbf{Break:}

Breakpoints can be set to debug the program execution. One or more breakpoints
can be set by selecting the required statements and clicking on the \textit{Break} button.
This action also toggles the selection and unselection of the breakpoints. If a
breakpoint is already set for a selected statement, clicking the \textit{Break} button removes
Figure 9.7: Execution trace when executing the program in Figure 9.2.

the breakpoint for that particular statement.

- **Watch:**

  Variables can be selected and kept on the watch list by highlighting them from
  the text fields in the statement boxes and clicking on the Watch button. One
  or more variables can be selected at a time and added to the watch list. When
  execution stops at a break point, or when single stepping action is under progress,
  the intervals for the variables in the watch list are output to the Execution Trace
  window.

- **Clear Watch:**

  Clears the list of variables that are currently in the watch list.

- **Step:**
This button is useful in single stepping over each statement in the program. Execution stops after the narrowing operations are performed for each statement. The processed statement is shown as selected in the Statements window. The intervals for the variables in the watch list are output to the Execution Trace window at the end of the Step action.

- **Step Over:**

  Execution continues until a statement is encountered for which the breakpoint is set. The intervals for the variables in the watch list are then output to the Execution Trace window.

- **Interval Stack:**

  When a variable is split, one set of intervals is pushed onto the interval stack and execution continues with the other set. Clicking on the Interval Stack button shows the top 5 entries on the interval stack in the Trace window.

- **Choice Stack:**

  When a Choice statement is executed, the program continues its execution with the first alternative. The other alternatives of the choice statement are pushed onto the choice stack. Clicking on the Choice Stack button shows the top 5 entries on the choice stack in the Execution Trace window.
• **Start Profile:**

The number of various narrowing operations and the number of times the split operation occurs can be profiled by clicking on the *Start Profile* button. Once clicked, the button is disabled.

• **Stop Profile:**

Clicking on the *Stop Profile* button stops the profiling operation. The *Start Profile* button is then enabled and the *Stop Profile* button is disabled.

• **Show Profile:**

The profile statistics are presented in the *Execution Trace* window when the user clicks on the *Show Profile* button. The statistics for the narrow operations shows in detail the number of narrowing steps performed for each statement, as well as the number of narrowing operations for each operator.

• **Clear Profile:**

The profile data collected so far is cleared.

### 9.3 Trace and Debug Windows

The *Execution Trace* window shows the intermediate values of the intervals, the results of the program, the profile statistics, etc. The *Debugging Messages* window (see Figure 9.1) shows any syntax errors in the program and also records the user actions when various buttons are clicked.

Most of the programs described in Chapter 4 can be easily input and executed using the GUI which is made available on the internet at [http://www.cs.brandeis.edu/~suresh/thesis](http://www.cs.brandeis.edu/~suresh/thesis).
Chapter 10

Conclusions

The main original contributions of this work can be summarized as follows:

1. Design and development of a concise but general programming language based on non-determinism, constraints and intervals. That design includes a GUI that is available for download through the Web.

2. Presentation of a varied corpus of examples that are representative of the breadth of problems that may be solved using the proposed language.

3. In-depth study of data parallelism and shared memory parallelism in the language implementation. This has been carried out using actual parallel computers.

4. Estimates through actual benchmarks of the attainable speed-ups in both kinds of parallelism. In addition, formal models of speed-up analyses were investigated using context-free grammars for shared memory implementation, and Markovian models for data parallel implementation.

5. Demonstration of the feasibility of developing useful preprocessors that transform annotated specialized programs into programs in the proposed language. They
include CSP, partial evaluation, and scheduling problems.

10.1 The Aim of the Theoretical Analysis

The performance of a program can be measured in two ways. The first is experimental: it consists of establishing a set of benchmark programs, and measuring their execution time. The second approach is to examine analytically (or symbolically) the program in question and develop machine independent analytic formulas representing their performance.

A major drawback of benchmark analyses is that they are specific to a given machine. A new set of benchmarks is needed when the same program is run on different machines. The work on micro-analysis (CoWe 92) is based on the introduction of time-variables to construct formulas (called time-formulas) that describe the micro-complexity of programs. Once the formula is obtained, one can bind the time variables to actual values corresponding to a specific machine and plot execution time versus various program parameters.

In this work, we have purposely incorporated micro-analyses of the parallel versions of narrowing, and the exploration of choices with enumeration. There are considerable benefits in doing so:

- The presented speed-up results are machine independent, and
- The formulas allow experimenting with different values of various parameters.

In Chapters 7 & 8 context-free grammars are used in constructing (approximate) time-formulas for the parallel versions. In a Markovian analysis, presented in Chapter 7, the flowchart of a program is viewed as probabilistic finite-state machine and a time-formula is derived using matrix algebra. Context-free languages (expressing parenthetical
structures) are used to provide time-formulas expressing speed-up. In that case, the parenthetical structures describe parallel executions that are spawned and eventually joined during program execution.

In this dissertation, the efficiency analysis is presented by: (1) time-formula generation and analysis, and (2) empirical results. It should be pointed out that it is trivial to determine the specific values of the time-variables by using the empirical results. This is not explicitly done in this work since we only had access to specific machines (one MIMD, and the other SIMD). Nevertheless, a reader should keep in mind that the micro-analysis is always useful in attempting to estimate efficiencies for different machines.

10.2 Future Work

As in any research work involving software, enhancements are always desirable. Although many language extensions can be envisioned, they would have to adhere to the initial design tenets of achieving simplicity combined with ease for parallel processing.

Among the many possible enhancements, the following would likely have wider interest. They are listed in the order of the efforts needed to accomplish them.

- Develop actual preprocessors for problem specifications that could be easily translated into the programs in the proposed language. As mentioned earlier, these could include preprocessors for CSP, scheduling, and partial evaluation.

- Augment the GUI with actions to automatically generate data parallel and OR parallel programs and display the results including speed-up data.

- Introduce data flow analyses to optimize parallel performance by minimizing com-
munication costs.

- Investigate the possible automatic generation of redundant constraints to speed-up convergence. (van Emden’s work (EMDE 97) is a first step in that direction.)

- Finally, this work suggests a possible hybrid implementation where the constraints are processed using the data parallel mechanism, and the non-deterministic choices are explored in parallel on the shared memory parallel computers. The data parallel processor accepts the input data from the shared memory processor, performs the narrowing operations in parallel, and transmits the results to the shared memory processor.

10.2.1 Distributed Implementation

The work on shared-memory parallelism presented in this dissertation paves the way for a distributed model. Distributed applets communicating with other applets located on different machines collaborate in solving a particular problem. For example, an applet can be built to narrow a given set of constraints with a given set of intervals. When a task contains subtasks, the applet continues execution with the first task and delegates the remaining tasks to another applet. This is similar to the shared-memory approach where the main process continues execution with the first task and pushes the remaining tasks onto a shared stack.

Distributed applets can be supported by implementing all objects as network objects. The methods of a network object can be invoked by other processes, in addition to the process that created the object. The initial connection between two processes occurs when one process registers an object with a name server under a unique name, and another process subsequently imports the object from that name server. Once the
<table>
<thead>
<tr>
<th></th>
<th>Shared-memory</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Narrowing</td>
<td>Sequential within a process.</td>
<td>Sequential within an applet.</td>
</tr>
<tr>
<td>Choice &amp; Split Enumeration</td>
<td>Parallel fork: Push remaining tasks onto a shared stack. Continue execution with the first.</td>
<td>Remaining tasks delegated to another applet and execution continues with the first.</td>
</tr>
<tr>
<td>Processes</td>
<td>Main process spawns sub-processes.</td>
<td>The applets exist independently on different machines and register themselves with a name server.</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Problem done when shared stack is empty and all processors are free.</td>
<td>Different applets coordinate with the name server when the sub-problems are completed.</td>
</tr>
</tbody>
</table>

Figure 10.1: Outline of similarities between parallel shared-memory and distributed versions.

connection is established, network objects can be passed between processes just like passing any other type of data (BrNa 97).

The table in Figure 10.1 summarizes the similarities between shared-memory representation and a distributed one.

### 10.3 Final Remarks

Language design and implementation often requires significant and time consuming efforts to provide users with efficient processing, seamless interaction with an operating
system, and helpful debugging facilities. In addition, the language implementor should purvey clear manuals and multiple examples of problems suggesting potential applications. Very few languages can be thoroughly completed for wide public acceptance within the scope of a doctoral dissertation in an academic setting. There is an undeniably strong academic component in this work, and that pertains to the study of parallelism and algorithm analysis. Nevertheless, the author’s goal is to transcend simple language prototyping by offering a user-friendly environment and practical examples covering a broad spectrum of applications. Such effort should enable potential users to gain an understanding of the usefulness of the new features that have been proposed.

It should be emphasized that, although this work presents specific implementations, they are per se secondary to the concepts and analyses that are contained within the dissertation. The implementations comprising the MIMD and SIMD architectures are needed to justify the feasibility of the proposed language and this work would be incomplete without the experience that has been gained through them.

The ultimate test of a language is its acceptance by a larger number of practitioners. This dissertation provides the groundwork for making that acceptance a reality. It is hoped that the simplicity and broad scope of the language will attract followers interested in solving combinatorial problems involving interval constraints.
Appendix A - BNF Specification

<program>
   ::= 'MAIN' '(' ')' '	'
       '{' <declarations> <traversal_mode> <split_strategy> <precision> <compound_statement> '}'<subprogram_declarations>

<declarations>
   ::= <declarations> 'VAR' <declaration> ';' |

<declaration>
   ::= <identifier_list> ':' <type>

<identifier_list>
   ::= <identifier> | <identifier_list> ',' <identifier>

$type$
   ::= <standard_type> | 'ARRAY' ['<integer_list> '] 'OF' <standard_type>

<integer_list>
   ::= <integer> | <integer_list> ',' <integer>

<standard_type>
   ::= 'INTERVAL' | 'REAL' | 'INTEGER'

<traversal_mode>
   ::= 'DFMODE' | 'RRMODE' | 'FFMODE' | 'FAIRMODE'

217
<split_strategy>
 := 'FRSPLIT'
 | 'MIDSPLET'
 | 'NOSPLIT'
 |

<precision>
 := 'PRECISION' <constant>
 |

<compound_statement>
 := '{' <optional_statements> '}'

<optional_statements>
 := <statement_list>
 |

<statement_list>
 := <statement>
 | <statement> ';' <statement_list>

<statement>
 := <expression> 'EQUALS' <expression>
 | <expression> <comp_op> <expression>
 | <procedure_statement>
 | <compound_statement>
 | 'BIND' <identifier> '(' <constant> ',', <constant> ')' 
 | 'IF' <test> 'THEN' <statement>
 | 'IF' <test> 'THEN' <statement> 'ELSE' <statement>
 | 'WHILE' <test> 'DO' <statement>
 | 'DO' <statement> 'WHILE' <test>
 | 'FOR' '(' <identifier> 'EQUALS' <expression> ';' <test> ';;'
 | <identifier> 'EQUALS' <expression> ')
 | <statement>
 | 'WRITE' <expression>
 | 'INT' <identifier_list>
 | 'BOOL' <identifier_list>
 | 'CHOICE' '{' <compound_statement_list> '}'
 | 'EITHER' <statement> 'OR' <statement>
 | 'SOME' '(' <identifier> 'EQUALS' <expression> ';;' <test> ';;'
 | <identifier> 'EQUALS' <expression> ')
 | <statement>

218
| `REDQ` <statement> `AFTER` <statement>
| `SPLIT` <identifier> `(' <expression> `,' <expression> `)`
| `(' <expression> `,' <expression> `)`
| FAIL

<compound_statement_list>
  ::= <compound_statement>
  | <compound_statement> `;' <compound_statement_list>

<procedure_statement>
  ::= <identifier>
  | <identifier> `(' <expression_list> `)`

<expression_list>
  ::= <expression>
  | <expression_list> `,' <expression>

<expression>
  ::= <expression1>
  | <expression> <op2> <expression1>

<expression1>
  ::= <expression0>
  | <expression1> <op1> <expression0>

<expression0>
  ::= <identifier>
  | <constant>
  | `L` `(' <identifier> `)`
  | `U` `(' <identifier> `)`
  | `(' <expression> `)`
  | <identifier> `[' <expression_list> `]`
  | <uop> <expression>
  | <expression> `POWER` <expression>

<op2>
  ::= `PLUS`
  | `MINUS`
  | `OR`

<op1>
  ::= `TIMES`
  ::= `DIV`
  ::= `AND`

<uop>
 ::= 'NOT'
 | 'UMINUS'
 | 'SQRT'
 | 'INTEGER'
 | 'SIN'
 | 'COS'
 | 'TAN'
 | 'ASIN'
 | 'ACOS'
 | 'ATAN'

<test>
  ::= <expression> <comp_op> <expression>
   | '(' test ')'  

<comp_op>
  ::= EQ
   | NEQ
   | LEQ
   | GEQ

<identifier>
  ::= 'IDENTIFIER'

<integer>
  ::= 'INTEGER'

<constant>
  ::= 'CONSTANT'

<subprogram_declarations>
  ::= <subprogram_declarations> <subprogram_declaration>
   |  

<subprogram_declaration>
  ::= <subprogram_head> '{' <declarations> <compound_statement> '}'

<subprogram_head>
  ::= 'PROC' <identifier> <arguments>

<arguments>
  ::= '(' <parameter_list> ')'
   |  

<parameter_list>

220
::= <identifier_list> ‘;’ <type> 
| <parameter_list> ‘;’ <identifier_list> ‘;’ <type>
A.2 - YACC Specification and Actions

program
  : definitions MAIN '(' ')' '{' declarations
     compound_statement '}'
subprogram_declarations
  { $$ = node("main",$6,$7,$9);
     install_procs($9); install_defines($1); install_DECLS($6);
     traverse_statements_main($7);
  }
compound_statement
  : '{' optional_statements '}'  {$$ = $2; }
optional_statements
  : statement_list
    | {$$ = NIL; }
statement_list
  : statement          
    | statement ';'; statement_list
    {$$ = node(":\",$1,$3,NIL);}
statement
  : expression comp_op expression
    {$$ = node($2,$1,$3,NIL); }
  | procedure_statement
  | compound_statement
  | IF test THEN statement
    {$$ = node("if-then",$2,$4,NIL); }
  | IF test THEN statement ELSE statement
    {$$ = node("if-then-else",$2,$4,$6); }
  | WHILE test DO statement
    {$$ = node("while",$2,$4,NIL); }
  | DO statement WHILE test
    {$$ = node("do","$2","$4","NIL"); }
  | FOR '{' opt_statement ';'; opt_test ';'; opt_statement '}' statement
    { $$ = node("for","node("for-test","$3","$5","7");","9","NIL"); }
  | WRITE expression_list
    {$$ = node("write","$2","NIL","NIL"); }
  | INT identifier_list
    {$$ = node("int","$2","NIL","NIL"); }
  | BOOL identifier_list
    {$$ = node("bool","$2","NIL","NIL"); }
  | CHOICE '{' compound_statement_list '}'
    {$$ = $3; }
  | EITHER '{' statement_list '}' OR C '{' statement_list '}'
    {$$ = node("cp","$3","$7","NIL"); }
  | REDO '{' statement_list '}' AFTER '{' statement_list '}'
    {$$ = node("redo","$3","$7","NIL"); }
  | SOME '{' opt_statement ';'; opt_test ';'; opt_statement '}' statement
    { $$ = node("some","node("some-test","$3","$5","7");","9","NIL"); }
  | TRUE
    { $$ = node("true","NIL","NIL","NIL"); }
  | FAIL
    { $$ = node("fail","NIL","NIL","NIL"); }
| CUT     { $$ = node("cut",NIL,NIL,NIL); }  |
| NOSPLIT { $$ = node("nosplit",NIL,NIL,NIL); }  |
| SPLIT identifier expr2 expr2  |
   { $$ = node("split",$2,$3,$4); }  |
| BIND identifier ' [ expression0 , expression0 ] '  |
   { $$ = node("bound",$2,$3,$4); }  |
Bibliography


P. Baptiste, C. L. Pape, and W. Nuijten, *Constraint-Based Optimization and Approximation for Job-Shop Scheduling*, in Proceedings of the AAAI-SIGMAN Workshop on


pp. 79-94.


Y. Colombani, *Constraint Programming: an Efficient and Practical Approach to Solving*
the Job-Shop Problem, Lecture Notes in Computer Science, no. 1118, Spinger, 1996.


P. A. Dwyer, Linear Computations, John Wiley & Sons, 1951.

M. H. van Emde, Canonical Extensions as Common Basis for Interval Constraints and Internal Arithmetic, Department of Computer Science, University of Victoria.


B. N. Freeman-Benson, J. Maloney, and A. Borning, An Incremental Constraint Solver,


P. V. Hentenryck, D. McAllester, and D. Kapur, Solving Polynomial Systems Using a


T. Hickey, CLP(F) and Constrained ODEs, Workshop on Constraint Languages and their use in Problem Modelling, International Logic Programming Symposium, 1994.


E. Shapiro, *The Family of Concurrent Logic Programming Languages*, ACM Computing


K. S. Trivedi, *Probability & Statistics with Reliability, Queuing, and Computer Science*


