Parallel Constraint Propagation

by

Sinesie Calin Somosan

M.Sc., Technical University of Timisoara, Romania, 1998

B.Sc., Technical University of Timisoara, Romania, 1997

A Thesis Submitted in Partial Fulfillment of the

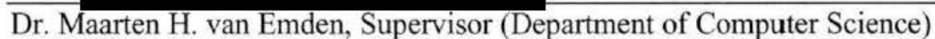
Requirements for the Degree of

MASTER OF SCIENCE

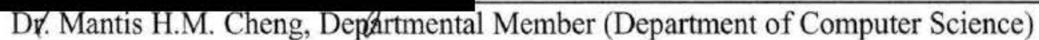
in the Department of Computer Science

We accept this thesis as conforming

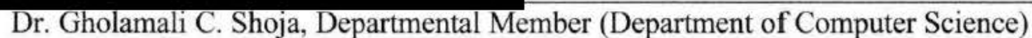
to the required standard



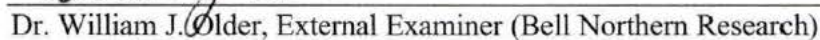
Dr. Maarten H. van Emden, Supervisor (Department of Computer Science)



Dr. Mantis H.M. Cheng, Departmental Member (Department of Computer Science)



Dr. Gholamali C. Shoja, Departmental Member (Department of Computer Science)



Dr. William J. Older, External Examiner (Bell Northern Research)

© Sinesie Calin Somosan, 2001

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

QA297.75
S66

Supervisor: Dr. Maarten H. van Emden

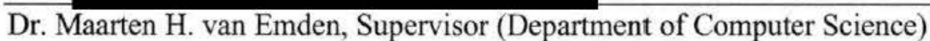
Abstract

Interval constraints is a recently developed method used in numerical computation, and has many applications, such as solving systems of non-linear equations, complex function plotting, etc.

An important aspect of solving a problem in terms of interval constraints is the constraint propagation process, which aims to eliminate the inconsistent values from the candidate intervals for the unknowns by repeatedly applying the constraint contraction operator on each primitive constraint. The fact that there is no restriction regarding the order, in which the constraint contractions have to be performed, makes constraint propagation suitable for parallel computation.

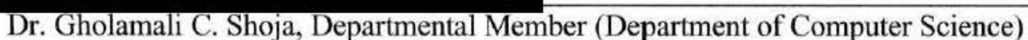
This thesis presents a distributed architecture for performing constraint propagation using coordination between a set of cooperative communicating agents. More specifically, the constraint propagation problem is described as a coordination of a set of cooperative agents, each agent having a very specific role in the process of solving the problem.

Examiners:

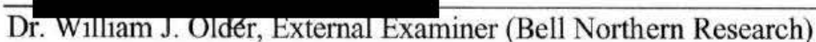


Dr. Maarten H. van Emden, Supervisor (Department of Computer Science)

Dr. Mantis H.M. Cheng, Departmental Member (Department of Computer Science)



Dr. Gholamali C. Shoja, Departmental Member (Department of Computer Science)



Dr. William J. Older, External Examiner (Bell Northern Research)

Table of Contents

Abstract ii

Table of Contents iii

List of Tables viii

List of Figures ix

Acknowledgement xi

Dedication xii

1 Introduction 1

1.1 The problem domain 1

1.2 A larger context 3

1.3 Related work 4

1.4 Proposed solution 5

1.5 Thesis overview 6

2 Informal Introduction to Interval Constraints 7

2.1 Real and floating-point numbers 7

2.2 Interval constraints method 8

2.3 Constraint propagation 12

2.3.1 A constraint propagation algorithm 12

2.3.2 Example of constraint propagation 15

2.4 Interval splitting ... 17

3 Parallel Systems ... 19

3.1 Granularity ... 19

3.2 Models for parallel processing ... 21

3.2.1 Shared memory paradigms ... 22

3.2.2 Distributed memory paradigms ... 23

3.3 Coordination languages ... 25

3.3.1 The COOL coordination language ... 26

3.4 Termination detection ... 27

4 Parallel Constraint Propagation ... 29

4.1 Suitability for parallelism ... 29

4.2 Proposed architecture ... 30

4.3 Sharing variables between constraints ... 33

4.4 Constraint propagation initialization ... 35

4.5 Termination detection ... 36

4.6 Communication between agents ... 39

4.7 The Monitor agent ... 40

4.7.1 The activity of the Monitor ... 40

4.7.2 Data handled by the Monitor ... 42

4.7.3 Messages handled by the Monitor ... 42

4.8 The Worker agent ... 43

4.8.1 The activity of a Worker ... 43

4.8.2 Data handled by a Worker ... 46

4.8.3 Messages handled by a Worker ... 47

4.9 The ConstraintSystem agent 48

4.9.1 The Activity of the ConstraintSystem 48

4.9.2 Data handled by the ConstraintSystem 49

4.9.3 Messages handled by the ConstraintSystem 50

4.10 Constraint system partitioning 51

4.10.1 Constraints clustering 51

4.10.2 Partitioning algorithm 53

4.11 Constraint propagation failure 55

5 Implementation issues 56

5.1 Variables/Constraints naming scheme 56

5.2 The passive objects 57

5.2.1 The `Constraint` class 57

5.2.2 The `Variable` class 58

5.2.3 The `Propagator` class 60

5.2.4 The `SharedVar` class 64

5.2.5 The `PPropagator` class 64

5.3 The active objects 67

5.3.1 The `Monitor` actor 68

5.3.2 The `Worker` actor 70

5.3.3 The `ConstraintSystem` actor 72

5.4 The overall interaction between actors 75

5.5 `Monitor` – `Workers` interaction 77

5.6 Constraint system partitioning 79

6 System's performance 81

6.1 Premises 81
6.2 Measurements significance 82
6.3 Test cases interpretation 85
6.3.1 Test case 1 85
6.3.2 Test case 2 85
6.3.3 Test case 3 86
6.3.4 Test case 4 87
6.4 The CPI parameter 88
6.5 Conclusions 89
7 Conclusions 90
7.1 Research motivation 90
7.2 Accomplishments 91
7.3 Future work 93
Glossary 94
Bibliography 102
Appendix A Source code 105
A.1 The passive objects 105
A.1.1 The `Variable` class header file 105
A.1.2 The `SharedVar` class header file 106
A.1.3 The `Constraint` classes header file 107
A.1.4 The `Propagator` class header file 108
A.1.5 The `PPropagator` class header file 109
A.1.6 The `ICSParser` class header file 110
A.1.7 The `ICSPartitioner` class header file 111

A.1.8 The `ICSDistributor` class header file 112

A.1.9 The `ICSTypes` header file header file 112

A.2 The active objects 113

A.2.1 The `Monitor` actor interface 113

A.2.2 The `Worker` actor interface 114

A.2.3 The `ConstraintSystem` actor interface 115

Appendix B `Propagator` usage 116

B.1 The problem 116

B.2 The source code 116

B.3 The result 118

List of Tables

Table 6-1 Parallel constraint propagation measurements 83

List of Figures

Figure 2.1 Primitive constraints 9

Figure 2.2 Interval constraints conceptual model 11

Figure 2.3 Constraints network 12

Figure 2.4 A sequential constraint propagation algorithm 13

Figure 2.5 Constraints-unknowns dependency network 16

Figure 4.1 Parallel constraint propagation - high-level architecture 31

Figure 4.2 Conceptual model for a parallel constraint propagation 33

Figure 4.3 Monitor agent state-chart 41

Figure 4.4 Worker agent state-chart 44

Figure 4.5 ConstraintSystem agent state-chart 49

Figure 4.6 The partitioning algorithm 54

Figure 5.1 The interface of the `Constraint` class 57

Figure 5.2 `Variable` class code snippet 59

Figure 5.3 `Propagator` class code snippet 61

Figure 5.4 `Variable-Constraint-Propagator` sequence diagram 63

Figure 5.5 The `SharedVar` class code snippet 64

Figure 5.6 `PPropagator` class code snippet 66

Figure 5.7 The interface of the `Monitor` actor 69

Figure 5.8 The interface of the `Worker` actor 71

Figure 5.9 The interface of the `ConstraintSystem` actor 73

Figure 5.10 The overall interaction diagram between actors 76

Figure 5.11 `Monitor` - `Workers` interaction diagram 78

Figure 5.12 The partitioning modules and their interaction 79

Figure 6.1 Test case 3 86

Figure 6.2 Test case 4 87

Acknowledgement

First of all, I would like to express my gratitude and appreciation to my supervisor, Professor Maarten van Emden, for the advice and guidance that he gave me during the complete course of this research and the preparation of this manuscript. His help was invaluable and greatly appreciated. I am also very grateful for the opportunity that he offered me to pursue my Master's degree under his supervision at the University of Victoria. The research assistantship provided by him is also very much appreciated.

I would like to thank my examining committee, Dr. M. Cheng, Dr. A. Shoja and Dr. W. Older for their precious time spent for reading this thesis and for their comments and suggestions for improving this thesis.

Many thanks to my colleague Gordon O'Connell for the valuable discussions that we had and for the suggestions and references that he provided me during the course of my research.

Finally, I thank my wife Varinia for her infinite love, support and motivation. I am also deeply indebted to my parents for making possible for me to be what I am today.

Dedication

To Varinia,

for her unconditioned love and support.

1 Introduction

1.1 The problem domain

Numerical computation has many applications in various scientific and engineering fields such as mechanics, chemistry, robotics, electronics, weather prediction, economics, and graphics, to name just a few. However, there are two main issues when conventional numerical computation methods are used to solve such kind of problems. Firstly, many of these numerical problems cannot even be effectively formulated in conventional terms. Secondly, one of the biggest problems that conventional numerical computation encounters is rounding errors. These errors are introduced due to the fact that computers have finite representation capabilities, and as a consequence, a real number from an idealized representation of a problem is approximated with a floating-point number inside a machine. This approximation leads to rounding errors, which in many applications are unacceptable and can lead to compromised results.

Interval constraints is a recently developed numerical computation method, which tries to eliminate the above indicated problems. It already has many applications, such as solving systems of non-linear equations, function plotting, etc. The main benefit of interval constraint is that it allows the potential of solving problems that cannot even be effectively formulated in conventional terms. Its main strength is the fact that it provides guarantees in terms of correctness and completeness with respect to the yielded result. It produces intervals as the solution to a problem, and guarantees, that if there is any solution for the unknowns, then it belongs to the computed intervals.

In the sphere of interval constraints a problem is described through a large set of relations, called primitive constraints, which have a large number of unknowns. Each unknown has an interval associated with it as opposed to just a real number, interval that indicates the set of values that the unknown can take. Finding a solution for such a problem means finding the smallest intervals for all the unknowns for which the given set of relations is still satisfied. To accomplish this, interval constraints uses a contraction operator associated with each primitive constraint to contract the intervals associated with the unknowns. Interval constraints uses the operations on intervals defined by interval arithmetic to implement its contraction operators.

An important aspect of solving an interval constraints system is the *constraint propagation* process. This process aims to eliminate the inconsistent values from the candidate intervals for the unknowns by repeatedly applying the constraint contraction operator on each primitive constraint. However, for many real world applications the interval constraints method does not provide satisfactory results in terms of speed. Although, the fact that there is no restriction regarding the order, in which the constraint contractions have to be performed, makes the constraint contractions process suitable for parallel computation.

Since the beginning of computers, almost any computation machine was conceived in the same vision of the *von Neumann* architecture: one central processing unit connected to a single bank of memory, used to execute one single program at a time. However, things have been changing dramatically recently, and parallel architectures have become valuable options for developing high performance systems that can be used to solve large problems in many fields of applications.

The main motivation for parallelism is the significant gain in speed that is achieved, and the fact that there are plenty of very complex problems that require this gain in performance, and for which the single processor machine is still not sufficient. Some examples of such kind of applications are: modeling global weather patterns, genetic studies, nuclear-level simulations, etc.

Moreover, there is also an economical reason why parallelism is tempting, and this is the fact that one single expensive very fast processor is no longer an attractive solution, as

opposed to a cluster of cheap off-the-shelf microprocessors interconnected through some kind of wires that offers the equivalent performance.

The aim of this thesis is to present a parallel architecture for solving the constraint propagation problem. The main motivation behind such a parallel architecture is that there are many numerical problems for which interval constraints requires a lot of computation and the speed-up gained by a parallel approach is worth while.

1.2 A larger context

Before proceeding further, it is worth determining the place of solving an interval constraints problem in a larger context.

Constraint programming is a very interesting alternative to conventional programming because it can solve problems that conventional programming cannot. It consists of specifying and solving a so-called *constraint satisfaction problem* (CSP) [8,9,10]. A CSP consists of a set of *unknowns* that are part of a problem, a set of associated *domains* of values, and set of *constraints*, each on a subset of the whole set of unknowns. A CSP has the goal of finding values for the set of unknowns, which simultaneously satisfy the whole set of constraints. Within a CSP a domain associated with an unknown represents the set of possible values that the unknown can take at a certain stage of solving the problem.

Constraint propagation is one of the most widely known ways of solving a CSP by finding a solution that satisfies a set of constraints. A constraint propagation algorithm is an instance of a more general mathematical framework called *chaotic iteration* [4]. For a particular CSP, a set of so-called *domain reduction functions* are constructed, according to the domain of computation and the set of constraints of the CSP. During the constraint propagation these domain reduction functions are used to reduce the domains for unknowns in a way that the restrictions imposed by constraints are always satisfied.

Considering the above remarks one can easily identify that solving an interval constraints problem is just a particular case of solving a CSP. More specific for interval constraints is

that the unknowns are real numbers and that their domains are intervals. The domain reduction function for interval constraints is the contraction operator.

1.3 Related work

The need for parallelism, for constraint solvers in general, and for constraint propagation in particular, has recently received wide attention. There are various solutions to this problem proposed by different authors.

In [5] a generic framework for distributed constraint propagation based on the notion of chaotic iteration [4] is presented. The framework is generic for two reasons. Firstly, in the sense that it can be instantiated to do propagation on a particular domain by specifying the computation domain and the domain reduction functions for that domain. Secondly, because the strategy used to schedule the application of the domain reduction functions can also be chosen. Informally, the framework consists of several agents, enhanced with communication mechanism, each executing the chaotic iteration algorithm. Locally, each agent manages a subset of the domain reduction functions and a subset of unknowns and exchanges information through asynchronous message passing with other agents. Globally, through cooperation, the system computes the desired solution to the whole problem. All agents are identical and execute the same algorithm.

A similar solution to the one presented in [5] is given in [6]. The difference is that, as opposed to [5], where the agents are distributed on a network of machines, here the system consists of a collection of parallel agents that communicate through a shared memory.

A more generic framework than the ones presented in [5,6], is presented in [7]. Some of the benefits added are the dynamic aspect of repartitioning the problem and the possibility of dynamically changing the domain reduction functions. The key idea, on which this framework is based, is to describe the constraint propagation as a coordination of cooperative agents using coordination languages. The coordination language used is MANIFOLD, which is a language for managing complex, dynamically changing interconnections among sets of independent, concurrent cooperative processes. The

framework exploits the relation between constraint propagation and cooperative agents. The components of the constraint propagation algorithm are mapped on components that MANIFOLD provides. Unknowns are associated with extended variables, which are some predefined processes in the language, and domain reduction functions are converted in worker agents. Channels are used to connect variables with workers. Domain reduction functions receive requests from variables, whenever the domains of the variables are modified, to reapply themselves. Variables are notified when changes appear. This process proceeds until no variable can have its domain reduced anymore.

A study of a parallelized version of an algorithm for constraint propagation, for both a multiprocessor and a multicomputer system, is presented in [11,12]. The multiprocessor version of the algorithm is suitable for the EREW PRAM model. The distributed version of the algorithm has two materializations, one for fine-grained interconnected networks, and one for coarse-grained interconnected networks. The distributed version of the algorithm is a particular case of the algorithm presented in [5].

1.4 Proposed solution

This thesis presents a distributed architecture for solving the constraint propagation process using coordination between a set of cooperative communicating agents. Roughly, the proposed solution is a virtually homogenous distributed architecture, in the sense that each computation node virtually runs the same propagation algorithm on a part of the problem to be solved. Concretely, the constraint system that has to be solved is partitioned in a set of subsystems that are solved in parallel, preferably one on each computation node. However, this homogenous aspect of the model is just a visualisation from the computation point of view that has to be carried out.

In reality, the constraint propagation problem is described as a coordination of a set of cooperative agents, each agent having a specific role in the process of solving the problem. Each agent communicates with the other agents so that the entire problem is solved as a common cooperative process. Concretely, a particular agent called Worker

solves each subsystem by continuously exchanging information about the partial solution that it has found with the other Worker agents.

Solving a constraint satisfaction problem is a process of alternating the constraint propagation process with the splitting process. There is no splitting present in the proposed architecture. However, due to the modular aspect of the presented architecture adding splitting should not be difficult, but it should be just a matter of creating a new type of agent and integrating it into the existing model.

The proposed architecture has been implemented in the coordination language COOL [24, 25, 26], and tested on a distributed system of Intel workstations connected through an Ethernet network running Linux.

1.5 Thesis overview

Chapter 2 contains an informal introduction to interval constraints and some related concepts. Chapter 3 introduces concepts and models related with parallel systems relevant in the context of this thesis. A brief overview of the most known parallel programming paradigms is given. Chapter 4 describes the architecture and a high level design of a parallel constraint propagation proposed in this thesis. This chapter starts with indicating the suitability that constraint propagation has for parallelism, and continues with a detailed description of the proposed model. Chapter 5 describes an implementation in C++/COOL of the model presented in chapter 4. The experimental results indicating the performance of the implementation given in chapter 5 are described in chapter 6. Chapter 7 concludes the thesis by presenting the motivation of this work, the accomplishments, and some possible future extensions.

A comprehensive glossary that aims to eliminate any ambiguity by defining the relevant terms and concept used in this thesis is also provided. Appendix A lists the source code header files of the classes that implements the core functionality for a constraint propagation, and also the interfaces for the main agents used to perform a parallel constraint propagation. Appendix B contains a sample program of using the `Propagator` class for a sequential constraint propagation.

2 Informal Introduction to Interval Constraints

This chapter gives a brief introduction to interval constraints method and some of the most important concepts related to it used in this thesis. A comprehensive presentation of interval constraints can be found in [1].

2.1 Real and floating-point numbers

This section presents the problem that lead to interval arithmetic and interval constraint methods and their use in numerical computation.

One of the problems that are faced when numerical computation problems involving real numbers are to be solved using computers is the finite aspect of computers. The problem results from the fact that computers have only a finite representation capability, whereas the ideal real number set is an infinite one. As a consequence, in reality only a finite subset of the real number set can be represented within a machine.

To facilitate numerical computations that involve real numbers to be performed using computers, the IEEE 754 standard [14] defines floating-point numbers and a set of associated operations.

If we consider R the set of extended reals [15] as being the real number set extended with the two infinity symbols and the extension of the relation $<$ to this set, then the floating-point set of numbers is a subset of R that is machine representable. Therefore, the

floating-point set of numbers includes $-\infty$, $+\infty$, and 0, and moreover it is an ordered set of numbers.

In this context, for any real number *r*, r^+ and r^- can be defined [15] as being the smallest floating-point number not smaller than *r*, respectively the greatest floating-point number not greater than *r*.

There are some problems with floating-point numbers. First of all, it is obvious that a real number *r* might not be representable exactly by a floating-point number. Secondly, an operation between floating-point numbers might yield a result that is not representable exactly by a floating-point number. Therefore, a real number *r* can be approximated either by r^+ or by r^-. The same thing can be said about the result of an operation (i. e. a+b). To overcome this problems, the IEEE 754 standard [14] defines rounding operations that allow software to approximate real numbers and results of operations to some close corresponding floating-point numbers. More specifically, the standard defines four rounding modes: round to nearest, round toward zero, round toward $+\infty$, and round toward $-\infty$.

It is worth mentioning that this rounding strategy solves some of the problems but not all of them. There are many complex problems where repeated rounding errors can lead to severe errors and even to the compromise of the final result. To overcome this kind of problems interval arithmetic provides a way of bounding numerical errors automatically, by returning as a result an interval that contains the real number, as opposed to just one floating-point number that is an approximation of the real number.

2.2 Interval constraints method

A problem in terms of interval constraints is described through a set of complex constraints. A *complex constraint* is a mathematical equality or inequality that involves an arbitrary number of unknowns. An *unknown* is the element for which the interval constraints method seeks a value. Each unknown has an associated interval, which indicates the domain of values that the unknown can take. The result provided by the interval constraints method is a set of intervals for the set of unknowns. The goal of the

method is to find as much information about the solution as possible. A better result means a smaller interval for an unknown, that is, more information about the solution.

To solve a problem described through a large set of complex constraints, the interval constraints method translates each complex constraint into a set of primitive constraints. A set of most common primitive constraints is presented in Figure 2.1.

$$\begin{aligned} & x + y = z \\ & x * y = z \\ & x^n = y, \text{where } n \text{ is integer} \\ & x = y \\ & x \leq y \end{aligned}$$

Figure 2.1 Primitive constraints

To illustrate this process of translation the following set of complex relations are considered:

$$\begin{cases} (x+2)^2 + (y-3)^2 = 1 \\ x + y = 1 \\ 2 * x + 6 \leq 3 * y \end{cases}$$

Using the set of primitive constraints listed in Figure 2.1 the translation process from complex constraints to primitive constraints yields the following set of primitive relations:

$$\begin{array}{lllll} x+2 = x_1, & y-3 = x_2, & x_1^2 = x_3, & x_2^2 = x_4, & x_3 + x_4 = 1, \\ x + y = 1, & 2 * x = x_5, & x_5 + 6 = x_6, & 3 * y = x_7, & x_6 \leq x_7 \end{array}$$

As can be noticed, new unknowns have been introduced, that is, x_i, where i =1,7. The intervals initially associated with these new unknowns are [-∞, +∞].

Next, the method tries to narrow the intervals associated with the unknowns using a propagation algorithm, algorithm described in section 2.3. For each primitive constraint there is an associated *contraction operator* [1], which is used to contract the intervals associated with the unknowns that are part of the primitive constraint. The method uses this translation because there are no easy ways to find the solution working directly on

the set of complex constraints, simply because there are no contraction operators available for complex constraints.

Of particular importance are the properties of a constraint contraction operator [2]:

- *Contractance:* the contracted intervals are contained in the original intervals, that is, the operator always shrinks an interval or leaves it unchanged.
- *Correctness:* every solution lies in the contracted intervals.
- *Monotonicity:* the contraction preserves inclusion.
- *Idempotence:* applying the contraction operator more than once in immediate succession yields the same result.

Interval constraints relies on interval arithmetic in the sense that the contraction operators introduced by interval constraints are defined using interval arithmetic operations. Interval arithmetic defines a set of basic operations, such as addition, multiplication, division, exponentiation, etc., on intervals. More specifically, each real number is replaced by an interval that contains it, and the operations are performed on intervals instead on numbers.

The interval constraints method defines an *interval constraint system*, which is used as a basis in the process of solving problems. An interval constraint system has the following components [1]:

- **A set of primitive relations**: These are the primitive constraints into which the complex constraints are translated. For each primitive relation there is a constraint contraction operator defined, which is used to calculate the intervals for the variables involved in the constraint.
- **A set of constraints**: This is the set that results after the complex constraints, which describe the problem, are translated into primitive constraints.
- **A sequence of unknowns**: These are the unknowns that appear in the set of constraints defined above.

- **A state**: This is the set of intervals associated with the unknowns. The state indicates the information about the solution, by specifying a value, as an interval, for each unknown.

Solving a problem translated into an interval constraint system means applying the contraction operators on the set of constrains, in a repeated manner, until this process has no longer effect in modifying the state of the system. This process is named the *constraint propagation* and is presented in section 2.3.

To summarize and bring together all the concepts presented so far in this section a UML conceptual diagram is shown in Figure 2.2. This diagram illustrates a static structure that models all the important concepts encountered in interval constraints and the relations between them.

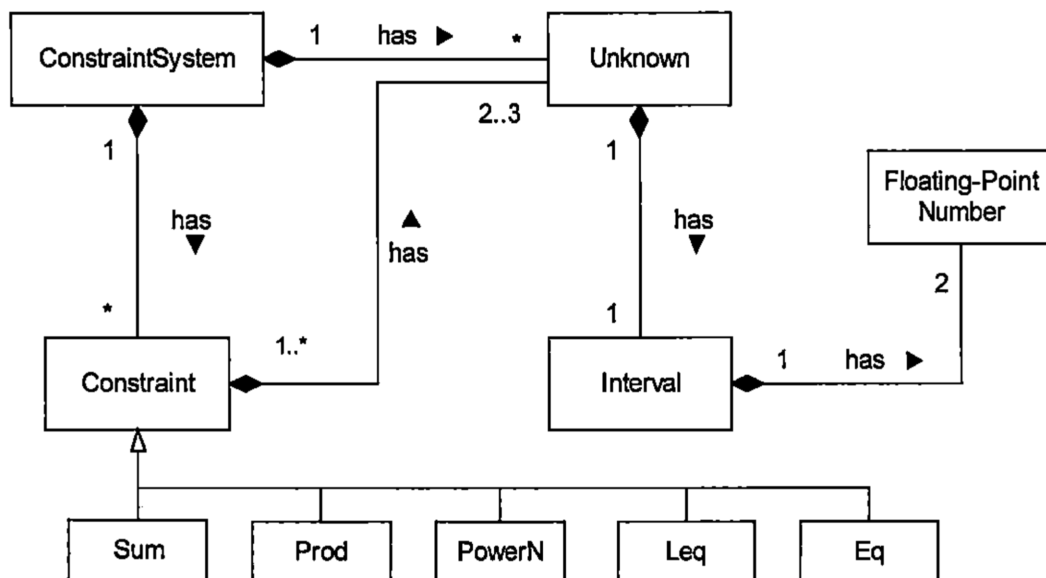


Figure 2.2 Interval constraints conceptual model

The main strength of interval constraints is that it provides certain guarantees with respect to the computed result. It is a method that does not produce an exact result for an unknown, but instead it yields an interval, and guarantees, that if there is any solution for the unknown, then it belongs to the computed interval.

2.3 Constraint propagation

This section describes a constraint propagation algorithm, which represents the key aspect in solving an interval constraint system.

2.3.1 A constraint propagation algorithm

The view of a constraint system to the constraint propagation algorithm is a network like the one presented in Figure 2.3. This is basically a set of constraints that share unknowns. The propagation algorithm proceeds by using the dependencies described by this network. The key point is the fact that constraints share unknowns, and whenever the interval associated with an unknown is changed, as a result of applying a contraction operator, all constraints that share that unknown are scheduled for future application.

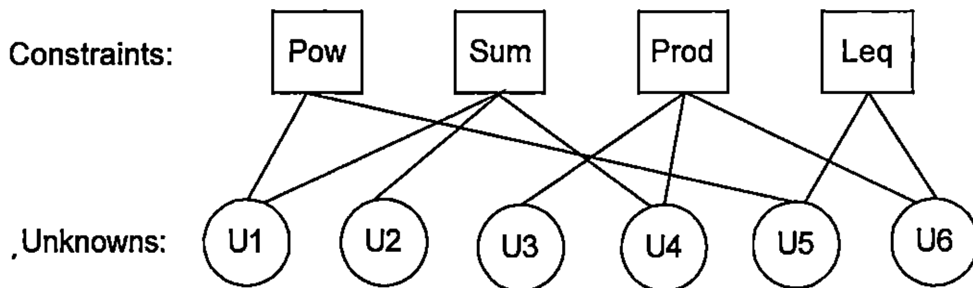


Figure 2.3 Constraints network

A propagation algorithm solves a constraint system by repeatedly applying the contraction operator on each constraint until there are no more changes of the unknowns associated with each constraint. The aim of the constraint propagation is to eliminate as many inconsistent values from the intervals of the constraint system as possible. When a stable point is reached, that is, no more intervals are changed by applying any contraction operator, the propagation algorithm ends, and the result is the maximum information about the solution that the interval constraints method can provide.

It is interesting and important to note that when a stable point is reached, it does not mean that a solution has been found. The only conclusion that can be drawn from the result is that, if the system has one or more solutions, then they are somewhere inside the intervals

that had been calculated during the propagation. However, this should not be considered a disadvantage, but on the contrary it makes the method one that provides certainty with respect to the result yielded. The certainty refers here to the fact that the solutions, if any, are inside the computed intervals. Moreover, combined with a splitting algorithm the computed intervals can be reduced enough, so that the information about the solution is practically sufficient.

A well-known algorithm that is used to perform the constraint propagation process is the Waltz algorithm [2]. The sequential version of it is presented in Figure 2.4.

```
Q ← the set of all constraints;
while Q not empty
     remove constraint C from Q;
     S' ← γC (S);
     if ∃ i : πi(S') is empty then stop. The system is
          inconsistent.
     for each variable ui that occurs in C do
          if  πi(S') != πi(S) then
               for each C' != C in which ui appears do
                    if C' not in Q then add C' to Q;
               endfor
          endif
     endfor
endwhile
```

Figure 2.4 A sequential constraint propagation algorithm

This algorithm starts with a set Q of active constraints, which initially contains the whole set of constraints. During every iteration an active constraint C is extracted from Q and the associated contraction operator γ_C is applied. As a result of applying the contraction operator, some of the unknowns that occur in the constraint might be modified, that is, the associated intervals might be narrowed. This is indicated by the fact that the state of the constraint system is changed from S to S'. A component of this state is indicated by

$\pi_i(S)$, and represents the interval associated with the unknown *i*. If at least one of these components becomes empty, then the system has no solution and the algorithm stops. Whenever an unknown u_i is changed, all the constraints that are inactive, and which have in their set of unknowns the same unknown u_i, that has just been modified, have to be reactivated.

Right after a constraint is applied it becomes inactive, and it is eliminated from the active set of constraints, that is, from *Q*. During this iterative process some constraints can be reactivated. The algorithm proceeds in this way, by applying the contraction operator on the active constraints, in a repeated way, until the set of active constraints becomes empty. This represents the termination condition, and indicates that the state of the system is stable and no further contraction can be done.

Looking at the algorithm, there is no indication regarding the order in which the active constraint should be applied. It is possible that the contraction operators for some constraints be applied more often than for other constraints. This might be either beneficial or inefficient. On the one hand, it might be beneficial when it happens for a constraint for which its associated contraction operator produces a significant contraction. On the other hand, it might be inefficient when some constraints, that, if applied would produce a significant contraction, are waiting very long to be applied for some other constraints that do not produce such a large contraction. Normally, in general it is hard to establish, and dependent on the problem to be solved, which would be the optimal sequence to apply the constraint contraction operators. In a larger context it looks appropriate to extend the base algorithm with a *strategy* that will dictate the sequence of activating constraints. However, it seems presumably that using a first-in first-out strategy for keeping active constraints would lead to an even distribution of activating constraints, which should be acceptable in most cases. This suggests that *Q* should be a FIFO queue, and this is the solution that is adopted in this thesis.

It is worth mentioning some of the properties that the algorithm presented in Figure 2.4 has [2,3]:

- The termination of the algorithm is guaranteed, since the contraction operators always narrow the associated intervals and on a machine the number of computable intervals is finite.
- The algorithm reaches a fixed point, which does not depend on the order in which the constraints are contracted. Of course, as discussed above, the activation order can influence the time required to reach the final result, but not the result.
- The algorithm gives a set of intervals containing all solutions, if any, even when it is stopped earlier that it would, had it terminated naturally. This makes it suitable for situations when there are time limitations, and when it is not desired to let the algorithm run for a very long time. Obviously, in such a case the solution would not be the best one, but it still might be sufficient for a particular problem.
- The algorithm is appropriate for dynamic systems, where a constraint can be added to the systems after the propagation started. The new constraint will have its unknowns updated when its contraction operator will be applied.
- And finally, but most important for us, the algorithm is suitable for parallelism since more constraints can be activated simultaneously.

2.3.2 Example of constraint propagation

Since constraint propagation is the topic of this thesis a detailed example that illustrates the way the propagation process works is presented in this section.

The following constraint system is considered as an example:

Constraints: $x * y = z,\quad x + y = 1,\quad z \leq 1$

Unknowns: $x \in [0, 3],\quad y \in [-\infty, +\infty],\quad z \in [-\infty, +\infty]$.

As can be noticed, the constraints are already in primitive format so no translation is required. There are three unknowns, x, y and z, for which the initial values are also indicated.

Figure 2.5 shows the dependencies that are established between primitive constraints and unknowns. This network of dependencies shows what constraints have to be reactivated when the interval associated with an unknown is changed.

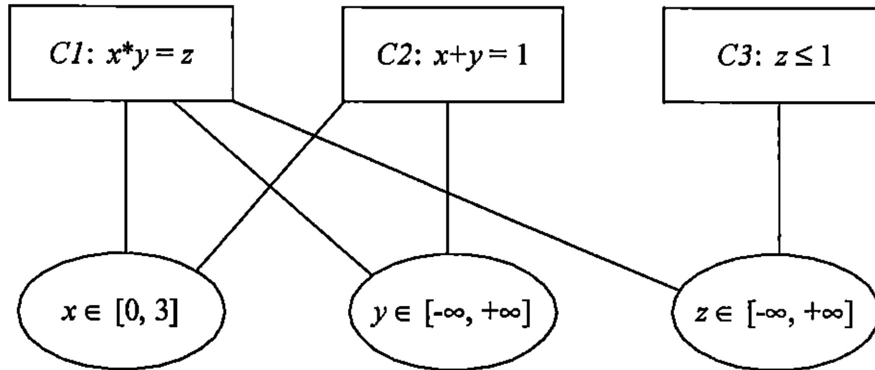


Figure 2.5 Constraints-unknowns dependency network

Initially all constraints are active, therefore, a possible configuration for the queue of active constraints Q is $Q = \{C1, C2, C3\}$. After each contraction, the order in which the constraints are placed/extracted into/from the queue follows a *FIFO* policy.

The constraint propagation algorithm proceeds as follows:

STEP 1: $Q = \{C1, C2, C3\}$, $x \in [0, 3]$, $y \in [-\infty, +\infty]$, $z \in [-\infty, +\infty]$

- *C1* is removed from Q
- Since $y \in [-\infty, +\infty]$ and $z \in [-\infty, +\infty]$ the contraction operator associated with *C1* produces no contraction.

STEP 2: $Q = \{C2, C3\}$, $x \in [0, 3]$, $y \in [-\infty, +\infty]$, $z \in [-\infty, +\infty]$

- *C2* is removed from Q
- Since $(x \geq 0, y = 1 - x \Rightarrow y \leq 1)$ and $(x \leq 0, y = 1 - x \Rightarrow y \leq 1) \Rightarrow y \in [-2,1]$
- Since y has changed *C1* is reactivated.

STEP 3: $Q = \{C3, C1\}$, $x \in [0, 3]$, $y \in [-2,1]$, $z \in [-\infty, +\infty]$

- *C3* is removed from Q
- Since $z \leq 1 \Rightarrow z \in [-\infty, 1]$

- Since z has changed, *C1* should be reactivated. However, since *C1* is already active *C1* is not added one more time into Q.

STEP 4: $Q = \{C1\}$, $x \in [0, 3]$, $y \in [-2,1]$, $z \in [-\infty, 1]$

- *C1* is removed from Q
- Since $x \in [0, 3]$ and $y \in [-2,1] \Rightarrow z \in [-6, 3] \cap [-\infty, 1] = [-6, 1]$
- Since z has changed, *C3* is reactivated.

STEP 5: $Q = \{C3\}$, $x \in [0, 3]$, $y \in [-2,1]$, $z \in [-6, 1]$

- *C3* is removed from Q
- No contraction is produced.
- Since Q is empty the algorithm stops and yields the solution

  $x \in [0, 3]$, $y \in [-2,1]$ and $z \in [-6, 1]$.

It is worth mentioning that the order in which the constraints are initially placed into the queue is arbitrary and does not influence the final result. However, a particular sequence may lead to a faster contraction.

2.4 Interval splitting

Usually, in many complex problems, constraint propagation alone is not sufficient for finding a satisfactory enclosure for all solutions for a constraint system. The reason for this lies in the fact that during the constraint propagation process it is possible to reach a stable point in which applying the associated contraction operator on any constraint produces no change in the state of the system. Moreover, it is possible that at that point the calculated intervals for some of the unknowns are too large, that is, the information about the sought solution is insufficient.

To overcome this problem, the constraint propagation process must be enhanced with interval splitting. *Interval splitting* is a step applied each time the propagation process stops and the calculated intervals for the unknowns are too wide. It consists of dividing

the interval obtained for an unknown, usually in to equal parts, and applying the propagation algorithm to the two newly created constraint systems. This procedure is repeated until the calculated intervals are sufficiently narrow. Therefore, solving a constraint system consists of an interleaved process of propagation and splitting.

Obviously, splitting is applied to those intervals that are too wide. It can be easily noticed that splitting is nothing more than just a backtracking process in which new constraint systems are created from existing ones followed by a trial of solving them.

The main objective of this thesis is a parallel constraint propagation and for this reason interval splitting in not taken into consideration and is left as future work.

3 Parallel Systems

Before presenting the conceptual model for the parallel constraint propagation solution suggested in this thesis, it is worth looking at some terminology used in the context of parallelism. This is necessary because there is a wide diversity regarding this terminology, and it will allow us to eliminate any possible confusion that might appear from using terms that might have a wide variation in meaning.

3.1 Granularity

By definition, *granularity* is the ratio between the amount of computation and the amount of communication within a parallel algorithm implementation [16]. Normally, a parallel algorithm is designed in such a way that sections of it can be run concurrently on different processors. These parts compose the parallel part of the algorithm. Moreover, within the parallel sections, each processor is assigned a specific task. In order to carry out its tasks, a processor needs to communicate and coordinate with processors that execute other tasks. Therefore, executing a parallel algorithm is a permanent alternation of computation-communication. The tasks that are executed might be as simple as performing some arithmetic operations, or it may be larger, such as, a subroutine that engages many operations.

A widely accepted classification for granularity is one with three relative levels, that is, *fine*, *medium*, or *coarse*. In fine grain parallelism, the task may span tens to hundreds of instructions. A vectorizing compiler often exploits this level of parallelism. Medium grain parallelism defines tasks at function or procedure level and may span thousands of instructions. Achieving this level of parallelism often involves both the programmer and

the compiler. Coarse grain parallelism results from the execution of programs that may span tens of thousands of instructions and that do not need to communicate too much while executing their task. This is the case encountered in loosely coupled systems.

Granularity is influenced by three characteristics of the algorithm and the hardware used to run the algorithm:

- **The structure of the problem.** In *data parallel programming*, few operations or possibly a single operation are performed on many pieces of data in parallel. We have the case of a small granularity, also called *fine-grained.* On the other hand, if large subroutines of an algorithm do not share variables, they can all be executed in parallel fashion, with little need for communication. In this case we have a *coarse-grained* situation.
- **The size of the problem.** Increasing the size of a problem but keeping the communication constant increases the granularity also. For instance, let's assume that an algorithm can be used to perform some simple arithmetic operations in parallel. If we use 10 computation nodes to perform 10 operations, each node will require 1 cycle to perform its tasks. However, if the size of the task is increase to 100 operations, while the number of computation nodes is not modified, we have a situation in which the time taken by a computation node to accomplish a task is longer. Therefore, the granularity is increased by a larger task size.
- **The number of computation nodes available.** This characteristic is implied by the previous one. If the number of processors is reduced while holding the size of the problem the same, then the task size increases and the granularity becomes coarser.

The study of granularity is very important when one has to decide which is the best suitable hardware paradigm that should be chosen for a particular problem that must be solved. For instance, SIMD machine are best suitable for very fine-grained kind of problems. A SIMD machine is built to handle very efficiently the communication between the computation nodes, and therefore, there will be little or no penalty with respect to communication. On the contrary, a MIMD machine is not so efficient for fine-

grained problems, since message passing involves more overhead for sending messages between processes. These kinds of architectures are more suitable for coarse-grained parallelism, where communication is reduced and the size of tasks is large.

Having granularity in mind, it is worth looking back at the problem debated in this thesis, and finding out how the constraint propagation can be categorized.

Firstly, a constraint activation takes only something in the order of 10-20 microseconds (Pentium III 733MHz), time dictated by a few floating-point computations, which is less than the time required to send a message over a network in a distributed system, or even than the time required for a process switch. Secondly, a problem that would produce attractive results in terms of performance, if solved using interval constraints, is described through something in the order of ten thousand primitive constraints, and about the same number of variables. At a first sight, this suggests that constrain propagation is more suitable for a fine-grained/tightly-coupled system than for a coarse-grained/loosely-coupled one. However, by partitioning the set of constraints in a way that reduces the communication fraction, a coarse-grained approach can also become feasible. A technique for such a kind of partitioning is presented in section 4.10. Moreover, there are already telephone switches for which the switching time is comparable with the communication time required by a packet to be sent from one switch to a different one. Therefore, it becomes attractive and feasible, especially financially, to have such a kind of a distributed system used to execute the propagation process.

3.2 Models for parallel processing

There is wide variety of models for parallel processing. The aim of this section is not to make an exhaustive presentation of the existing models, but just to give a brief description of some the most widely used ones that have significance in the context of this thesis, and that will allow a better classification of the model proposed in this thesis.

3.2.1 Shared memory paradigms

A shared memory model is characterized by having multiple processors that exchange information by reading and writing a globally shared random access memory space. A system that supports this model is called a *tightly coupled* system or a *multiprocessor* system. This paradigm is very useful for decoupling issue of program control flow from issues of communication, data mapping and synchronization.

The advantage of this model is that it is relatively easy to program since there is no explicit communication between processors, communication being handled via the global memory. However, the shared memory architecture does not scale well, and the main problem occurs when a number of processors attempt to access the global memory at the same time, leading to a bottleneck.

Providing a physical shared memory space, especially for a large parallel system, is quite difficult and not always scalable, but the abstraction of having a shared memory space, even if the implementation it hides is based on a distributed memory, is for many types of applications useful. Examples of such kind of applications are linear equation solving, split and merge sorting, distributed speech recognition systems, computer chess, etc. For many of these applications there is not necessarily a need for a physically shared memory space, but merely a need for logically shared data. Algorithms for such kind of applications have been proved to be much harder implemented efficiently using message passing than shared data, just because with message passing it is difficult to efficiently provide logically shared data. Therefore, regardless of whether the underlying physical memory is distributed, the abstraction of having a shared memory space is a valuable paradigm.

There are various approaches to provide a high-level abstraction of a shared memory space. One way to do this is the so-called *virtual shared memory*. In this model a view of the whole memory is presented as if were shared, but the underlying physical implementation may or may not be. Another way to emulate a shared memory is to extend techniques for cache coherence used in multiprocessors to software coherence [18]. A different model is based on building systems using a set of useful sharing primitives [19].

A well-known example of a shared memory model is Linda [17]. The shared memory in Linda, called *tuple space*, consists of a collection of logical entities called *tuples*. There are two kinds of tuples: process tuples, which are under an active evaluation, and data tuples, which are passive. A data tuple is a collection of fields. Each field has a fixed type associated with it. A field can be *formal* or *actual*. Formals are space holders with types but no values, whereas an actual field has a value of its particular type. The process tuples are all executing simultaneously and are exchanging data by generating, reading, and consuming data tuples. A process tuple that finishes transforms itself into a data tuple. The tuple space is the fundamental medium of communication in Linda. Linda processes communicate and synchronize themselves through the tuple space. Both the sender and the receiver of a tuple interact with the tuple space in an arbitrary order. It is worth noting that Linda is a model not a language, and that a particular realization of it is C-Linda, which is a tool that extends the C language with support for parallel programming.

3.2.2 Distributed memory paradigms

A distributed memory model is composed of a set of processors, each having its own memory as opposed to one large memory pool for all processors, as in the shared memory model. The system that supports such kind of a model is called a *multicomputer*, a *loosely coupled* system or a *distributed* system.

In a distributed memory model, a parallel program is composed of a set of processes that cooperate to solve a common task. Each process owns a local collection of data, which cannot be directly accessed by any other process. A key point of this model is that processes communicate and synchronize by sending each other messages. A secondary aspect of this approach is that because there is no sharing of variables, the problem of concurrent writes to the same memory location is eliminated without explicit need of synchronization. Due to the way these models operate, by exchanging messages, they are also known as *message passing* systems.

Although, it has the disadvantage of requiring explicit communication, this model has a very important advantage over the shared memory model, and it consists in greater scalability.

The model is based on the idea of several sequential processes that are running in parallel to each other. It is worth to note that in this context it is not important if these processes are really running on distinct processors, or if they are scheduled on one processor by a multitasking operating system or kernel. In this light, it is easy to realize that a distributed-memory model can be used to run a parallel application even on a single processor.

A very delicate and important issue with distributed-memory models is the correct detection of termination of the parallel program. While the parallel program is running, processes communicate by exchanging messages, and normally, to assure scalability for instance, there is no central controller to monitor the state of all processes and decide on global termination. Thus, distributed termination must be implemented as a cooperation of all processes that are engaged in solving the problem. It is worth mentioning that termination detection in a parallel program is not a trivial problem, and it is essential for the correctness of the program as a whole.

An influential role in the distributed-memory paradigm has been played by a model developed by Hoare [21] called *Communicating Sequential Processes*. According to this model, a parallel program consists of a network of processes, each one with its own local private data. These processes are unnamed and can communicate between them through communication channels using explicit message passing. The main features of the Communicating Sequential Processes model are communication management by means of input and output commands and channels, exploitation of parallelism by means of parallel commands, and deterministic execution by means of guarded commands.

A well-known realization of the Communicating Sequential Processes model is the Occam language [20] designed by David May at Inmos. Occam was designed for transputers and can be used to easily describe concurrent processes that communicate via one-way channels. A transputer is a single chip that has a single processor, its own memory, and four communication links. This allows transputers to be networked directly and used very efficiently for massively parallel programs. The main design goal of Occam was to provide a language that could be directly used for a network of processing

elements, and in many respects Occam is intended as an assembly language for such systems.

There are many other distributed-memory-based languages, toolkits or frameworks that have been developed. Some of them are totally new languages, some of them are just extensions of some existing sequential languages. In addition to the above-mentioned Occam, another example is High Performance Fortran (HPF).

An importance place is occupied by the Message Passing Interface (MPI) [22,23], which is an effort to provide a general, portable message passing interface standard across the whole parallel processing community. In principle an algorithm for a particular paradigm is portable on any platform that supports that paradigm. However, when the source code is to be ported many problems appear, and the possibility of being able to do this easily is a desirable goal. One primary benefit of such an approach would be the protection of the investment in the code written for a particular platform. Another one would be the flexibility of being able to initially develop the code on one widely, available architecture (i.e. a network of workstations), before running it on a very particular, hard-to-access, target hardware. As a response for this need MPI was developed. Its primary goal is to provide source-code portability of message passing programs written in C or Fortran across a variety of architectures. It also offers support for heterogeneous parallel architectures. More specifically, MPI comprises a library. An MPI application consists of a collection of processes that executes programs written in a standard sequential language augmented with calls to an MPI library of functions for sending and receiving messages. The MPI routines provide the programmer with a consistent interface across a wide variety of different platforms.

3.3 Coordination languages

A widely accepted way to describe a parallel program is as a coordination problem between a set of separated activities. There are two main aspects in writing parallel programs using this kind of coordination approach. Firstly, the set of activities that are meant to be executed in parallel must be written. This can be accomplished using a

computation language, such as C, Fortran, etc. Secondly, these activities must be combined together, and the communication and synchronization issues regarding them must be specified. This second set of requirements is provided by a coordination language.

In a broader context, according to David Gelertner and Nicholas Carriero [17], a complete programming model can always be thought as made of two separate, orthogonal components: the computation model and the coordination model. While the first is meant to build a sequential computational activity, the coordination model is the "glue" that binds separate activities into an ensemble. A *coordination language* embodies a coordination model, and it provides operations to create computational activities and to support communication and synchronization among them. Examples of coordination languages are COOL [24] and Linda [17].

Of particular importance is that a coordination language is not an entire environment for parallel computation. It is just a new language that embodies the coordination model only. Theoretically, a particular compiler allows the combination of a coordination language with any computing language.

3.3.1 The COOL coordination language

The solution proposed in this thesis uses as a coordination language COOL, and as a computation language C++. COOL [24, 25, 26] is an object coordination language for Crisp. Crisp is a highly efficient runtime system for building distributed reactive control systems.

COOL has a simple semantic for specifying distributed, concurrent, communicating state machines. It provides a high-level descriptive language in which it is easy to describe active objects interfaces, behaviours, and their coordination. An object in COOL, called *actor*, is a single unit of concurrency that has its own thread of control. Actors communicate with each other through an order-preserving, asynchronous message passing mechanism. Atomicity is also provided, which means that a message received by an actor will run to completion without interruption. Since Crisp/COOL applications are mainly meant to communicate over an environment with a low probability of message

loss, such as ATM networks, there is no guarantee of message delivery. However, the language provides timers, mechanism that can be used to deal with the issue of loss of messages. Since, COOL has also minimal support for data structures and control structures it is possible to compile COOL applications into C/C++ very efficiently.

3.4 Termination detection

Termination detection is a very important and complicated problem in a distributed-memory environment, especially because there is no shared data to easily signal this condition. This section indicates some of the existing algorithms on this topic.

In [30], there is a description for an algorithm for detecting termination for a diffusing algorithm. A *diffusing* algorithm is one in which all activity begins at one point and "diffuses" through the network via messages. As it will be described in chapter 4, this resembles the problem described in this thesis, that is, propagation begins and ends at the same agent, that is, the Monitor agent. The algorithm is called *Dijkstra-Scholten*, and is based on augmenting the underlying algorithm A with the construction and maintenance of a spanning tree of the graph processes that are currently involved in the computation. It is a very interesting and scalable approach. The only drawback that one can see is that for a system where there are many messages sent its cost is quite expensive. The total number of messages sent in A augmented with the detection algorithm is always $2m$, where m is the number of messages sent in A.

A token-based termination detection algorithm proposed by Safra is presented in [34, 35]. It assumes all processes connected in a ring and it allows an initially designated process to conclude that termination has been reached. This algorithm is a generalization of the *Dijkstra-Feijen-van Gasteren* [34] termination detection algorithm, which assumes a synchronous message-passing model, to an asynchronous message-passing model.

Mattern proposed an algorithm that detects termination very fast based on a *credit-recovery* scheme [34]. The algorithm uses an initiating process that assigns each message and each process a credit value, which is always between 0 and 1. Whenever a process sends a message, it assigns it half of the credit that it possesses. Whenever a process

receives a message, it collects the credit carried by the message. When a process become passive, it sends its credit to the initiator. The algorithm terminates when the initiator has recovered its initial credit of 1.

All algorithms indicated above, and also the one used in this thesis, are distributed algorithms for detecting termination, using exactly one centralized initiating process that initiates the distributed computation and concludes termination. In such a system, normally, there is a need for one more phase after the termination is detected, phase in which the other processes are announced about the termination. Configurations that use an arbitrary number of active processes in the initial configuration are called decentralized, and normally they require a decentralized termination detection algorithm. Such an algorithm does not require an additional phase in which the termination is announced, since each node can individually detect termination. However, these kinds of algorithms are more complex and computationally more expensive. An example of such an algorithm is the *Shavit-Francez* algorithm [34], which is a generalization of the *Dijkstra-Scholten* algorithm [30].

A comprehensive collection of algorithms for termination detection is presented in [34].

4 Parallel Constraint Propagation

This chapter presents the conceptual model of the parallel constraint propagation architecture proposed in this thesis. The proposed architecture is based on coordination between a set of cooperative communicating agents. More specifically, the constraint propagation problem is described as a coordination of a set of cooperative agents, each agent having a specific role in the process of solving the problem. Each agent communicates and exchanges information with the other agents, so that the entire problem is solved as a cooperative process.

The proposed model suggests a modular approach based on agents that communicate through a message passing mechanism. The *agent* term in this context is used as a part of the conceptual model, and denotes an active entity that has a modular unit of execution, encapsulates some private local data, has its own thread of control, and can execute concurrently and asynchronously with other agents.

A concrete implementation of the conceptual model presented in this chapter is described in the next chapter. The implementation uses C++ as computation language and COOL as a coordination language. Even if the implementation was tested on a distributed system of workstations, the model is not limited just to that target platform.

4.1 Suitability for parallelism

Analyzing the way a constraint contraction operators works [2] there are two properties that are important from the parallelism point of view, that is, monotonicity and contractance (the complete set of properties that characterizes a contraction operator have been given in section 2.2). As a reminder, the contractance property states that the

contracted intervals are contained in the original intervals, whereas the monotonicity property states that contraction operator preserves interval inclusion.

As a consequence of these properties, the result for an interval will be always the same and virtually there is no restriction with respect to the order in which the constraints should be activated while performing constraint propagation. This results from the fact that a contraction operator always tries to narrow the intervals that it operates on by performing an intersection between the new calculated value for an interval and the old value for the same interval. Therefore, this is a guarantee that an interval will never grow, but always shrink. This implies that the final result for an interval will be always the same, even if the paths that lead to it might be different. Of course, it is possible that activating the set of constraints in a particular sequence will lead fastest to the final intervals, but this is just a performance matter, because the final intervals will be the same regardless of the activation sequence that was followed. These remarks support the idea that it is possible to activate constraints in parallel, and by this to speed up the constraint propagation process.

In addition to the obvious suitability for parallelism, which result from the way the contraction process is performed, there is also a very simple but powerful motivation for using parallelism in the contraction process, which is, the worthwhile speed-up that parallelism provides. As mentioned earlier, the gain in performance provided by a parallel approach, regardless whether a multiprocessor or a distributed system is used, has already been proven in many applications.

4.2 Proposed architecture

A simple solution for a distributed constraint propagation algorithm is to have several computation nodes executing in parallel the same sequential constraint propagation algorithm, each computation node operating on a subset of the whole set of constraints. In other words, the whole constraint system is partitioned in several subsystems, each subsystem being solved by one computation node. Of course, to be able to solve the

problem, the nodes will have to communicate. A high level view of such a solution is shown in Figure 4.1.

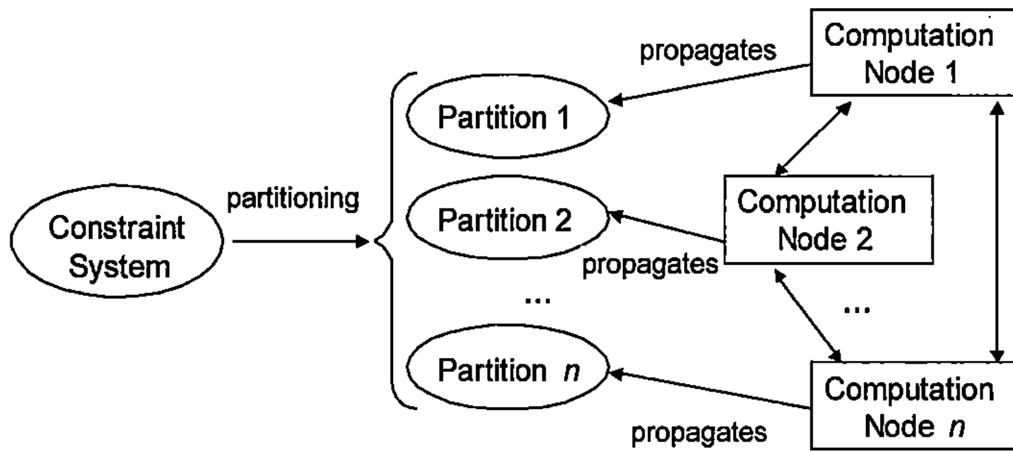


Figure 4.1 Parallel constraint propagation - high-level architecture

This approach is very attractive, firstly due to its simplicity, and secondly due to its scalability. A desirable feature when designing a parallel algorithm is scalability, which is translated in the fact that adding new computation nodes to the system used to solve the problem does not require modifying the algorithm, but just increases the computation power. The architecture suggested above seems to be a scalable one. If more computation power is needed, more computation nodes can be added, the problem can be partitioned into smaller subsystems, and presumably, a speed-up is achieved.

Based on the above remarks, the architecture proposed in this thesis has a number *n* of agents, called *Workers*, one for each subsystem of the whole constraint system. Naturally, there should be one Worker per computation node, but this is not a necessity.

This approach assumes no central unit of coordination, which makes it even more attractive from the scalability point of view. On the contrary, the control is distributed among all Workers that take part in solving the problem in a cooperative way.

The computation process has to be started somewhere. The key idea on which the algorithm is based is the following: the computation process starts and stops at the same point. In other words, there is an agent from where the computation process is initiated,

and where the computation process ends. To be more specific, there is a particular agent, called *Monitor*, which starts the computation process.

The Monitor has the job to partition the constraint system in subsystems and to spread the subsystems between all computation nodes. To preserve modularity and provide flexibility, the partitioning process is not performed directly by the Monitor. Instead, the Monitor creates a new agent, called *ConstraintSystem*, which encapsulates a particular constraint system that has to be solved, and which is parameterized with the number of partitions in which the systems must be split. The ConstraintSystem agent is the one that actually partitions the constraint system, distributes it to the Workers, and collects the results at the end.

The Monitor has also the duty to detect the termination of the constraint propagation process and to inform the ConstraintSystem to collect the results.

Apparently, the Monitor might seem like a central entity that breaks the scalability of the algorithm proposed here. However, this is not the case, because the Monitor has only minor needs with respect to the computation requirements.

Figure 4.2 shows a UML static diagram that illustrates the conceptual model of a parallel propagation process presented in this section. The diagram shows a static view of the presented model and contains the most relevant concepts and the associations between them.

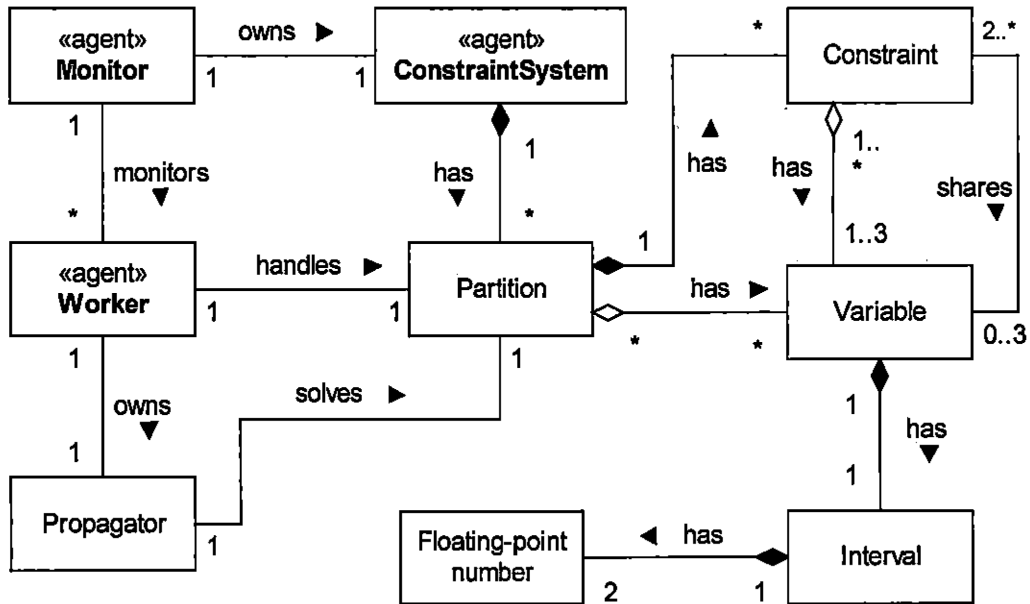


Figure 4.2 Conceptual model for a parallel constraint propagation

As can be noticed, the conceptual model presented in Figure 4.2 extends the one from Figure 2.2, which gives a view of interval constraints. The most important new concept introduced is the notion of *partition*, and it is brought in by parallelism. New concepts are also the *propagator*, which practically represents the propagation, and the agents used to cooperatively perform it.

4.3 Sharing variables between constraints

In the context of interval constraints, a *variable* is regarded as an unknown in the implementation domain, and represents an interval containing all values of the unknown that might occur in the solution.

When a constraint system is partitioned in subsystems, two kinds of variables appear. On one hand, there are *local variables*, that is, variables that are local to a partition and that are not present in any other subsystem. On the other hand, there are variables that are shared across subsystems, called *shared variables.* The shared variable concept defined here, and used in this thesis, should not be confused with the meaning that a shared variable has in the shared memory model briefly described in section 3.2.1.

To illustrate the distinction between shared and local variables the following simplistic constraint system is considered:

$$\begin{cases}(x+1)^2+3=y\\ y^3+2*z=7\end{cases}$$

The process of translating these complex relations into primitive relations yields the following set of primitive constraints:

$$x+1=x_1, \qquad x_1^2=x_2, \qquad x_2+3=y,$$
$$y^3=x_3, \qquad 2*z=x_4, \qquad x_3+x_4=7$$

If two partitions are considered, the first one consisting of the first three primitive constraints, and the second one consisting of the last three primitive constraints, the following types of variables can be distinguished:

x, x_1, x_2 :	local to the first partition
z, x_3, x_4 :	local to the second partition
y :	shared between partitions

A new issue that appears in parallel constraint propagation arises from the fact that constraints share variables. Each variable has an associated interval. This implies that each time a Worker performs a contraction on one of the variables associated with its set of constraints, the other Workers, that operate on constraints which share the same variable, have to be informed about the new value for the changed variable. To solve this problem, each Worker, for all its variables that are shared with other Workers, has to keep a list that indicates which Workers have to be informed when a variables is changed. This approach eliminates the need for a central coordinator for Workers by distributing the coordination at Worker level. The only action that a Worker has to perform when a contraction takes place is to inform the other Workers that a new value has been found for a specific variable.

It is possible for a Worker to perform a contraction using stale values for some variables involved in a constraint. This does not destroy the correctness of the final result, because only smaller intervals are taken into consideration by a Worker when it receives new values for some of its variables. A Worker achieves this by doing an intersection between

the current interval that it has for a variable and the new interval that it receives for that variable. Eventually, the node that performs contractions using stale values for some of its variables will receive the updated values for those variables and will update its state.

Another aspect that results from sharing variables among several Workers is the necessity of uniquely identifying each variable involved in the constraint propagation process. This issue has to be taken into account too, and appears when a Worker changes a variable and the other Workers have to identify the specific variable for which they receive the update.

4.4 Constraint propagation initialization

The parallel constraint propagation is started with the creation of the Monitor and Workers agents. The Monitor is parameterized with the number of Workers that are to be used to solve the problem. This is the only parameterized information that the Monitor has to know.

In order to decouple the creation of the Monitor from Workers, the following protocol is used. After the Monitor is created it advertises its presence by registering its identity and location with a naming service (COOL/Crisp provides a simple lookup/subscribe naming service). The same naming service is used by each Worker to subscribe to the Monitor as a Worker. More specifically, each Worker, after it is created and started, periodically sends a lookup message to the naming service until it acquires a reference to the Monitor. When a reference is acquired, the Worker subscribes to the Monitor by indicating its location and availability for work. When the Monitor has received a number of subscriptions equal to the number of Workers that it requires, this initial phase is completed. At this point a network of connected agents in a star topology with the Monitor in the centre exists.

This approach is very flexible and does not impose any restrictions regarding the order in which the Monitor and Workers are created. It is possible to create them sequentially or in parallel in an arbitrary order.

In order to distribute the constraint propagation process among several Workers, the constraint system has to be partitioned in a number of subsystems, equal to the number of

Workers. Each system contains a subset of the whole set of constraints and the associated unknowns. This partitioning process is the next step that has to be performed. To accomplish this, the Monitor creates a ConstraintSystem agent that has the knowledge about the constraint system that has to be solved. The ConstraintSystem agent received from the Monitor the number of partitions in which the problem has to be divided and the identity of the Workers that are to be used to solve the problem. The ConstraintSystem creates the partitions, and then, distributes them to the available Workers. After the distribution process is finished everything is ready for the actual constraint propagation process.

4.5 Termination detection

Termination detection in a distributed-memory environment is a complicated problem, since there is no shared data to easily signal this condition, and moreover, since normally there is no central unit of coordination, but on the contrary, the coordination is distributed among all participating entities. This problem is faced in the model proposed in this thesis too. Therefore, an important issue regarding the parallel version of the constraint propagation algorithm is the answer to the question "*How does one know when the algorithm terminates?*", since there is no central coordinator for the algorithm. Some well-known algorithms for detecting termination are briefly indicated in section 3.4.

Referring back to the sequential constraint propagation algorithm shown in Figure 2.4, termination is reached when the set of active constraints has become empty (it is important to emphasize that there is a **single** set of constraints in this case). This corresponds to the situation when applying the contraction operator on any constraint does not produce any change to the set of unknowns. Moreover, since there is only one single unit of execution, the execution of the propagation algorithm consists of only one continuous phase.

However, the parallel case is more complex than the sequential one. During the parallel constraint propagation, a Worker can be in one of two states, that is, it can be either active or idle. A Worker is *active* when its set of active constraints is not empty, that is, it

has constraints to contract. When the active set of constraints becomes empty, a Worker becomes *idle*. It is possible that during the parallel constraint propagation a Worker contracts all its constraints and becomes idle. However, this does not mean that the propagation is finished, since it is possible that other Workers are still active contracting their sets of constraints. Moreover, a Worker that is idle can become active again when it receives new values for some of the variables that it handles. As opposed to the sequential case, during the parallel constraint propagation, a Worker can change its state dynamically, several times. Therefore, the parallel constraint propagation can be considered terminated only when all Workers are idle.

A simple way to detect termination is to have each Worker, which participates in the collective computation, indicate its state to a monitoring entity. Since during the propagation process the Monitor is not engaged in any other activity, it can handle this monitoring activity.

The simplest way to detect termination is to have each Worker send periodically an "I am active" message to the Monitor as long as it is active. The Monitor will just have to keep a timer for detecting termination that is reset any time a message "I am active" is received. When all Workers become idle, no "I am active" messages will be received by the Monitor anymore, and the timer for detecting termination will trigger indicating that the propagation is finished. Even if this approach is attractive, due to its simplicity, it is not very efficient since involves many redundant "I am active" messages sent by each Worker to the Monitor. Although, by choosing a proper interval for sending the "I am active" message, it seems presumable that the method is valuable and does not affect too much the scalability of the system. However, in this thesis, a different, more efficient method is used for detecting termination.

Any time when a Worker changes its status, it indicates this dynamically by sending a message to the Monitor signalling its current status. When a Worker becomes active, it sends an "I am active" message, and when it becomes idle, it sends an "I am idle" message. The Monitor keeps track of the status of each Worker, and whenever it receives a message from a Worker, indicating a change in its status, it sets a boolean value accordingly. Moreover, if an "I am idle" message is received, it checks to see if all

Workers are idle, and if they all are, then, and only then, it sets up a timer, that if it succeeds to trigger, indicates to the Monitor that the parallel constraint propagation is terminated. The timer that the Monitor sets can only be reset by an "I am active" message from a Worker. Therefore, the Monitor needs to cancel the timer whenever an "I am active" message is received.

The use of the timer by the Monitor might seem unnecessary. However, this is not the case, since without it, the Monitor might detect a false termination. It is possible to have situations in which all Workers seem idle to the Monitor for a short period of time, but, at the same time, a message, that will set a Worker active again, is travelling between two Workers. To be more specific an example is given. Let's consider a situation in which all Workers are idle except Worker *W1*. *W1* has only one more constraint to contract. It contracts it, and as a result, the value of a shared variable is changed, change that requires that an update message be sent to the idle Worker *W2*. *W1* sends this update message, but it also sends the "I am idle" message to the Monitor. As soon as *W2* receives the update message, it becomes active, and it sends an "I am active" message to the Monitor. However, since the time required by a message to travel between agents has a finite variable non deterministic value, it is not possible to say which message will be received first by the Monitor. If the "I am idle" message from *W1* is received first, then the Monitor concludes that all Workers are idle, and, without the timer, the propagation would be wrongly considered terminated. Therefore, special care must be taken to avoid this situation, and the timer is a possible solution.

In a concrete implementation, the interval of time used by the Monitor to set up the timer can be empirically determined, and depends on the time required by messages to travel between agents. Normally, this time has to be sufficiently long so that no false termination can be detected.

In a real problem there are a large number of constraint and the Workers are normally most of the time active. Presumable, their status does not change very often. Therefore, the number of messages sent by Workers to the Monitor during the parallel constraint propagation is small and the scalability is not affected.

After the Monitor concludes that the propagation is terminated the solution has to be collected. The Monitor sends a message to the ConstraintSystem agent indicating that the results must be collected. The ConstraintSystem collects the results and indicates the accomplishment of this task to the Monitor. At this point the Monitor can dismiss all Workers and the constraint propagation is considered completely finished.

Regarding the complexity, the additional number of messages sent by the termination detection algorithm used in this thesis is $m+2k$, where k is the number of times all nodes change status from idle to active, or active to idle, and m is the total number of messages that would be sent in the system without the termination detection algorithm. Normally, for a system in which many messages are sent, this is a less costly than the *Dijkstra-Scholten* algorithm [30].

It is possible that a constraint system has no solution case in which the constraint propagation fails. This situation is described in section 4.11.

4.6 Communication between agents

The model described in this chapter assumes a message passing mechanism. The agents involved in the constraint propagation process have to exchange information. The communication between them is performed using messages. This is the only way through which the agents can share data. Messages have to be exchanged between all agents, that is, Monitor, ConstraintSystem, and Workers.

There are different ways in which messages can be delivered in a message passing system. Message delivery can be reliable or unreliable. In a reliable scenario there is guaranteed that any sent message reaches its destination. On the contrary, in an unreliable scenario messages can be lost, and if reliability is required the application itself has to use a proper protocol to achieve reliability. This can be accomplished using timers and retransmissions. Moreover, messages can be delivered in the order in which they were sent or not. This is also a significant aspect since depending on the application it might be influential for the correctness of the final result.

The positive aspect for a parallel constraint propagation is that there is a large tolerance regarding both these characteristics of a message passing system. Order preserving is not important because an interval associated with an unknown never grows but always shrinks. A Worker that receives an update message for a variable will only update its value if the interval is narrowed. Therefore, if a larger interval is received, the message that carries it will not have a negative influence on the final result. The message is just not useful and it is ignored. For the same reason, if some update messages are lost, the final result is also not compromised. In this case, it is possible for the propagation to stop with larger intervals, but this is not a problem since by just starting the propagation again, from the point where it stopped, eventually the same final intervals will be calculated.

There are situations where message delivery must be guaranteed, such as system initialization or termination detection. For those cases proper care must be taken so that messages are reliably delivered.

A concrete implementation that follows the model described in this chapter is presented in chapter 5, and uses the message passing mechanism provided by COOL. Since COOL provides an unreliable order-preserving asynchronous message passing mechanism, special care has been taken in all places where messages have to be reliably delivered to destination. Timers, facility offered by COOL, have been used for retransmissions in order to provide guarantee of message delivery.

4.7 The Monitor agent

This section describes the role of the Monitor agent in the parallel constraint propagation process.

4.7.1 The activity of the Monitor

The Monitor has three important functions. First of all, it has to initialize the whole system for the constraint propagation process. Secondly, it has to monitor the status of the Workers in order to detect the termination of the constraint propagation process. Finally, its last function is to request the ConstraintSystem agent to collect the results after the

constraint propagation process terminates and to dismiss the Worker agents. Figure 4.3 shows a state-chart that depicts the behaviour of the Monitor agent.

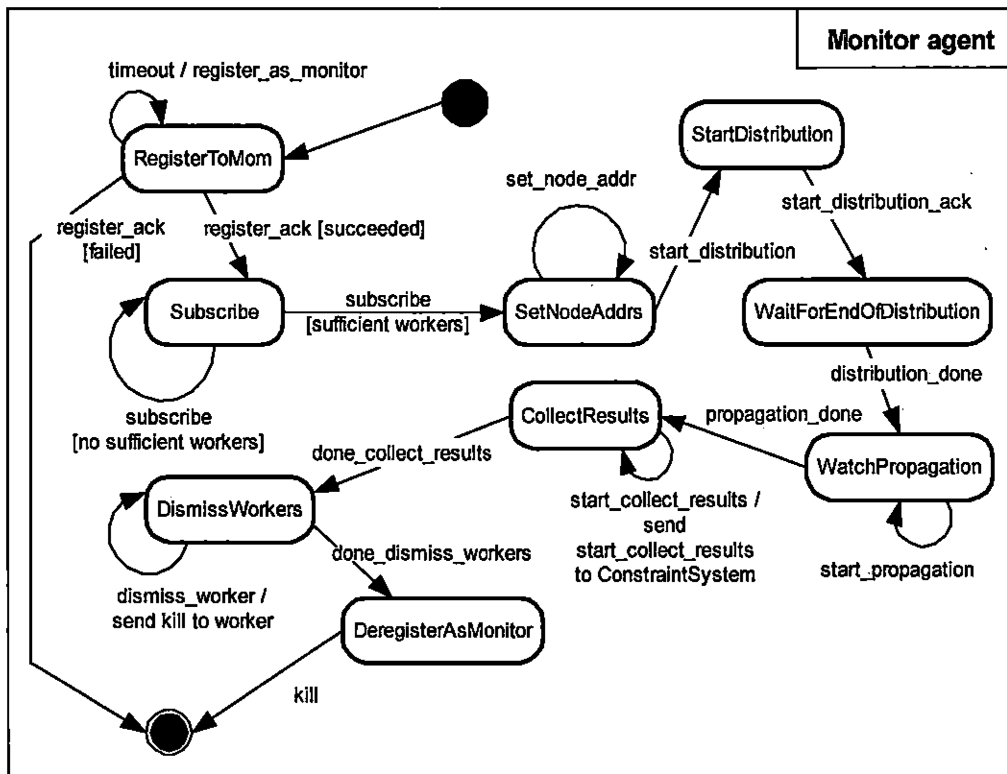


Figure 4.3 Monitor agent state-chart

As presented in section 4.4, firstly, an agent has to become Monitor by registering itself with the naming service provided by COOL. This happens while the agent is in the *ResisterToMom* state. Secondly, the Monitor has to wait in the *Subscribe* state until a sufficient number of Workers have subscribed to it as Workers. Since the Worker agents need to communicate between them, each one needs to know the identity of the other ones. While in *SetNodeAddrs* state, the Monitor makes a mapping between a unique identification assigned to each Worker and the location (address) of that Worker, and also distributes these mappings to all participating Worker agents. Next, the Monitor creates the ConstraintSystem agent and sends it the number of partitions in which the constraint system has to be partitioned and the identity of the Workers that will be used to solve it. After this step is completed, the Monitor waits in the *WaitForEndOfDistribution* state

until the ConstraintSystem agent notifies it that the constraint system has been partitioned and distributed to the Workers.

At the end of the initialization phase, after each Worker has the data set that it has to operate on, the Monitor has to indicate to each Worker to start the constraint propagation process. At this point, the Monitor sends a message to each Worker indicating to start the propagation. During the constraint propagation, the Monitor is in *WatchPropagation* state and it has a passive role, that is, it monitors the status of its Workers with the aim to detect the termination of the propagation.

The monitoring function, which the Monitor has to perform, starts after the initialization phase is finished. The Monitor has to keep track of the status of all Workers. When all Workers are idle, the termination condition is satisfied, and the Monitor can conclude that the constraint propagation process is finished. Termination detection is described in section 4.5. When this happens, the Monitor sends a message to the ConstraintSystem to collect the results, and waits in the *CollectResults* state until all results are collected. Then, it can dismiss the Workers by sending dismiss messages to all of them. This ends the parallel constraint propagation. As the last step the Monitor deregisters itself as being Monitor.

4.7.2 Data handled by the Monitor

The Monitor has a simple structure. It only needs to keep a list that indicates the identity and the status of each Worker.

The identity of the Workers that participate at the constraint contraction process is collected during the initialisation phase. The status of each Worker is required by the Monitor for detecting the termination condition.

4.7.3 Messages handled by the Monitor

The communication between the Monitor and the other agents is performed using an asynchronous message passing mechanism. This section presents the messages handled by the Monitor. In addition to receiving messages, the Monitor also sends messages to the

other agents. These messages are described in the sections where the corresponding agents are presented.

The most representative messages handled by the Monitor are the following ones:

- `subscribe()`: This message is sent by a Worker during the initialization phase. The Monitor records the identity of the sending Worker. The Worker that sends this message will be used during the propagation phase to solve a subsystem of constraints.
- `set_node_status(status)`: This message is used by a Worker to indicate its status to the Monitor, that is, idle or active. When the Monitor receives this message it updates its monitoring information and checks to see if the termination condition is met.
- `failed()`: This message is sent by a Worker to indicate that the constraint propagation failed. When received, the Monitor sends `kill()` messages to all agents. This terminates the constraint propagation.

4.8 The Worker agent

This section describes the role of a Worker agent in the system used to perform the constraint propagation.

4.8.1 The activity of a Worker

One of the main goals of this work is to provide a scalable solution for the constraint propagation process. This implies that there should be no central unit of coordination, because such a central point would introduce a bottleneck when the system grows very large. On the contrary, a homogenous solution in which the coordination is distributed among all computation nodes that participate at the computation process, and in which each computation node has the same role, does not introduce such kind of scalability problems.

The solution proposed here provides such kind of scalability due to the fact that there is no central coordinator, but on the contrary, the coordination is distributed among all Workers agents. A Worker is a modular unit that encapsulates all its internal state. This modular approach allows an easy expansion of the system, by simply adding more Workers agents when more computation power is required.

The function of the Worker agent is to run the constraint propagation algorithm on the subsystem of constraints that has been assigned to it by the ConstraintSystem agent. A state-chart showing the behaviour of a Worker agent is indicated in Figure 4.4.

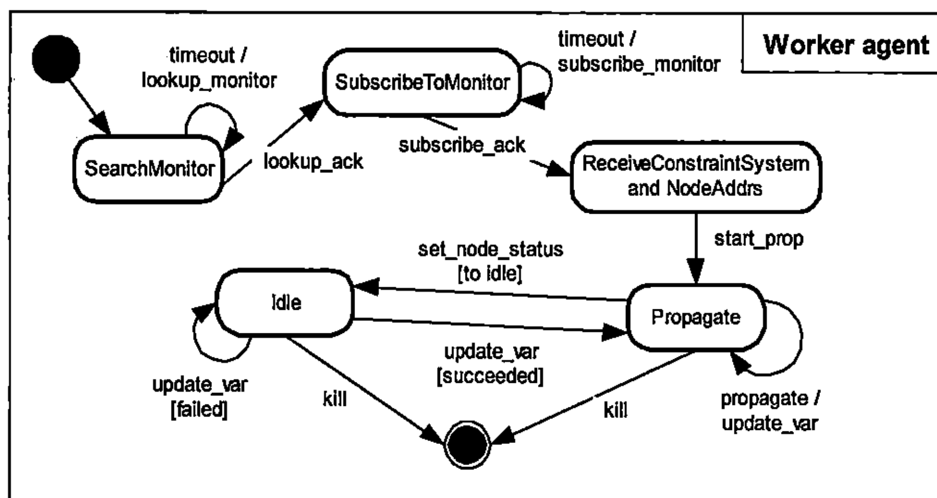


Figure 4.4 Worker agent state-chart

After a Worker agent is created and started, it has to subscribe itself to the Monitor as a worker. Since the creation of the Monitor and Worker is totally decoupled, a Worker uses a retry policy until it succeeds to subscribe itself to the Monitor. Right after it starts, a Worker enters into a retry loop in which periodically contacts the naming service provided by COOL and asks for a reference to the Monitor. This is accomplished while the Worker is in the *SearchMonitor* state. After a reference to the Monitor is acquired a Worker subscribes itself to the Monitor and enters into the *ReceiveConstraintSystem* state in which it waits to receive the constraint subsystem that it will have to solve. While in this state the Worker also receives the set of mappings that map the unique identification of each Worker to its location. A Worker stays in this state until it receives a message from the Monitor to start the propagation process.

When the message to start propagation is received, a Worker concludes that the initialization phase has finished and starts the propagation. The main activity of a Worker agent is to apply the contraction operators on its constraints and to inform other Workers about changes in its set of variables. The propagation process is performed while the agent is in the *Propagate* state, that is, the Worker is active applying contraction operators. Any time a Worker changes its status, that is, becomes idle or active, it has to inform the Monitor about its new status. During the propagation the state of a Worker can change several times between *Idle* and *Propagate*. Eventually, when the propagation is terminated, a Worker sits idle and waits in the *Idle* state. Finally, it may receive some requests for the values of some of the variables that it has, and in the end, it receives a dismiss message from the Monitor. This dismiss message determines the receiving Worker to terminate itself.

As it was mentioned in the previous paragraph, throughout the propagation process a Worker's main activity is to contract its active constraints. During the actual process of constraint propagation, any time a new value for a variable is found, the other Workers that share that variable have to be informed. At a first sight, it would seem naturally that as soon as a variable is changed the other Workers that share it to be informed. However, this is not always the best choice from the point view of the overall performance delivered by the system. Another alternative would be to perform several contraction and only at the end to send update messages for the modified variables to the corresponding Workers. This approach might reduce the overall traffic between Workers, on the one hand, by carrying updates for several variables in the same message, and on the other hand, by eliminating some redundant updates by just sending the most recent one. The appropriate choice depends on the target system for which a concrete implementation is made. For a multiprocessor system, in which the penalty introduced by the communication is comparable with the computation time, the former approach is appropriate, since any update can speed-up the overall propagation. On the contrary, for a distributed system, the ratio of communication is desirable to be reduced as much as possible since it is costly. Moreover, for this case, sending too many update messages might cancel the speed-up that is expected from performing propagation in parallel. Therefore, the latter approach is more appropriate for this case.

4.8.2 Data handled by a Worker

Each Worker runs the constraint propagation algorithm on a subsystem of constraints. In order for a Worker to carry out its activity it needs the following categories of information:

- The set of constraints on which it has to run the constraint propagation algorithm, that is, the subsystem that it has to solve.
- For each shared variable involved in its set of constraints a list of Workers which have to be informed at any time the value of the variable is changed. This list is used to communicate to the other Workers the new value for the shared variable.
- For each variable involved in its set of constraints an initial value.
- A unique identification, used to indicate the status of the Worker, that is, idle or active. This status information is required by the Monitor, which keeps track of the status of all Workers, in order to detect the termination condition of the constraint propagation.
- The identity of the Monitor, which has to be informed any time the status of the Worker changes.

During the initialization process, a Worker receives a set of constraints that it has to operate on. For each constraint it receives the constraint type and the IDs of the variables involved in it. Based on this information, the Worker has to construct the actual constraint objects and to add them into its internal set of constraints. Therefore, the first data structure that a Worker has to keep it a set of constraints.

Based on the set of constraints and their corresponding variables, a Worker has to build, for each local variable, a set of constraints that share that variable. This set is used whenever a value for a local variable is modified to find out what constraints have to be reactivated.

The third type of information is the set of active constraints. It seems appropriately to keep this set of constraints as a queue that follows a FIFO policy. The motivation for

using a FIFO is described in section 2.3.1, but essentially, this will assure an even distribution of activating constraints.

A Worker also has to keep, for each shared variable, a list of Workers that have to be informed when that shared variable is changed. This information is received during the initialization process.

4.8.3 Messages handled by a Worker

A Worker has to communicate with other agents in order to cooperatively solve the constraint propagation process. This communication is performed using a message-passing mechanism.

The most important messages handled by a Worker are the following:

- `add_constraint(type, id, listOfVarIds)`: This message is used by the ConstraintSystem to sequentially send the constraints of the list of constraints on which the Worker has to run the constraint propagation algorithm.

- `add_node_for_variable(nodeId, varId)`: This message is sent by the ConstraintSystem agent, and informs the receiving Worker about the identity of a Worker that has to be notified when the specified shared variable is changed.

- `update_var(varId, value)`: Initially, this message comes from the ConstraintSystem agent. It is used during the initialization process to inform the receiving Worker about the value of a variable. Subsequently, the message is also sent during the actual propagation process by a different Worker to inform the receiving Worker about a new value for a shared variable.

- `start_prop()`: This message sent by the Monitor. It indicates to the receiving Worker to start the constraint propagation process.

- `get_var(varId)`: After the constraint propagation is finished, the values of the variables must be collected. This message is sent by the

ConstraintSystem agent and represents a request to collect the value for a variable. The Worker has to send back the value for the indicated variable. The receiving Worker replies with a corresponding message.

- `kill()`: This message is sent by the Monitor when the propagation is terminated. The receiving Worker concludes that the propagation is finished, and that its services are no longer required, and as a consequence, it releases all resources and terminates itself.

4.9 The ConstraintSystem agent

To achieve modularity and gain flexibility a special agent, which encapsulates the constraint system that must be solved, exists. This agent is the ConstraintSystem agent.

4.9.1 The Activity of the ConstraintSystem

The ConstraintSystem agent has two main functions, one during the initialization phase, and one after the constraint propagation is finished. A state-chart showing the behaviour of the agent ConstraintSystem is given in Figure 4.5.

Firstly, during the initialization phase, the ConstraintSystem agent has to partition the constraint system in subsystems in order to distribute it to the participating Workers. The way in which the constraint system is partitioned is presented in section 4.10. The ConstraintSystem agent is created by the Monitor agent, which also sends it the number of partitions in which the system of constraints must be partitioned, and the identities of the Workers that are to be used for the propagation. This information is received while in the *GetNodeAddrs* state. After this information is received, the ConstraintSystem agent distributes the identities of the Workers to each Worker. Next, the agent creates the partitions and distributes them, one to each Worker. The elements that compose a partition are distributed while the agent is in *DistributeVariables*, *DistributeConstraints* and *DistributeNodesForVars* states. Finally, after the distribution phase is terminated, the agent sends a message to the Monitor notifying it about this. Next, the agent waits in the *DistributionDone* state for a confirmation from the Monitor agent.

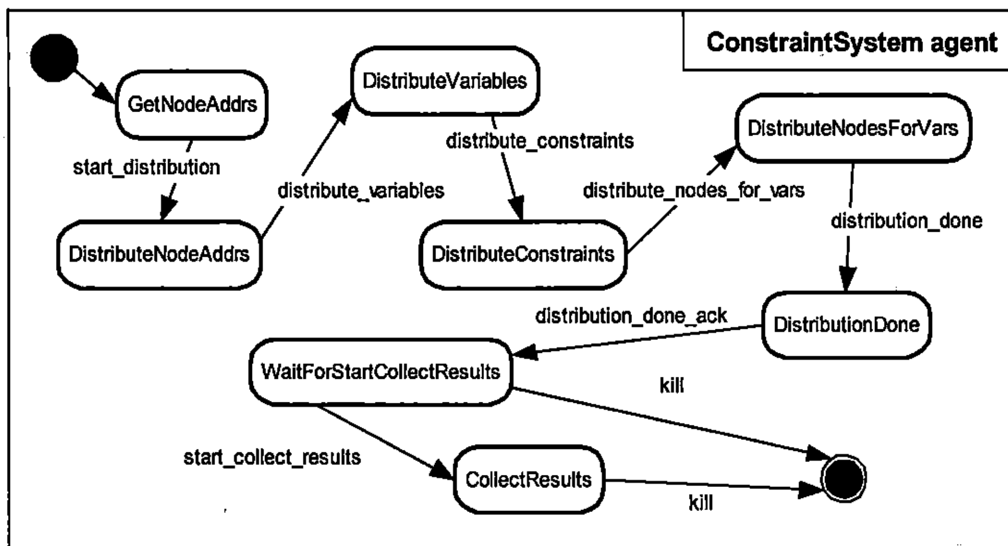


Figure 4.5 ConstraintSystem agent state-chart

Secondly, the ConstraintSystem agent has to collect the results, that is, the values of the variables that are part of the constraint system. During the actual propagation, the ConstraintSystem agent just waits in the *WaitForStartCollectResults* state for the termination of the propagation. When this happens, a message from Monitor informs the ConstraintSystem agent that the results must be collected. Since the ConstraintSystem agent has the knowledge about the constraint system, the responsibility of collecting the results is its. Next, as the last phase of its existence, while in the *CollectResults* state, the ConstraintSystem agent, for each variable for which it wants to collect a value, sends a message to an appropriate Worker requesting the value of that variable. When the values for all variables have been collected the final task of the ConstraintSystem agent is finished and the Monitor is informed about this.

4.9.2 Data handled by the ConstraintSystem

The information that the ConstraintSystem agent has to know, which is actually the core of the problem, is a set of constraints that is part of the constraint system that has to be solved.

Due to the fact that at the end of the constraint propagation the ConstraintSystem agent has to collect the results, that is, the values for all variables involved in the constraint propagation, the ConstraintSystem agent has to know which variables are involved in this process. This information is actually an intrinsic part of the constraints themselves. Also, because the ConstraintSystem agent is the one who partitions the constraint system, it seems appropriate that the ConstraintSystem agent should be the one who assigns unique identification numbers to all variables involved in the constraint propagation process.

For each shared variable the ConstraintSystem agent has to keep a list, which contains the Workers that share that variable. This information is required because it has to be provided to each Worker, that shares that particular variable, during the initialization of the constraint propagation. Therefore, in order to be able to generate these lists, the ConstraintSystem agent needs to know the identity of the Workers available for the constraint propagation.

Based on the list of constraints and the number of available Workers, the ConstraintSystem agent has to partition the existing set of constraints in a set of subsystems equal to the number of Workers. The partitioning procedure is presented in section 4.10.

Finally, the ConstraintSystem agent must know the identity of the Monitor agent. This is required since the ConstraintSystem agent needs to inform the Monitor when the distribution is terminated, and also, when the results have been collected.

4.9.3 Messages handled by the ConstraintSystem

The communication between the ConstraintSystem agent on one hand, and the Monitor and Workers on the other hand, is performed using an asynchronous message passing mechanism. This section presents the messages handled by the ConstraintSystem agent.

The most relevant messages handled by the ConstraintSystem agent are the following:

- `start_distribution()`: Message sent by the Monitor to indicate that the distribution of the subsystems should start. As a result of receiving this

message, the ConstraintSystem agent partitions the constraint system in subsystems and distributes them to the participating Worker agents.

- `start_collect_results()`: This message is sent by the Monitor after the propagation is finished. It indicates to the ConstraintSystem agent that it can start to collect the results.
- `get_var_ack(id, value)`: This message is sent by a Worker, as a reply, when the ConstraintSystem agent collects the value for the indicated variable.
- `kill()` : This message is sent by the Monitor when the constraint propagation failed. When this message is received the ConstraintSystem terminates itself.

4.10 Constraint system partitioning

One of the tasks that the ConstraintSystem agent has to perform, during the initialization phase, is to partition the constraint system in subsystems, so that the subsystems can be distributed among the Workers. The partitioning process consist of creating a set of disjoint subsystems of constraints equal to the number of partitions in which the system must be partitioned.

The ConstraintSystem agent has to know the number and the identity of the Workers that will be used to perform the constraint propagation process. The number of Workers is required because the constraint system is partitioned in a number of partitions equal to the number of available Workers. The identity of the Workers is needed because the partitions have to be sent to them, and also because after the propagation is finished the ConstraintSystem agent has to collect the results by contacting the Workers.

4.10.1 Constraints clustering

An important aspect when performing the partition process is to choose the partitions in a way that minimizes the communication between Workers. This is desired because normally, for the system that exists today on the market, the cost of communication is the one that limits the performance by introducing a bottleneck. The communication between

Workers is required because constraints share variables, and each Worker that shares a particular variable has to be informed about the most recent value for that variable each time a new value is found for that variable.

Each complex constraint has to be translated in a set of primitive constraints. This translation process introduces *auxiliary variables*. In a real application the result is a large set of primitive constraints that has to be partitioned into subsystems. If this partitioning is performed randomly, that is, constraints are assigned arbitrary to partitions, it is very likely that, at least regarding the volume of traffic required to update shared variables between Workers, the partitions are not optimal, that is, the traffic between Workers is not as low as it could be.

A better approach for building the partitions is one in which the primitive constraints that are produced by decomposing a complex constraints are grouped together in a cluster. An approach like this is also mentioned in [1], but its purpose is different. This grouping makes sense because the auxiliary variables introduced by the translation process become local variables and are used only locally within the group. A way to keep the traffic between Workers low is to avoid splitting a group generated from a complex constraint by sending the composing constraints to different Workers. Instead, if a group is sent to only one Worker, the result is that the Worker does not have to send updating information about the auxiliary variables of a group, because those variables are only used locally. Therefore, creating partitions based on the complex constraints, before the complex constraints are translated into primitive constraints, prevents the introduction of additional traffic between Workers after the complex constraints are translated into primitive constraints, since all auxiliary variables introduced after translation are local to a partition.

To exemplify the clustering process described previously, the following simple but illustrative constraint system composed of two complex constraints is considered:

$$\begin{cases} (x+1)^2 + 3xy = y \\ 2 * y^2 + 3 * z = 7 * y \end{cases}$$

Translating these two complex relations into primitive relations yields the following set of primitive constraints:

$$x+1=x_1,\quad x_1^2=x_2,\quad x*y=x_3,\quad 3*x_3=x_4,\quad x_2+x_4=y$$
$$y^2=x_5,\quad 2*x_5=x_6,\quad 3*z=x_7,\quad 7*y=x_8,\quad x_6+x_7=x_8$$

If two partitions are created, one containing the primitive constrains resulted from the first complex constraint, and the other one containing the primitive constraints yielded by the second complex constraints, there is only one shared variable, that is, *y*. All auxiliary variables are local variables in this case and an optimal set of partitions is produced. However, if instead the primitive constraints $x_1^2=x_2$ and $2*x_5=x_6$ switch places, that is, the first goes into the second partition and the second goes into the first partition (no clustering is used), the auxiliary variables x_1, x_2, x_5 and x_6 become shared variables too, and as a consequence more variables are shared, which leads to more traffic between Workers. Therefore, keeping the auxiliary variables introduced by a complex constraint grouped together has a beneficial outcome, and it is the first step in creating some partitions that keep the traffic between Workers low.

4.10.2 Partitioning algorithm

Clustering is necessary but still not sufficient to keep the traffic low between partitions since by just randomly assigning complex constraints to partitions it is very likely to produce partitions with many shared variables. Therefore, assigning complex constraints to partitions requires an adequate strategy that keeps the number of shared variables low.

A possible solution is to use a *greedy* strategy to assign the clusters to partitions. This implies creating the partitions based on a local optimization method. The algorithm used in this thesis is presented in Figure 4.6.

```
ICS ← the interval constraint system;
P ← the set of partitions, initially empty;
C ← the set of clusters;
i = 0;
while C not empty
```

```
find cluster j in C that added to P[i] adds a
     minimum cost to ICS;
remove j from C;
add j to P[i];
i = (i+1) % no_partitions;
```

Figure 4.6 The partitioning algorithm

As the algorithm from Figure 4.6 indicates, the partitions are created as a sequence of steps, at each step a cluster of primitive constraints being added to a partition in a way that keeps the partitions optimal. The optimality here refers to the communication cost between partitions, which is reflected by the number of shared variables between partitions.

The algorithm starts by selecting the partition to which a cluster will be added, that is, the partition `i`. Next, it finds the cluster `j` that added to the partition `i` adds a minimum cost to the set of partitions already created. The optimal cluster `j` is removed from `C` and added to partition `i`. The last statement selects the next partition and the process is repeated until all clusters have been assigned to a partition.

As a local optimization method, this strategy may not produce the best solution, but it is one computationally efficient way of approaching the problem, and presumably, yields very good (or even the best possible) results.

The method used in this thesis to create the partitions is a combination of the two strategies presented above, that is, clustering followed by a greedy strategy of assigning clusters to partitions. Concrete implementation details are presented in section 5.6.

As output, the partitioning process yields a number of constraint subsystems. Also, it generates, for each shared variable, a list that contains the identity of all Workers that share that variable. This information has to be sent to each Worker involved in the constraint propagation process, so that when a shared variable is changed the Worker can inform all the other Workers about this change.

It is also worth mentioning that for many real problems there are natural ways of partitioning them. For those cases, it is very likely that a natural partitioning process

yields better partitioning than an automated partitioning process. However, it is also possible that natural partitioning works best only for a particular number of nodes (for example even number of nodes). In those cases an automated partitioning process can be very useful.

4.11 Constraint propagation failure

The constraint propagation fails when a Worker finds an empty interval for a variable. When this happens the Worker becomes idle "forever" and ignores any update messages that it might receive from other Workers. It also informs the Monitor that the constraint propagation failed by sending it a `failed()` message. When the Monitor receives this message it sends a `kill()` message to all agents, that is, the Workers and the ConstraintSystem. This terminates the constraint propagation and releases all agents.

5 Implementation issues

This chapter presents an implementation of a parallel constraint propagator that reflects the conceptual model presented in the previous chapter. The interfaces for the main classes are presented and also the interaction between them.

C++ is used as a computation language and COOL as a coordination language. The implementation follows object oriented practices and principles. The core part of the implementation consists of a set of C++ classes that implement the required functionality. This approach was used since COOL is a pure coordination and communication language and because C++ offers greater flexibility in implementing the required functionality. COOL is used to provide the communication and coordination facilities that are required for a parallel propagation. The agents are written in COOL and they make use of the core C++ classes by creating instances of them. The agents coordinate their activity by communicating using a message passing mechanism provided by COOL. As a result the whole system is a collection of two kinds of objects: passive objects written in C++, and active objects written in COOL as actors.

The implementation is tested on a distributed network of workstations running RedHat Linux 6.2 connected through an Ethernet network.

5.1 Variables/Constraints naming scheme

Since objects, more specifically constraints and variables, are distributed across machine boundary, an appropriate naming scheme that provides a unique way of identification is required. To accomplish this a simple scheme is used based on unique identification numbers. Each variable is assigned an identification number that is unique across the

whole system. Similarly, the constraints are also uniquely identified using the same scheme.

Another reason for using this naming scheme is that there is a need to exchange information regarding the value of a variable between the C++ objects and the COOL agents. COOL does not have pointers, and has only a limited set of primitive data types, and it does not allow passing complex objects as parameters to messages. The adopted scheme is a simple and convenient solution to this problem.

5.2 The passive objects

As indicated in the introductory part of this chapter, the passive objects are written in C++, and consists of a collection of classes that implement the core functionality of the system. This section presents the most relevant passive objects and details regarding their implementation.

To provide flexibility and especially the benefits of code reuse, two categories of passive objects have been created. The first one contains the `Real`, `Variable`, `Constraint`, and `Propagator`, and can be used either in a parallel or a sequential program. The second category includes `SharedVar` and `PPropagator`, and they are specific for an application that performs a parallel propagation.

5.2.1 The `Constraint` class

The `Constraint` class is shown in Figure 5.1, and is a simple abstract class that contains only two significant methods.

```
class Constraint {
public:
    virtual bool shrink(set<Variable*>&) = 0;
    ConstraintId getId();
};
```

Figure 5.1 The interface of the `Constraint` class

There is an abstract method `shrink(set<Variable*>&)` that is supposed to implement the contraction operator for a concrete constraint. The method has an output parameter that consists of a set of `Variables` that have been changed as a result of applying the contraction operator. Normally, for a sequential propagator this output parameter is not required because the changes that might happen to a `Variable` can be directly propagated by that `Variable`, since everything is local within the same memory space. More details regarding this are given in section 5.2.3. However, for the parallel propagation case, a `Worker` needs to send update messages to the corresponding `Workers` whenever a shared variable is changed. This is the reason why the output parameter indicated previously is required.

The second method, `getId()`, returns the unique identification number associated with this `Constraint`.

Depending on the concrete constraint, the `shrink()` method implements the appropriate contraction operator. There are concrete constraint classes that extend the abstract `Constraint` class for each primitive constraint. Here is the place where the core implementation of the interval arithmetic operations and of the constraint contraction operator is encountered. As concrete implementation for these operations, this thesis uses the same approach as the one presented in [1].

A concrete constraint class has a number of pointers to the `Variable` objects that it shares. The pointers to these `Variables` are supplied to the constructor of a concrete constraint, and while the constructor is executed, corresponding references are added to the constructed object to each `Variable`, by calls made to each `Variable` object. The methods `addConstraint()` and `delConstraint()`, provided by the `Variable` class, are used for this purpose.

5.2.2 The `Variable` class

The `Variable` class encapsulates the characteristics of an unknown as it is defined by interval constraints, and in addition attributes that are required in the context of using the class together with a `Propagator`. A code snippet of it is shown in Figure 5.2.

```
class Variable {
private:
     Real value;
     VariableId id;
     set<Constraint*> cons;
     stack<Real> stack;
protected:
     Propagator *propagator;
public:
     Variable(VariableId id, Real &val, Propagator *p);
     bool addConstraint(Constraint *c);
     bool delConstraint(Constraint *c);
     int getNoRefCons()

     bool update(Real &val);
     Real getValue();

     vector<Constraint*>& getConstraints();
     VariableId getId();

     void store();
     void restore();
};
```

Figure 5.2 `Variable` class code snippet

A `Variable` has a value, a unique identification number, a set of constraints that share it, and a stack that can store its current value. Since a `Variable` makes sense only in the context of a constraint system, the class also contains a pointer to a `Propagator`, which is initialized while the object is constructed.

The `addConstraint()` and `delConstraint()` methods are used when a `Constraint` object is respectively crated and deleted. Since constraints share variables

a Variable object can be referred by more than one Constraint object. Moreover, when the value of a Variable is changed, the corresponding Constraints must be reactivated. These are the reasons why these methods are required, and why a Variable object keeps a set of pointers to Constraint objects.

The getNoRefCons() method returns the number of Constraint objects that refer this Variable object. This method is required by the Propagator object in order to decide whether a Variable object must be garbage collected. This situation arises when a Constraint object is removed from the Propagator. Since the Propagator object becomes the owner of a Variable after the first Constraint object, that shares it, is added to the Propagator, the Propagator has to decide if a Variable must be deleted or not.

The next two methods are used to update the value of a Variable and to retrieve its value. The update() method tries to update the current value of the Variable by doing an intersection with the old value. The method returns true if the value is changed.

The next two methods are used to retrieve the set of Constraints shared by the Variable and the unique identification number associated with the Variable.

The last two methods are used to store, respectively, restore the current value of a Variable from an internal stack. This feature, of storing the current value, and restoring it later, is required by both splitting and probing.

5.2.3 The Propagator class

The Propagator class encapsulates a constraint system as it is defined by the interval constraints method. The Propagator object uses Variable and Constraint objects, connected in a network of constraints, on which it runs the propagation algorithm.

A code snippet of the Propagator class that lists the most relevant methods and member fields is shown in Figure 5.3.

```
class Propagator {
```

```
private:
      set<Variable*> v;
      set<Constraint*> c;
      deque<Constraint*> qac;
      stack<deque<Constraint*> > qacStack;
public:
      void addConstraint(Constraint *c);
      void delConstraint(Constraint *c);

      void activateAll();
      void variableChanged(set<Constraint*> &cons);

      void store();
      void restore();

      bool shrink(unsigned long maxIter, unsigned long
           &actualIters, set<Variable*> &changedVars);
      bool probe(Variable *var, bool propagateChanges);
      bool done();
};
```

Figure 5.3 `Propagator` class code snippet

The `Propagator` class is a holder for a set of `Constraint` and `Variable` objects, that is, for a constraint system. The first category of provided methods are used for adding or removing a `Constraint` to or from the set of constraints. There is no need for some corresponding methods for handling `Variable` objects since this is done transparently for the user of the class while a `Constraint` is added respectively removed. Details regarding this are illustrated in Figure 5.4.

The `store()` and `restore()` methods are used to store and restore respectively the state of the `Propagator` at a particular time. The state of the `Propagator` consists of the set of values, one for each `Variable`, and the contents of the queue of active

`Constraints`. These methods are required if the user of the class intends to implement, in addition to propagation, probing and/or splitting.

The `shrink()` method implements the actual propagation algorithm. It takes as input parameter the maximum number of iterations that it should perform. An iteration represents the contraction of one single constraint. The output parameters indicate the actual number of iterations performed and a set that contains the `Variables` that have been changed. This set of `Variables` can be used to do some probing and if the `Propagator` is used for a parallel propagation to allow the calling entity to notify other `Propagators` about the `Variables` that have been changed. The method extracts constraints from the queue of active constraints and applies the corresponding contraction operator on them.

The `probe()` method performs probing for the given `Variable`. If the probing succeeds the method returns `true` and the value of the `Variable` is updated. The input parameter `propagateChanges` indicates if the changes should be propagated or not, that is, in the case the probing succeeds, if the `Constraints` dependent on this `Variable` should be activated or not.

The `activateAll()` method is used to activate all the `Constraints`, and is used at the beginning of the constraint propagation and when probing is done. The method is also required if splitting is implemented.

The `variableChanged()` method is a call back from a `Variable` object when its value changes. As a consequence, the `Propagator` activates all the `Constraints` sharing the calling `Variable`.

The `done()` method indicates if the propagation if finished or not, that is, if the queue of active constraints is empty or not.

A sequence diagram illustrating the way `Variable` and `Constraint` objects are created and deleted is shown in Figure 5.4. The diagram also shows the sequence of calls that take place when a propagation is performed, that is, a call to `shrink()` is made, and when a `Variable` is updated.

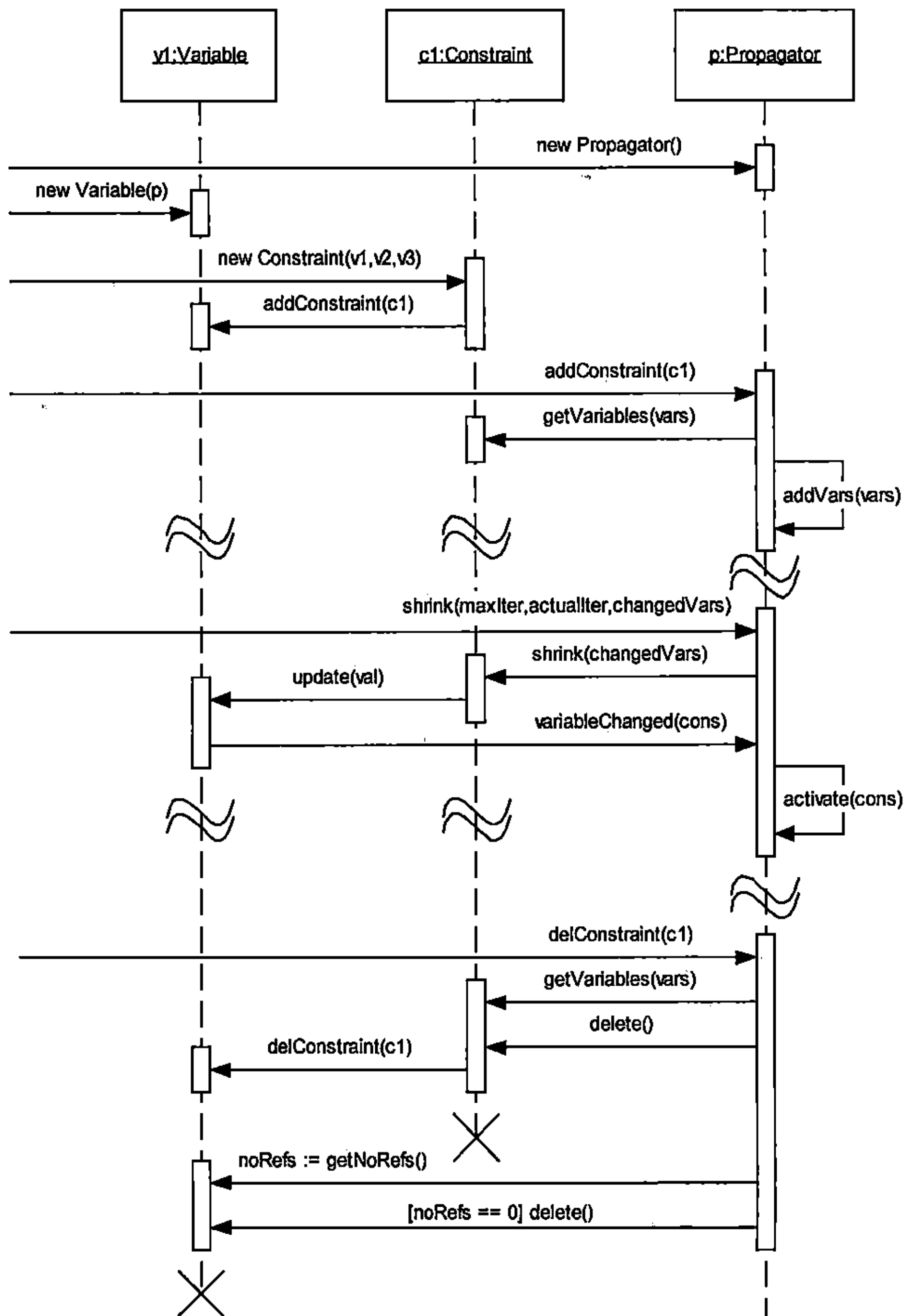


Figure 5.4 `Variable-Constraint-Propagator` sequence diagram

The sequence diagram in Figure 5.4 shows only the relevant method calls and parameters. To simplify the diagram, only one `Variable` object is indicated, even if

there are references to more objects. The diagram also shows how the Propagator object builds internally the network of Constraints and keeps it updated.

Due to their modularity, the Propagator, Constraint, and Variable classes can be used to execute the constraint propagation algorithm in a sequential or in a parallel application. A sample program that shows how they can be used in a sequential application is given in Appendix B.

5.2.4 The SharedVar class

The SharedVar class, shown in Figure 5.5, extends the Variable class with functionality required for a parallel propagation. The extension consists in adding a list that contains the nodes that share the Variable. There are two additional methods provided by this class, one used to add a node, and one used to retrieve the list of nodes.

```
class SharedVar : public Variable {
private:
    vector<NodeId> nodes;
public:
    void addNode(NodeId n);
    vector<NodeId>& getNodes();
};
```

Figure 5.5 The SharedVar class code snippet

5.2.5 The PPropagator class

The PPropagator class encapsulates a constraint subsystem, and is just a wrapper for the Propagator class and the other elements required to perform a parallel propagation. Each Worker agent creates an instance of this class, and it uses it to solve the partition of the constraint system that has been assigned to it. A code snippet of the PPropagator class is shown in Figure 5.6.

In spite of the fact that the class has a relatively large number of methods, its complexity is relatively reduced. The large number of methods is just a set of primitive operations that are required since COOL is a pure coordination/communication language and the facilities to specify/create object directly is limited. Therefore, the primitive methods indicated in Figure 5.6 correspond, to COOL messages.

Internally, the `PPropagator` class keeps the current state of the propagation process as a `Propagator` object. All the information encapsulated into the `PPropagator` class is used by the `Worker` agent to run the propagation algorithm. There are two maps, one that maps variable ids into `Variable` objects, and one that maps constraint ids into `Constraint` objects. There is also a queue that keeps the set of changed variables. This is required to keep the list of shared variables that have been changed as a result of applying the contraction operator on some active constraints. As a result of a shared variable being changed the `Workers` that share it must be notified. The `ccv` (current changed variable) field is used to indicate a variable whose value has to be sent to the other `Workers` that share it. There is also an iterator, `cn` (current node), that indicates the current `Worker` to which the update message is to be sent. A very important field defined by `PPropagator` is `cpi` (constraints per iteration). This field indicates how many constraint contractions have to be performed by each call to `shrink()`. The significance of this parameter is described at the end of section 4.8.1. This parameter controls how many contractions a Worker performs before any update messages are sent.

```
class PPropagator {
private:
     map<VariableId, Variable*> v;
     map<ConstraintId, Constraint*> c;
     Queue<Variable*> qcv;      // q of changed variables
     Variable* ccv;             // current changed variable
     vector<NodeId>::iterator cn;  // current node
     Propagator p;
     unsigned long cpi;         // constraints per iteration
public:
```

```
    bool addVariable(VariableId);
    bool addVariable(VariableId, double);
    bool addVariable(VariableId, double, double);
    bool addSum(ConstraintId, VariableId, VariableId,
         VariableId);
    bool addProd(ConstraintId, VariableId, VariableId,
         VariableId);
    bool addLeq(ConstraintId, VariableId, VariableId);
    bool addEq(ConstraintId, VariableId, VariableId);
    bool addPowerN(ConstraintId, VariableId, VariableId,
         int);
    bool addConstraintForVariable(ConstraintId,
         VariableId);
    bool addNodeForVariable(NodeId, VariableId);

    void activateAll();
    bool shrinkDone();
    bool shrink(bool probe);

    bool hasChangedVariables();
    VariableId getNextChangedVariableId();
    double getLBVariable(VariableId);
    double getUBVariable(VariableId);
    bool hasNodes();
    NodeId getNextNodeId();

    bool updateVariable(VariableId, double, double);
};
```

Figure 5.6 `PPropagator` class code snippet

With respect to the methods defined by this class, there is a first set of `add*` methods that are used during the initialization phase to initialize the `Propagator` with the partition of the constraint system that has to be solved by this `Propagator`.

Many methods are just wrappers for calls made to methods defined by the `Propagator` class. After the initialization is terminated, all constraints are activated using a call to the `activateAll()` method. The `shrink()` method is just a wrapper for a call made to the `shrink()` method defined by the `Propagator` class. The method `shrinkDone()` tests to see if the propagation is finished, that is, the queue of active constraints is empty.

The next set of methods are used to retrieve the new values of the shared variables that have been changed, as a result of applying the contraction operator, and also to retrieve the identity of the `Workers` that have to be informed about the change. These methods are used by the `Worker` to send updating messages to other `Workers`.

The `updateVariable(VariableId, double, double)` method is used to update a shared variable as a result of receiving an incoming update message.

5.3 The active objects

The active objects represent the agents that were introduced in the model presented in chapter 4, and are implemented in COOL as actors. The interface of each actor describes an object as an asynchronous state machine. An actor can be in one of several states. While in a particular state, an actor can receive and handle only the messages listed under that state.

This section describes the interfaces, and some implementation details, for the most relevant actors used within the system. The behaviour of each of these agents is described also, at a conceptual level, in sections 4.7, 4.8 and 4.9.

5.3.1 The `Monitor` actor

The functionality of the Monitor agent was presented in section 4.7, and is implemented in COOL as an actor whose interface is presented in Figure 5.7. The `Monitor` actor can be in one of several states, as indicated in Figure 5.7.

```
actor class Monitor {
Start:
    message INIT(string m);
RegisterToMom:
    message register_as_monitor();
    message register_ack(bool succeded);
Subscribe:
    message subscribe(Worker w);
SetNodeAddrs:
    message set_node_addrs();
    message set_node_addr_no_ack(int wId);
    message set_node_addr_ack(int wId);
StartDistribution:
    message start_distribution();
    message start_distribution_ack();
WaitForEndOfDistribution:
    message distribution_done();
WatchPropagation:
    message start_prop();
    message start_prop_ack(int wId);
    message set_node_status(int wId, bool s);
    message propagation_done();
    message failed();
CollectResults:
    message start_collect_results();
    message start_collect_results_ack();
    message done_collect_results();
```

```
DismissWorkers:
     message dismiss_workers();
DeregisterAsMonitor:
     message deregister_as_monitor();
     message deregister_ack(bool succeded);
     message kill();
}
```

Figure 5.7 The interface of the `Monitor` actor

After the actor is created and initialized, it enters in the `RegisterToMom` state, in which it tries to find `Mom` and to register as `Monitor`. Next, after the registration is completed successfully, the `Monitor` enters in the `Subscribe` state. The `Monitor` stays in this state and processes subscribe messages from `Workers` until a sufficient number of actors register as `Workers`. Since each `Worker` needs to know the identity of the other `Workers`, the `Monitor` distributes this information to `Workers` while in `SetNodeAddrs` state. At this point, from the `Monitor` point of view, the distribution can be started. Next, the `Monitor` waits until the distribution is terminated. This stage is reached when it receives the message `distribution_done()`.

During the actual propagation, the `Monitor` actor is in the `WatchPropagation` state. While in this state, a `set_node_status(int wId, bool s)` message is used to track the status of its `Workers`. This message indicates a change in status, to idle or active, for a particular `Worker`. When the termination condition is reached, that is, all `Workers` are idle, the `Monitor` self-triggers a `propagation_done()` message, which eventually changes its state to `CollectResults`. While in this state, the `Monitor` waits for the `ConstraintSystem` actor to collect the results of the propagation.

Eventually, the `Monitor` receives the `done_collect_results()` message, which indicates that all results have been collected and that the `Workers` can be released. This is accomplished while in the `DismissWorkers` state. During the last phase of its life,

the actor deregisters itself as `Monitor` from `Mom` and self-triggers a `kill()` message that terminates itself.

The `CollectResults` state can be skipped when the `failed()` message is received. This indicates that the constraint propagation performed by the sending `Worker` has failed. As a consequence, the `Monitor` sends a `kill()` message to the `ConstraintSystem` actor and then goes directly into the `DismissWorkers` state.

As can be noticed, there are many acknowledgement messages. This is due to the fact that COOL assumes that messages are transmitted over an unreliable medium. Therefore, a proper mechanism of acknowledgement/retransmission has been implemented where necessary.

5.3.2 The `Worker` actor

The `Worker` actor is one of the most important within the system. The `Worker` actor is the one that caries on the actual propagation process. The description of the Worker agent is presented in section 4.8, and the interface of the actor that implements it is shown in Figure 5.8. During its lifetime, a `Worker` can be in one of several states.

```
actor class Worker {
Start:
     message INIT(string m);
SearchMonitor:
     message lookup_monitor(int n);
     message lookup_ack(actor a);
SubscribeMonitor:
     message subscribe_to_monitor();
     message subscribe_ack(int i);
ReceiveConstraintsSystem:
     message set_node_addr(int nodeId, Worker addr);
     message addVar(int varId, real lb, real ub);
```

```
    message addSum(int conId, int v1Id, int v2Id, int
         v3Id);
    message addProd(int conId, int v1Id, int v2Id, int
         v3Id);
    message addLeq(int conId, int v1Id, int v2Id);
    message addEq(int conId, int v1Id, int v2Id);
    message addPowerN(int conId, int v1Id, int v2Id, int
         n);
    message add_node_for_variable(int nodeId, int varId);
    message start_prop();
Propagate:
    message propagate();
    message update_var(int vId, real lb, real ub);
    message send_node_status();
    message set_node_status_ack();
    message get_var(int vId);
    message kill();
}
```

Figure 5.8 The interface of the `Worker` actor

During the initialization, the `Worker` receives the name of the `Monitor` actor. Then, it goes into the `SearchMonitor` state. While in this state, the actor tries to find a reference to the `Monitor` actor by periodically sending a `lookup_monitor()` message to the `Mom` actor, until it receives acknowledgement. The `Mom` actor is used by the `Monitor` and the `Workers` to find each others. Next, the `Worker` goes into the `SubcribeMonitor` state where it tries to subscribe to the `Monitor` as a `Worker` by repeatedly sending a `subcribe_to_monitor()` message.

After the `Worker` actor successfully subscribes to the `Monitor` actor, it goes into the `ReceiveConstraintsSystem` state where it is ready to receive its partition of the constraint system on which it will have run the propagation algorithm. While in this state, the actor can handle different kind of messages that allow it to step-by-step build the

constraint network that it will work with. The `Worker` stays in this state until the `start_prop()` message is received, which indicates that the propagation can be started.

The next state of its lifetime is the `Propagate` state. This is the state during which the actual propagation process takes place. While in this state, the actor can handle several messages. The most significant one is the `propagate()` message. This is a self-triggered message that handles the contraction of one or more active constraints. During the execution of this message, care is taken, so that all required `Workers` are informed of the new value of any shared variable that may change as a result of applying the contraction operator.

Another significant message that a `Worker` may receive, while in the `Propagate` state, is `update_var()`. This message indicates that the `Worker` might have to update the value of a variable that is referred within the message. Whenever the state of the `Worker` changes (idle/active), the `send_node_status()` message is self-triggered. This message informs the `Monitor` actor that the status of this `Worker` has been changed. The current value of a variable can be obtained by sending a `get_var()` message to the appropriate `Worker`. The `Worker` stays in the `Propagate` state until a `kill()` message is received. This message indicates to the `Worker` that the propagation is finished and that it can terminate itself.

5.3.3 The `ConstraintSystem` actor

The model that the `ConstraintSystem` actor follows is described in section 4.9, and as indicated there, the `ConstraintSystem` actor encapsulates a constraint system, and takes care of partitioning it, distributing it, and eventually collecting the results. The interface of the `ConstraintSystem` actor is shown in Figure 5.9.

```
actor class ConstraintsSystem {
Start:
     message INIT();
GetNodeAddrs:
```

```
    message set_node_addr(int nodeId, Worker addr);
    message start_distribution();
DistributeNodeAddrs:
    message distribute_node_addrs();
    message resend_node_addr(int i, int j);
    message set_node_addr_ack(int ackNo);
DistributeVariables:
    message distribute_variables();
    message variable_distributed_ack(int ackNo);
DistributeConstraints:
    message distribute_constraints();
    message constraint_distributed_ack(int ackNo);
DistributeNodesForVariables:
    message distribute_nodes_for_variables();
    message node_for_variable_distributed_ack(int ackNo);
DistributionDone:
    message distribution_done();
    message distribution_done_ack();
StartCollectResults:
    message start_collect_results();
    message kill();
CollectResults:
    message collect_var(int vId, int nodeId);
    message get_var_ack(int vId, real lb, real ub);
EndCollectResults:
    message done_collect_results();
    message done_collect_results_ack();
}
```

Figure 5.9 The interface of the `ConstraintSystem` actor

The `ConstraintSystem` actor is created by the `Monitor` actor. Right after it is created, the actor has to load the constraint system that has to be solved and to partition it

in a number of partitions equal to the number of Worker actors. To accomplish this, it uses a set of passive objects that implement the required functionality. The concrete implementation uses a configuration file that indicates the name of a text file that contains the constraint system and the number of Workers that must be used to solve it. The ConstraintSystem actor uses this information as input, opens the file, loads the constraint system, and creates a network of constraints by partitioning the constraint system in the corresponding number of partitions. The partitioning process is a combination of clustering and greedy strategy, and follows the model presented in section 4.10. Details regarding the concrete implementation are presented in section 5.6.

Next, the actor has to distribute the partitions of the constraint system to the participating Workers, and for this, it needs the addresses of all Workers. This information is received at the beginning of its lifetime, while it is in the GetNodeAdrs state.

In spite of the fact that its interface is quite large, having many states and messages, the ConstraintSystem actor is not a complex one. During the early stage of its lifetime, represented by the set of Distribute* states, the actor distributes the partitions to the participating Workers. After the distribution is finished, the actor waits idle during the whole propagation process for the end of the propagation, when it receives the start_collect_results() message, from the Monitor, indicating that the results must be collected. Since the ConstraintSystem has the knowledge of the way the system is partitioned, it is the one that collects the results. This is performed during the CollectResults state. Eventually, after all results have been collected, the Monitor is notified about this and that the Workers can be dismissed. This is accomplished by self-triggering the done_collect_results() message. After this, the actor terminates itself.

The ConstraintSystem actor can terminate its execution earlier, without the need to collect the results, when the constraint propagation fails. This happens when the kill() message is received, while the actor is in StartCollectResults state.

5.4 The overall interaction between actors

Figure 5.10 contains an UML sequence diagram that shows the overall interaction between the main actors that compose the system. This diagram shows nicely the way the actors are brought to life, and also the way they interact during their lifetime. This is still a high level diagram, in the sense that only the most representative messages are indicated. The part that contains the distribution of the constraint system is just schematically indicated through the `distribute()` message. In reality, there are a set of messages that are sent during this phase, between the `ConstraintSystem` actor and the `Worker` actors. The diagram does not show the case when the constraint propagation fails.

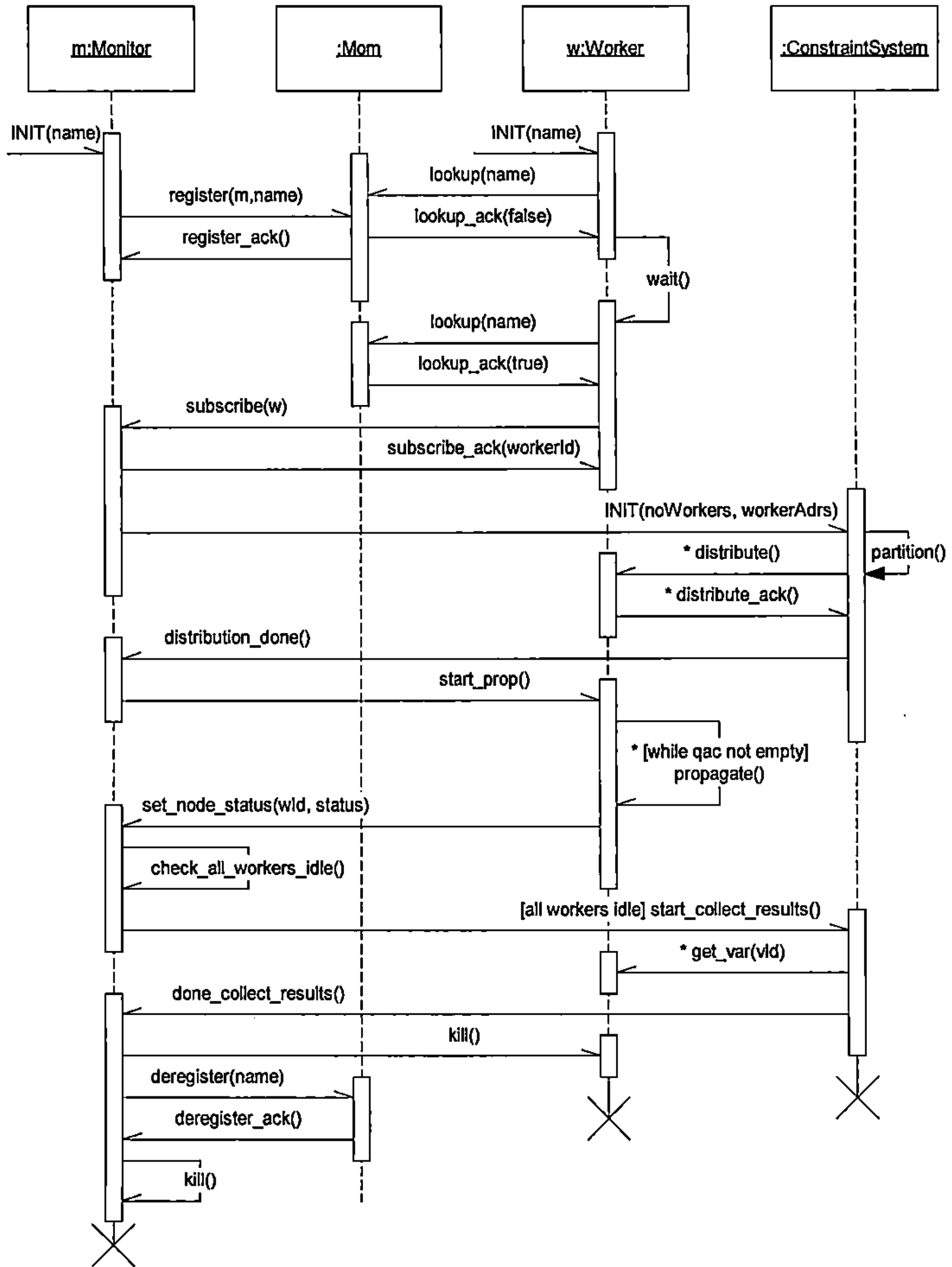


Figure 5.10 The overall interaction diagram between actors

5.5 Monitor – Workers interaction

This section describes the interactions between the Monitor and the Worker actors, starting from the moment the propagation is initiated, until all Workers become idle, that is, the moment the propagation process terminates. A UML sequence diagram that shows this is shown in Figure 5.11. The interaction diagram shows both the activity of the Monitor actor and the activity of the Worker actors.

The propagation is started by the Monitor by sending start_prop() messages to each Worker. As a consequence of receiving this message, each Worker starts the propagation process, which keeps going as long as there are active constraints in its corresponding queue of active constraints (qac). This propagation is kept alive by each Worker by self-triggered self-addressed propagate() messages. During each propagate() message, a parameterized number of constraint are contracted. The propagate() messages are interleaved with the update_var() messages received from other Workers.

Whenever a Worker changes its state, it sends an appropriate message to the Monitor indicating its new state, that is, idle or active. A Worker can change its status several times during the entire propagation process as perceived by the Monitor actor. For instance, Figure 5.11 shows the way the Worker w2 changes its status according to the following sequence: active-idle-active-idle. In the example presented in Figure 5.11 only w2 changes its status twice, w1 being active during the whole propagation process.

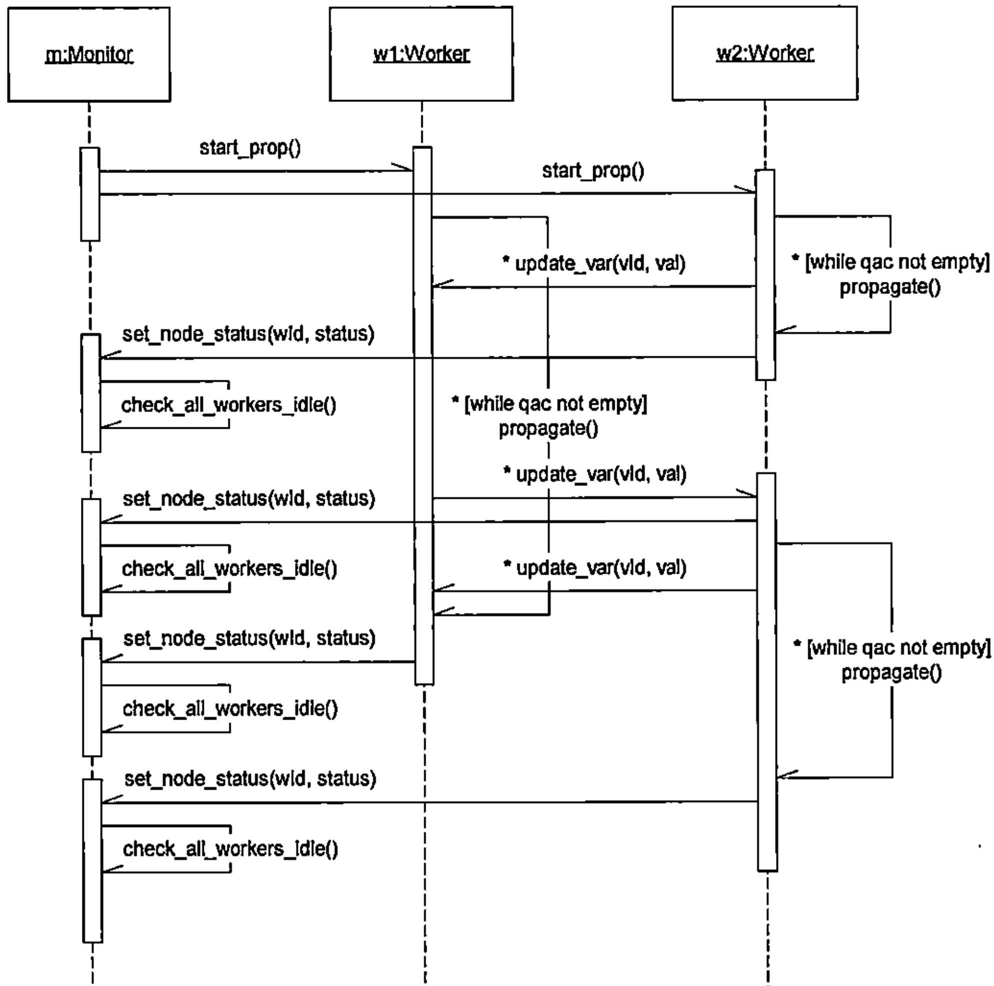


Figure 5.11 `Monitor` - `Workers` interaction diagram

The diagram also shows the way the termination condition is detected. Whenever the `Monitor` receives a `set_node_status(wId, status)` message, it checks to see the status of all `Workers`, and if they are all idle, then the propagation is considered terminated. There is an important aspect that is not captured in the diagram, and that is the way the `Monitor` concludes that all `Workers` are idle. When the `set_node_status(wId, status)` message is received, and all `Workers` are idle, a timer is started. This timer can be cancelled only by a new `set_node_status(wId, status)` message that indicates that a Worker has become active again. The reason why this can happen is presented in section 4.5. If the

timer succeeds to reach the timeout state, then, and only then, the `Monitor` considers the propagation terminated.

5.6 Constraint system partitioning

A theoretical description of the procedure of partitioning a constraint system is presented in section 4.10. The partitioning process is accomplished by the `ConstraintSystem` actor.

A configuration file specifies the number of partitions to be created and the name of a text file that contains the constraint system to be partitioned. Based on this information, the `ConstraintSystem` actor opens the file, loads the constraint system, and then creates a network of constraints by partitioning it in the corresponding number of partitions. To accomplish this, the `ConstraintSystem` actor uses a set of objects as indicated in Figure 5.12.

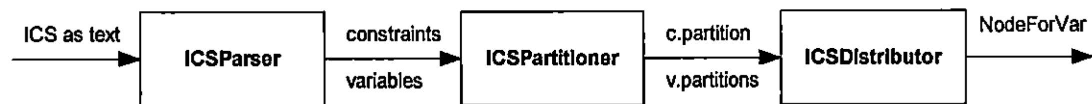


Figure 5.12 The partitioning modules and their interaction

The file that contains the interval constraint system (ICS) specifies it as a set of clusters of primitive constraints. The content of the file is parsed by the ICSParser module that creates as output a set of constraints and set of variables. Each constraint has associated a cluster id from which is part of. The constraints indicate the variables that they share through some unique ids.

Next, the partitions are created. The `ICSPartitioner` module takes the list of constraints and the list of variables, and generates the partitions, using a greedy strategy, to assign clusters to partitions, as presented in section 4.10. Each constraint c is assigned to a partition, situation illustrated in Figure 5.12 by generating the output *c.partition.* Moreover, for each variable v, a set of partitions where the variable is to be distributed is calculated. This is indicated in Figure 5.12 by illustrating the output *v.partitions.*

Finally, the `ICSDistributor` module creates the network of constraints and the associated shared variables. A set of `NodeForVar` pairs is generated, each pair indicating a node that has to be informed when a shared variable is changed.

All this information constitutes the network of constraints and the associated shared variables, and are used by the `ConstraintSystem` actor to distribute the partitions to the participating `Worker` actors.

6 System's performance

This chapter presents some experiments and measurements that reflect the performance of the implementation of the model proposed in this thesis.

6.1 Premises

Typically, in terms of computation costs, performing a constraint contraction requires only a small number of floating-point operations, which takes about 10-20 microseconds (Pentium III at 733MHz). On the other hand, to perform the experiments, due to lack of availability of faster connections, a distributed system of Linux workstations connected through a 10Mbps Ethernet network has been used. For this system, the round-trip communication time was in the range of 500-1000 microseconds. This lead to a situation in which there was a huge discrepancy between the time required to perform an elementary operation, that is, a constraint contraction, and the communication time required to transmit the changes between the computation nodes. In addition, due to difficulty of coding some very large problems as test cases, the problems to be solved were quite small. As a consequence of all these settings, a Worker was most of the time idle, waiting for `update_var()` messages, and the performance of the system was very poor.

The problem is the huge discrepancy between the time required to perform a contraction and the communication time required to transmit a change, due to the poor communication time of the system used as a test platform. However, the target system for the proposed model is a system in which these two times are about the same, since only in that case the changes produced by performing contractions in parallel can be

transmitted as soon as possible from one Worker to a different one and be beneficial for the receiving Worker.

Since a system for which this requirement is met was not available, the time required to perform a constraint contraction was "stretched", so that it became about the same with the time required to transmit changes. In other words, the whole computation process was deliberately made slower, in order to emulate the target configuration. The measurements presented in this chapter have been made for these settings.

6.2 Measurements significance

The results are summarized in Table 6-1 and show the measurements for four different test cases. The table shows a comparison between the performance of the system running with 1, 2 and 3 Workers. Each Worker is executed on a separate machine. To provide a better understanding of the results, a brief explanation of how the system works and how the times are calculated is given.

After the Monitor starts, it waits until the required number of Workers subscribe to it. This time is not included in measurements. Then, the Monitor starts the distribution of the constraints system. The 3rd column of the table indicates this time, that is, the distribution time. When the distribution is finished, the Monitor starts the propagation by sending a `start_prop()` message to each Worker. The time elapsed from this moment until all Workers become idle is required to perform the constraint propagation process, as it is perceived by the Monitor. This time is indicated in the 4th column of the table.

As described in chapter 4, each Worker can be either idle or active. A Worker becomes idle when it has contracted all the constraints that it possesses, that is, its queue of active constraints is empty. A Worker is active when it is contracting constraints, that is, its queue of active constraints is not empty. A Worker can change its status several times during the propagation. For instance, a Worker can become idle and later be reactivated because of some shared variable that are changed by a different Worker. The number of times a Worker changes its status during the propagation process is indicated in column 9. Columns 10 and 11 indicate the time a Worker is idle, respectively active, during its

Table 6-1 Parallel constraint propagation measurements

		Monitor			Worker						Speedup
		Status			Updatings		Contr	Status			from 1 to 2
		Dirstr [ms]	Prop [ms]	Total [ms]	Total	Succ		Chg	Idle [ms]	Prop [ms]	or 3 nodes
Test case 1 — 26 vars 21 cons 3 clusters											
1 Node	M	4	25	29							
	W0				0	0	50	1	0.00	25.00	
2 Nodes	M	58	34	92							
	W0				3	3	13	7	2.78	8.20	
	W1				5	1	47	3	4.50	24.50	0.32
3 Nodes	M	49	26	75							
	W0				6	5	13	9	2.61	8.10	
	W1				5	1	43	3	1.10	20.30	
	W2				6	4	19	5	1.60	10.30	0.39
Test case 2 — 9 vars 7 cons 7 clusters											
1 Node	M	1	123	124							
	W0				0	0	248	1	0.00	123.00	
2 Nodes	M	22	178	200							
	W0				164	163	45	319	52.00	23.40	
	W1				180	172	141	337	73.00	73.00	0.62
3 Nodes	M	26	160	186							
	W0				72	68	70	137	72.10	35.00	
	W1				70	68	72	137	71.30	36.30	
	W2				136	134	142	112	62.20	74.34	0.67
Test case 3 — 26 vars 14 cons 6 clusters											
1 Node	M	3	321	324							
	W0				0	0	641	1	0.00	320.50	
2 Nodes	M	51	363	414							
	W0				127	126	502	127	37.80	301.00	
	W1				127	126	503	129	37.00	302.00	0.78
3 Nodes	M	79	235	314							
	W0				123	123	191	127	9.00	115.00	
	W1				124	123	190	127	9.80	114.00	
	W2				127	126	376	125	2.80	226.00	1.03
Test case 4 — 53 vars 32 cons 9 clusters											
1 Node	M	7	7010	7017							
	W0				0	0	14020	1	0.00	7010.00	
2 Nodes	M	99	6131	6230							
	W0				286	284	10723	13	1.20	6123.00	
	W1				285	285	10729	12	1.00	6129.00	1.13
3 Nodes	M	157	3623	3780							
	W0				332	317	4634	4	0.90	2317.00	
	W1				326	303	4612	6	1.00	2306.00	
	W2				309	293	7234	2	0.31	3617.00	1.86

Legend	
M:	Monitor
Wi:	Worker i, where i=0..2
Monitor	
Distr:	Distribution time
Prop:	Propagation time
Total:	Total time
Worker	
Total:	Total number of *update_variable* messages
Succ:	Succesfull *update_variable* messages
Chg:	Number of times the status of a worker has been changed (from idle to propagation or propagation to idle)
Contr:	Number of constraint contractions performed by this Worker
Idle:	Idle Time
Prop:	Propagation time

lifetime. These elements can be used to give an indication of how well the computation capacity of a Worker is used. As will be seen later in this chapter, it is desirable that a Worker is idle as little as possible during the constraint propagation process as it is seen by the Monitor.

A Worker receives update messages from a different Worker when a shared variable is changed. Such a message is useful, causing the receiving Worker to successfully update its local copy of the shared variable, only when it carries a smaller interval for the variable. Therefore, from the total number of update messages that a Worker receives (column 6), only a fraction is successful (column 7).

The total number of constraint contractions performed by a Worker is indicated in column 8, and depending on the size of the problem, this can show if a problem requires a large or a small number of contractions to reach the final result.

When all Workers become idle, the Monitor stops the timer that measures the propagation time. This time, indicated in column 4, includes the time a Worker is active and the time a Worker is idle. The time indicated in column 5 is the total time required to solve the problem and includes both the distribution time (column 3) and the propagation time (column 4). This time is used to make a comparison between the performance of the

system with 1 node and 2 or 3 nodes. The speed-up indicated in column 12 is calculated, using the following formula:

$$\text{Speedup} = \frac{\text{Total}_i}{\text{Total}_1}$$

In the above formula Total_i is the total time required to solve the test case with *i* nodes, where $i = 1..3$.

6.3 Test cases interpretation

This section presents the four test cases used to perform comparisons and also gives an interpretation of the results shown in **Table 6-1**.

6.3.1 Test case 1

The following system of equations cited in [31] has been used as test case 1:

$$\begin{cases} x_1 + x_2 + x_3 + x_4 - 1 = 0 \\ x_1 + x_2 - x_3 + x_4 - 3 = 0 \\ x_1^2 + x_2^2 + x_3^2 + x_4^2 - 4 = 0 \\ x_1^2 + x_2^2 + x_3^2 + x_4^2 - 2x_1 - 3 = 0 \end{cases}$$

This test case illustrates a problem that requires a small number of contractions to perform a complete propagation, and as a consequence, it converges very fast to the final result. As can be noticed, there is no speed-up for this particular problem. The reason for this is not only that the problem converges fast, but also that the penalty in time required to perform the distribution of the partitions to the participating Workers is very large compared with the time required to perform the actual constraint propagation.

6.3.2 Test case 2

Test case 2 is the following one [37]:

$$\begin{cases} x_1x_2 - x_2 - 2x_1 + 2 = 0 \\ x_1x_2 - 3x_1 - x_2 + 3 = 0 \end{cases}$$

This test case shows a very small problem that requires a larger number of contractions, to converge to the final result, than the previous one. Similar as the previous case, there is no speed-up achieved for this case either. As can be noticed, the time a Worker is idle is very large compared with the time a Worker performs contractions, and this is the reason for such a poor performance. The reason for this is that the problem is very small, and as a consequence, a Worker is most of the time idle waiting for `update_var()` messages from other Workers. However, compared with the previous test case, despite the problem has a smaller size it performs better for the parallel case, and the reason for that seems to be the fact that the problem converges slowly, that is, it requires a large number of contractions to reach the final result.

6.3.3 Test case 3

Test case 3 shows an increase of performance when using 3 Workers. This is due to the fact that the problem converges slowly, and as a consequence, the time required to perform the constraint contractions is large. A speed-up is also obtained. The high number of successful `update_var()` messages are very important for the receiving Workers not only because the assure progress, but also because they act as shortcuts to the final result, and they shorten the propagation process.

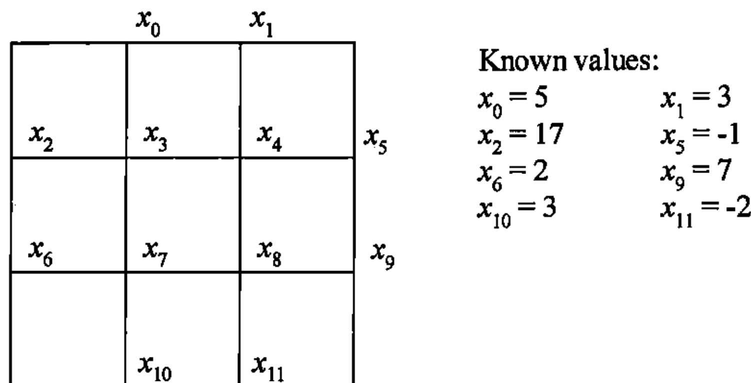


Figure 6.1 Test case 3

Test case 3 is built as a grid of 4 by 4 nodes. The values of the nodes on the edge of the grid are known. The value of a node inside the grid is unknown but is calculated as a mean of the values of the neighbouring nodes. Figure 6.1 shows the grid, the unknowns assigned to each node and the values for the known nodes.

The unknowns for the grid given in Figure 6.1 are x_3, x_4, x_7 and x_8. The following system of equations can be built:

$$\begin{cases} 4x_3 = x_0 + x_2 + x_4 + x_7 \\ 4x_4 = x_1 + x_5 + x_3 + x_8 \\ 4x_7 = x_3 + x_8 + x_6 + x_{10} \\ 4x_8 = x_4 + x_7 + x_9 + x_{11} \end{cases}$$

This proves to be a slowly convergent problem, ideal for our experiments. Another advantage of this problem is that the grid can be enlarged as much as required without too much difficulty.

6.3.4 Test case 4

This test case is the largest one. It shows a gain in performance for both the two and three node cases. As can be noticed, the problem requires a great amount of computation time, denoted by the large number of contractions that have be performed.

	x_0	x_1	x_2	
x_3	x_4	x_5	x_6	x_7
x_8	x_9	x_{10}	x_{11}	x_{12}
x_{13}	x_{14}	x_{15}	x_{16}	x_{17}
	x_{18}	x_{19}	x_{20}	

Known values:

$x_0 = 9$	$x_1 = 2$	$x_2 = 11$
$x_3 = -3$		$x_7 = -6$
$x_8 = 7$		$x_{12} = 10$
$x_{13} = 1$		$x_{17} = 2$
$x_{18} = -4$	$x_{19} = 1$	$x_{20} = 7$

Figure 6.2 Test case 4

This test case is similar with the previous one except that a 5 by 5 grid is used. Figure 6.2 shows the used grid and also the values for the known nodes.

The unknowns for the grid given in Figure 6.2 are $x_4, x_5, x_6, x_9, x_{10}, x_{11}, x_{14}, x_{15}$ and x_{16}. The system of equations used to calculate their values is the following one:

$$\begin{cases} 4x_4 = x_5 + x_9 + x_0 + x_3 \\ 4x_5 = x_1 + x_4 + x_6 + x_{10} \\ 4x_6 = x_5 + x_{11} + x_2 + x_7 \\ 4x_9 = x_4 + x_{10} + x_8 + x_{14} \\ 4x_{10} = x_5 + x_{11} + x_9 + x_{15} \\ 4x_{11} = x_6 + x_{12} + x_{10} + x_{16} \\ 4x_{14} = x_9 + x_{15} + x_{13} + x_{18} \\ 4x_{15} = x_{10} + x_{16} + x_{14} + x_{19} \\ 4x_{16} = x_{11} + x_{15} + x_{17} + x_{20} \end{cases}$$

6.4 The CPI parameter

As shown by the test cases, successful `update_var()` messages are beneficial for reducing the time required to perform propagation. This suggests that as soon as a Worker finds a new value for a variable it should send `update_var()` messages to the other Workers that share the same variable. However, as was described at the end of section 4.8.1, for a distributed system it is desirable to reduce the communication between nodes and to increase the computation time between consecutive communications. Therefore, there are two contradictory requirements. On the one hand, `update_var()` messages should be sent as much as possible, not only because they lead to progress, but also because they produce shortcuts in the propagation process. On the other hand communicating too often, and reducing the ratio between computation and communication, reduces the performance of a distributed system.

The CPI (constraints per iteration) parameter is used to allow a compromise between the two contradictory elements. As described in section 5.2.5, CPI indicates how many constraints a Worker contracts before it sends `update_var()` to the other Workers.

The results indicated in Table 6-1 are for a CPI equal to 1. As explained in section 6.1 the experiments have been performed in a special context in which the communication time is about the same as the contraction time, so the CPI is not of much relevance. However, if the problem to be solved is very large, it is very likely that a distributed system in which the communication time is larger than the time required to perform a constraint contraction can also be used with success in solving it. In that case, CPI can be used to tune the system.

6.5 Conclusions

There are several conclusions that ca be drawn from analyzing the results produced by the four test cases indicated in Table 6-1 and also from performing some experiments without the special settings indicated in section 6.1:

- Successful `update_var()` messages are very important in reducing the time required to perform propagation. Not only that they assure progress, but they also act as shortcuts for the receiving Workers.
- However, for a distributed system, in which there is a large communication time compared with the contraction time, it is not appropriate to send `update_var()` messages after each constraint contraction.
- Slowly convergent problems benefit from a distributed propagation since they require more contractions to reach the final result.
- The larger the problem to be solved, the shorter the time a Worker is idle, and more likely a speed-up will be obtained.
- In order to have a real gain in performance when using a larger number of computation nodes, there is a need for a system for which the communication time is comparable with the time required to contract a constraint.

7 Conclusions

This chapter presents the conclusions of the research carried out for this thesis. Firstly, the motivation of this research is presented, followed by the major accomplishments. The chapter concludes by indicating some potential improvements and ways of extending the current work and its results.

7.1 Research motivation

Interval constraints is a numerical method that has remarkable properties with respect to the result yielded. However, there are many complex numerical problems for which the time required by intervals constraints to produce a result is relatively long. Although, there is still a positive aspect that consists of the fact that the way the primitive constraints must be activated is practically unconstrained, which makes the constraints propagation process suitable for parallelism.

On the other hand, parallelism, in the form of some cheap hardware interconnected through some kind of cheap high-speed communication medium, has been proved to be a very attractive and ubiquitous solution for solving complex problems, for both economical and performance reasons.

The goal of this research was to present a flexible parallel architecture for an interval constraints propagator that can be implemented on a cheap distributed system, motivated by the intrinsic suitability for parallelism that constraints activation have, and by the gain in performance provided by a parallel approach.

7.2 Accomplishments

This thesis presents a distributed architecture for solving the constraint propagation process based on coordination. The presented model is based on a set of communicating agents that cooperatively participate in the process of solving the constraint propagation.

One of the main advantages that the proposed model has is flexibility. This characteristic results from the fact that the architecture is a modular one, based on agents that encapsulate their data and communicate through a message passing mechanism. The message passing mechanism is a very flexible and elegant model that can be used to write parallel programs not only for a real distributed memory model, but also for a system that underneath uses a shared memory model. This allows that the proposed architecture be implemented on different kind of concrete systems. More specifically, in addition to the main target, that is, a distributed system composed of a set of workstations connected through a communication network, it is possible to have implementations, following the same model, for other kind of systems, such as a tightly-coupled parallel system that emulates a message passing model.

The thesis also presents a concrete implementation of the proposed model that runs on a distributed system, consisting of a set of workstations, running Linux, and connected through an Ethernet network. The core functionality specified by the architecture is implemented in C++ as a collection of passive objects. The passive objects are used by a set of agents that communicate and coordinate their activity using facilities provided by COOL. The whole constraint propagation process is solved as coordination between the agents.

The experimental results, for the distributed system used as a test platform, show that the parallel approach is worthy for large problems, in which the time Workers are idle is reduced. This requirement is a result of the fact that the communication time required to send an update message from a Worker to another one is much greater than the time required to perform a contraction. If the problem is large, the penalty introduced by communication is compensated since the Workers are busy anyway with doing contractions. Therefore, the optimal preferred situation is the one in which a Worker is

busy most of the time performing contractions on a large set of constraints and, the time it waits, blocked, for update messages from other Workers, is a lot less, ideally null. Moreover, it is desirable that the communication time required to send an update message for a variable be comparable with the time required to perform a few contractions. In order to have a speed-up, for an implementation of the model described in this thesis for a distributed system, the ratio between computation and communication has to be sufficiently large (CPI parameter). Finally, the experiments show that slowly convergent problems benefit from a parallel approach, since they require many contractions to be solved, that is, long computation time.

On the other hand, the experimental results presented in chapter 6 show that update messages are beneficial in reducing the propagation time by acting as shortcuts for the propagation. However, as indicated in the previous paragraph, for a distributed system, to accomplish a speed-up, it is desired to have a large ratio between computation and communication. These two conclusions are contradictory and might suggest that a multiprocessor system is more appropriate for a parallel constraint propagation. However, today's hardware is changing very fast and there are already fast telephone switches, interconnected through high-speed optical fibers, for which the computation time is comparable to the communication time. Therefore, the proposed model is valuable for such kind of systems.

Another significant conclusion, indicated by measurements also, shows that grouping together the primitive constraints that are produced by the same complex constraint is also very important when a constraint system is partitioned in subsystems. The groups that are formed must not be broken and must be assigned to the same partition. In this way, no additional communication between Workers is introduced when shared variables need to be updated. Moreover, clustering combined with a greedy strategy for assigning the clusters to partitions lead to the creation of some optimal partitions for which the cost of communication is reduced, as opposed to the situation when the partitions are randomly created.

7.3 Future work

As presented in section 2.4 solving a constraint system implies an alternation of constraint propagation and splitting. However, the solution presented in this thesis includes only propagation. Therefore, a natural next step would be to extend the current architecture with splitting capabilities. The modular aspect of the proposed architecture should make the addition of splitting a reasonable task in terms of complexity. A possible solution would be to have a new Split agent that would implement the functionality required to perform splitting.

The propagation process assumes a set of primitive constraints. However, a problem is normally described through a set of complex constraints. Therefore, the complex constraints have to be translated into a set of corresponding primitive constraints that are used by the propagator. This process of translation can be done manually, but for large and complex systems of equations it becomes tedious and error prone. Thus, an appropriate module that does the translations would be beneficial. The translator could be implemented as a parser that takes as input a complex constraint as string and generates as output a list of primitive constraints. In this way, the primitive constraints that are yielded by a complex constraints can be grouped together, situation preferred, as presented in section 4.10, in order to avoid additional overhead introduced by updating shared variable.

As indicated by the experimental results, the system performs better if the communication between Workers is minimized. This goal can be achieved while the constraint system is partitioned in subsystems, by creating some partitions between which the number of shared variable, that is, the messages required for updates, is minimized. In the current implementation, the partitions are created using a greedy strategy combined with clustering. This local optimization method produces good results, but may not produce the best ones. A possible improvement would be to envisage a system that always produces the most optimal partitions, that is, the ones in which the number of shared variables is minimum.

Glossary

Active constraint	A *constraint* for which its associated contraction operator has to be applied or reapplied.
Active entity	An entity that can execute *concurrent*ly and *asynchronous*ly of other entities. For example, a *process*, a *thread*, an *agent*, or an *active object*.
Active object	An *object* that has its own *thread* of control and that can execute *independent*ly of other active objects.
Actor	A concrete implementation in COOL of an *agent* from the *conceptual model*.
Agent	In the context of this thesis an agent is an *active object* that is part of the *conceptual model*.
Channel	A way of designating the source and the destination of a message, in a *message passing* scheme, used in Occam [20]. A channel connects two and only two *processes*, providing a one-to-one communication, that is, a single process can send to the channel and a single process can receive from the channel.
Chaotic iteration	A general mathematical framework that formalizes *constraint propagation* algorithms used in solving several kinds of *constraint satisfaction problem*s.
Complex constraint	A *constraint* which is not *primitive*, and which usually specifies a more complicated mathematical relation, but which can be

	decomposed in a set of *primitive constraints*. See also *constraint* and *primitive constraint*.
Computation node	Also shortly called just node. A machine or a processor, equipped with computation power. For this application a node has to be equipped with a floating-point unit required for the computations that have to be performed in the process of solving a *constraint system*.
Concept	An abstraction of a set of phenomena (ideas or things) which have common properties.
Concurrency	Some authors consider concurrency as a synonym for *parallelism*. However, it might be useful to see it not as execution of source program text, but execution of machine instructions. Compare *parallelism*.
Consistent	A *constraint system* is consistent if all its *unknowns* have *domains* for which all *constraints* are satisfied. Compare *inconsistent*.
Constraint	A mathematical relation (an equality or an inequality) that involves a number of *unknowns*, and that specifies a restriction that the *unknowns* must satisfy. There can be *primitive* or *complex constraints*.
Constraint propagation	The procedure of solving a *constraint system* by repeatedly applying the *contraction operator* on each *constraint* until there are no more changes of the intervals associated with the *unknowns* of each *constraint*.
Constraint satisfaction problem	A CSP consist of a set of *unknowns* that are part of a problem, a set of associated *domains* of values, and set of *constraints*, each on a subset of the whole set of *unknowns*. A CSP has the goal of finding values for the set of *unknowns*, which simultaneously satisfy the set of *constraints*.

Constraint subsystem — Synonym for *partition.*

Constraint system — The basis used by the interval method in the process of solving a problem. A constraint system consists of a set of *primitive relations*, a set of *constraints*, a set of *unknowns*, and a state.

ConstraintSystem — An *agent* that encapsulates a particular *constraint system.* It takes care of partitioning it, of distributing it to the *Worker agents* and eventually of collecting the results.

CPI — *Constraints Per Iteration.* Represents the number of *constraints* a *Worker* contracts before it sends update messages for the changed *variables* to the corresponding *Workers*.

CSP — See *Constraint satisfaction problem.*

Contraction operator — An operator associated with each *primitive constraint* used to contract the intervals associated with the *unknowns* that are part of the *constraint.*

Data parallelism — A *parallel* programming paradigm that exploits the *concurrency* that derives from applying the same operation on multiple elements of the same data structure. In its simplest form the data parallel programming model extends a sequential programming language with *parallel* constructs for handling large aggregates of data such as arrays. It is specific for the *SIMD* model.

Distributed memory — A *MIMD parallel* processing *model* composed of a set of processors, each having its own private memory as opposed to one large memory pool for all processors, as in the *shared memory model.*

Distributed — See loosely coupled system.

Domain — The set of values that an *unknown* can take, so that the *constraints* the *unknown* is part of are still satisfied. In the strict

	context of the *interval constraints* method it is the interval associated with the *unknown*.
EREW	*Exclusive Read Exclusive Write*. A subclass of the *PRAM model*, in which only one exclusive operation of reading or writing is permitted to a particular location of memory address at a given time.
Inconsistent	A *constraint system* is inconsistent if it has at least one *constraint* whose *unknowns* have *domains* of values for which the *constraint* is unsatisfied. Compare *consistent*.
Interval	A continuous set of ordered values delimited by a lower and an upper bound.
Interval constraint system	See *constraint system*.
Independent	Executing independently means executing *concurrently* and *asynchronously* with the rest. It refers to the execution of a *process*, *thread*, or any other *active entity*.
Lightweight process	Also known as *thread*. A process that has its own registers, stack, and program counter, but which shares other data, such as the heap and file descriptors, with other *threads*. See also *process*.
Loosely coupled system	Also know as *multicomputer* or *distributes system*. In contrast to the *tightly coupled system*, the processors of this system do not share memory or a common clock. Instead, each processor has its own memory and communicates with other processors through different kind of communication lines, such as high-speed networks or telephone lines. Compare *tightly coupled system*.
Message passing	A *parallel* computing *paradigm* in which a problem is solved

by a set of cooperating *process*es for which the only way to communicate and synchronize is sending each other messages.

MIMD — *Multiple Instructions Multiple Data.* An architecture consisting of several processors capable of executing individual instruction streams on different sets of data. Compare *SIMD.*

Model — An abstraction of a system that contains only essential details, and which is aimed at simplifying the reasoning about the system.

Multicomputer — See *loosely coupled system.*

Multiprocessor — See *tightly coupled system.*

Monitor — An *agent* that initializes, starts, monitors, and decides when the results of the *constraint propagation* must be collected.

Node — Short name for *computation node.*

Object — An instance of a class that has an identity and encapsulates attributes and behaviour. Informally, an example of a *concept.*

Paradigm — Synonym for *model.*

Parallelism — In the context of this thesis, refers to computation performed *concurrently*, regardless if the performing system is a *multiprocessor* or a *multicomputer*, of a textual representation of a *program.* Some authors use it just to refer to the computation performed by a *multiprocessor* system. See also *concurrency* and *distributed.*

Partition — In the context of *parallelism*, a *constraint system* must be split in several parts called *partition*s or *constraint subsystem*s.

Passive object — An *object* that has no *thread* of control.

PRAM — *Parallel Random Access Machine.* An idealized *model*, often used in theoretical studies of *parallel* algorithms, in which any

processor can access any memory location in the same amount of time.

Primitive constraint — A simple mathematical relation that has a *contraction operator* associated. See also *constraint* and *complex constraint.*

Process — Also known as *heavyweight process.* The execution of a *program.* A *process* might require internal *concurrency,* need satisfied by *lightweight processes* or *threads.* See *program* and *lightweight process.*

Program — A computational activity written in a programming language and represented as text.

Propagator — A class that encapsulate and is able to solve a *partition* of a *constraint system.*

r^- — The greatest floating-point number not greater than r.

r^+ — The smallest floating-point number not smaller than r.

Scalable — A system is scalable if it is capable of delivering an increase in performance proportional to an increase in size (resources).

Shared memory — A *MIMD parallel* processing *paradigm* characterized by having multiple processors that exchange information by reading and writing a globally shared random access memory space. Compare *distributed memory.*

Solution — A set of values, one for every *unknown,* such that every *constraint* is satisfied. See *unknown* and *constraint.*

SIMD — *Single Instruction Multiple Data.* A widely accepted way of classifying computer architectures based on the instruction streams and data streams a specific hardware handles. A SIMD *model* is composed of many processors, each with its own memory. Each processor executes the same instruction simultaneously on its set of data. Compare *MIMD.*

Sound	In the context of *interval constraints* it refers to the properties of the interval method. It indicates that the solution provided by the interval method is correct and complete.
State	The fourth component of an *interval constraint system*. This is the set of *intervals* associated with the *unknowns*. The state indicates the information about the *solution*, by specifying a value, as an *interval*, for each *unknown*. See also *constraint system*.
Task	Some authors use task as synonym for *process*. However, in this thesis a task has its primary meaning of an assigned piece of work (activity) that has to be performed.
Thread	See *lightweight process*.
Tightly coupled system	Also know as *multiprocessor* systems. Such a system has more than one processor that are closely connected and that share the computer bus, the memory and the peripheral devices. Compare *loosely coupled system*.
Transputer	A single chip that has a single processor, its own memory, and four communication links. It can be networked directly with other transputers and used very efficiently for massively *parallel* programs.
Tuple	The components of the Linda [17] memory *model* (called tuple space). There two kinds of tuples: process tuples under active evaluation and data tuples that are passive.
Unknown	A placeholder for one value. The *interval constraints* method seeks a value for it. Occurs in a *constraint*. See *solution*. Compare *variable*.
Variable	Occurs in a computer *program*; its value might vary even if it might not do so. Compare *unknown*.

Vector processing	Is a style of *program* execution that exploits the large potential for *concurrency* present in loops. A vector processor provides special hardware for operating on vectors of data.
Vectorizing compiler	A compiler which locates expressions within a loop that can be evaluate for the entire iteration space at once, by reordering statements in a loop so that such execution is equivalent to the initial code.
Worker	An agent that performs the actual constraint propagation on a partition of a constraint system.

Bibliography

[1] Huan Wu. Defining and Implementing a Unified Framework for Interval Constraints and Interval Arithmetic. Master thesis, Department of Computer Science, University of Victoria, May 1999.

[2] Frederic Benhamou and William Older. Applying interval arithmetic to real, integer and boolean constraints. Journal of Logic Programming, 1994: 19, 20: 1-679.

[3] Ernest Davis. Constraint Propagation with Interval Labels. Artificial Intelligence, 32(3): 281-331, July 1987.

[4] Krzysztof R. Apt. The Essence of Constraint Propagation. Theoretical Computer Science, 221(1-2): 179-210, 1999.

[5] Eric Monfroy and Jean-Hugues Réty. Chaotic Iteration for Distributed Constraint Propagation. Proceedings of The 14th ACM Symposium on Applied Computing, SAC'99 Artificial Intelligence and Computational Logic Track, San Antonio, Texas, USA, March 1999, ACM Press.

[6] Eric Monfroy and Jean-Hugues Réty. Itérations Asynchrones: Un Cadre Uniforme pour la Propagation de Contraintes Parallèle et Répartie. Proceedings of Journées Franchophones de Programmation Logique et Contrainte, JF-PLC'99, Lyon, France, 1999, (in French).

[7] Eric Monfroy. A Coordination-based Chaotic Iteration Algorithm for Constraint Propagation. ACM SAC '2000 Como, Italy.

[8] A. K. Mackworth. Constraint Satisfaction. In Encyclopedia of Artificial Intelligence, 2nd edition, Wiley and Sons, 276-285, 1992.

[9] M. Yokoo, E. Durfee, T. Ishida, K. Kuwabara. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. In Proceedings of the Twelfth International Conference on Distributed Computing Systems, 614-621, 1992.

[10] A. Barr, E. A. Feigenbaum. The Handbook of Artificial Intelligence. Vol. 1, Morgan Kaufmann, 1981.

[11] Ying Zhang, Alan K. Mackworth. Parallel and Distributed Algorithms for Finite Constraint Satisfaction Problems. In Proceedings of SPDP'91, 394-397, 1991.

[12] Ying Zhang, Alan K. Mackworth. Parallel and Distributed Finite Constraint Satisfaction: Complexity Algorithms and Experiments. Technical Report 92-30, Department of Computer Science, University of British Columbia, Vancouver Canada, 1992.

[13] Pascal van Henterynck, Laurent Michel, and Yeves Deville. Numerica: A Modeling Language for Global Optimization. MIT Press, 1997.

[14] IEEE standard for Binary Floating-Point Arithmetic. Standard 754-185, ANSI/IEEE, New York, 1985.

[15] Timothy Hickey, Q. Ju, and Maarten van Emden. Interval Arithmetic: from Principles to Implementation. Technical Report DCS-260-IR, Department of Computer Science, University of Victoria, 1999.

[16] Lewis Rewini. Introduction to Parallel Computing. Prentice Hall, 1992.

[17] Nicholas Carriero and David Gelernter. How to Write Parallel Programs - A First Course. MIT Press, 1990.

[18] J. Goodman, M. Verson, and P.Worst. Efficient Synchronization Techniques for Large-Scale Cache-Coherent Multiprocessors. Proc. 3rd International Conference on Architectural Support Programming Languages and Operating Systems, California, 1988.

[19] H. E. Bal, M. Frans Kaashoek and Andrew Tanenbaum. ORCA: A Language for Parallel Programming of Distributed Systems. IEEE Transactionson Sofware Engineering, vol. 18, No. 3, Mar.1992, pp.190-205.

[20] David May. Occam. ACM SIGPLAN Notices, vol. 18, no. 4, Apr.1983, pp.69-79.

[21] C. A. R. Hoare. Communicating Sequential Processes. Communication of ACM, vol. 21, No.8, Aug. 1978, pp. 666-677.

[22] Message Passing Interface Forum. http://www.mpi-forum.org/.

[23] Ian Foster. *Designing and Building Parallel Programs.* http://www-unix.mcs.anl.gov/dbpp/.

[24] Mantis H. M. Cheng. COOL: A Crisp object coordination language. July 11, 1998.

[25] Gordon O'Connell. COOL Compiler User Manual. Report, Department of Computer Science, University of Victoria, May 4, 2000.

[26] Gordon O'Connell and Mantis H. M. Cheng. A COOL Language Report. Report, Department of Computer Science, University of Victoria, March 6, 2000.

[27] Craig Larman. Applying UML and patterns, An Introduction to Object-Analysis and Design. Prentice Hall, 1998.

[28] Bernd Bruegge and Allen H. Dutoit. Object-Oriented Software Engineering, Conquering Complex and Changing Systems. Prentice Hall, 2000.

[29] Jeff Magee and Jeff Kramer. Concurrency - State Models & Java Programs. Willey, 1999.

[30] Nancy A. Lynch. Distributed Algorithms. Morgan Kaufmann Publishers 1997.

[31] Mohamed S. Abdallah. JIC: An Interval Constraints Extension to Java. Master thesis, Department of Computer Science, University of Victoria, March 2000.

[32] Patrick Donovan Dowler. Interval Arithmetic and Constraints Systems: Java Implementation and Applications. Master thesis, Department of Computer Science, University of Victoria, April 2000.

[33] Maarten H. van Emden. Computing functional and Relational box Consistency by Structured Propagation in Atomic Constraint Systems. Report DCS-266-IR, Department of Computer Science, University of Victoria, 2001.

[34] Gerard Tel. Introduction to Distributed Algorithms. Cambridge University Press, 1994.

[35] Edsger W. Dijkstra. Shmuel Safra's version of termination detection. Note EWD-998, 1987.

[36] Hesham El-Rewini and Ted G.Lewis. Distributed and parallel Computing. Manning, 1998.

[37] Alexander Morgan. Solving Polynomial Systems Using Continuation for Engineering and Scientific Problems. Prentice-Hall, 1987.

Appendix A Source code

This appendix contains the source code for the most relevant parts of the system. First part contains the header files for the passive objects. The second part contains the source code for the actors.

A.1 The passive objects

This section contains the header files for the passive objects. The passive objects are described in section 5.2, and implement the core functionality of the constraint propagation.

A.1.1 The `Variable` class header file

```
#ifndef VARIABLE_H
#define VARIABLE_H

#include "Constraint.h"
#include "real.h"
#include "cptypes.h"
#include "Propagator.h"
#include <set>
#include <stack>
#include <deque>

class Constraint;
class Propagator;

class Variable
{
private:
	VariableId id;
	Real val;
	set<Constraint*> c;   // constraints that share this variable
	std::stack<Real> stack;
protected:
	Propagator *propagator;
	Variable() {}
public:
```

```
      Variable(VariableId, Propagator *prpg);
      Variable(VariableId, const Real&, Propagator *prpg);
      Variable(VariableId, const double, Propagator *prpg);
      Variable(VariableId, const double, const double,
            Propagator *prpg);
      virtual ~Variable() {}

      VariableId getId() { return id; }
      Real getValue() { return val; }
      double getLB() { return val.getLB(); }
      double getUB() { return val.getUB(); }
      set<Constraint*>& getConstraints() { return c; }

      bool update(const double lb, const double ub);
      bool update(const Real &v);

      void addConstraint(Constraint* con);
      void delConstraint(Constraint* con);
      int getNoRefCons() { return c.size(); }
            // gets the number of refering Cons

      void store();
      void restore();
};

#endif
```

A.1.2 The SharedVar class header file

```
#ifndef SHAREDVAR_H
#define SHAREDVAR_H

#include "real.h"
#include "cptypes.h"
#include "Variable.h"
#include <vector>

class SharedVar : public Variable
{
private:
      vector<NodeId> nodes;   // ids of nodes that share this variable
      SharedVar() {}
public:
      SharedVar(VariableId i, Propagator *p) : Variable(i,p) {}
      SharedVar(VariableId i, const Real& v, Propagator *p)
            : Variable(i,v,p) {}
      SharedVar(VariableId i, const double lu, Propagator *p)
            : Variable(i,lu,p) {}
      SharedVar(VariableId i, const double lb, const double ub,
            Propagator *p)
            : Variable(i,lb,ub,p) {}

      void addNode(NodeId n) { nodes.push_back(n); }
      vector<NodeId>& getNodes() { return nodes; }
```

```
      bool local() { return nodes.size() == 0; }
};

#endif
```

A.1.3 The Constraint classes header file

```
#ifndef CONSTRAINT_H
#define CONSTRAINT_H

#include "Variable.h"
#include "cptypes.h"
#include <set>

class Variable;

class Constraint
{
public:
      virtual ~Constraint() {};
      virtual bool shrink(set<Variable*>&) = 0;
            // if shrink succeeds set contains the changed variables.
      virtual void getVariables(set<Variable*>&) = 0;
      ConstraintId getId() { return id; }
protected:
      ConstraintId id;
};

/**
 * Primitive constraint of type x + y = z
 */
class Sum : public Constraint
{
private:
      Variable *x, *y, *z;
public:
      Sum (ConstraintId, Variable*, Variable*, Variable*);
      ~Sum();
      bool shrink(set<Variable*>&);
      void getVariables(set<Variable*>&);
};

/**
 * Primitive constraint of type x * y = z
 */
class Prod : public Constraint
{
private:
      Variable *x, *y, *z;
public:
      Prod(ConstraintId, Variable*, Variable*, Variable*);
      ~Prod();
      bool shrink(set<Variable*>&);
      void getVariables(set<Variable*>&);
};
```

```
/**
 * Primitive constraint of type x <= y
 */
class Leq : public Constraint
{
private:
      Variable *x, *y;
public:
      Leq(ConstraintId, Variable*, Variable*);
      ~Leq();
      bool shrink(set<Variable*>&);
      void getVariables(set<Variable*>&);
};

/**
 * Primitive constraint of type x = y
 */
class Eq : public Constraint
{
private:
      Variable *x, *y;
public:
      Eq(ConstraintId, Variable*, Variable*);
      ~Eq();
      bool shrink(set<Variable*>&);
      void getVariables(set<Variable*>&);
};

/**
 * Primitive constraint of type x^n = y
 */
class PowerN : public Constraint
{
private:
      Variable *x, *y;
      int n;
public:
      PowerN(ConstraintId, Variable*, Variable*, int);
      ~PowerN();
      bool shrink(set<Variable*>&);
      void getVariables(set<Variable*>&);
};

#endif
```

A.1.4 The `Propagator` class header file

```
#ifndef PROPAGATOR_H
#define PROPAGATOR_H

#include "Variable.h"
#include "Constraint.h"
#include <set>
#include <stack>
```

```
#include <deque>

class Variable;
class Constraint;

class Propagator
{
private:
      set<Variable*> v;          // the set of variables
      set<Constraint*> c;        // the set of constraints
      deque<Constraint*> qac; // the queue of active constraints
      stack<deque<Constraint*> > qacStack;

      Constraint *crtActiveCon;
      bool propagateChanges;
public:
      Propagator();
      ~Propagator();

      void addVariable(Variable *var) { v.insert(var); }
      void addConstraint(Constraint *con);
      void delConstraint(Constraint *con);

      void activateAll();
      void variableChanged(set<Constraint*> &cons);

      void store();
      void restore();

      bool shrink(unsigned long maxIter, unsigned long &actualIters,
            set<Variable*> &changedVars);

      bool done() { return qac.empty(); }

      bool probe(Variable *var, bool propagateChanges);
private:
      bool tryLeft(int fraction, Variable *var);
      bool tryRight(int fraction, Variable *var);
};

#endif
```

A.1.5 The PPropagator class header file

```
#ifndef PPROPAGATOR_H
#define PPROPAGATOR_H

#include "SharedVar.h"
#include "Constraint.h"
#include "cptypes.h"
#include "Propagator.h"
#include <map>
#include <vector>
#include <deque>
#include <iostream>
```

```
class PPropagator
{
private:
      map<VariableId, SharedVar*> v;    // map[variableId,variable]
      map<ConstraintId, Constraint*> c; // map[constraintId,constraint]
      set<SharedVar*> cv;               // set of changed variables
      SharedVar* ccv;                   // current changed variable
      vector<NodeId>::iterator cn;      // current node

      Propagator p;
      unsigned long cpi;                // contractions per iteration
public:
      PPropagator(unsigned long cpi) { this->cpi = cpi; }
      ~PPropagator();

      bool addVariable(VariableId);
      bool addVariable(VariableId, const double);
      bool addVariable(VariableId, const double, const double);

      bool addSum(ConstraintId, VariableId, VariableId, VariableId);
      bool addProd(ConstraintId, VariableId, VariableId, VariableId);
      bool addLeq(ConstraintId, VariableId, VariableId);
      bool addEq(ConstraintId, VariableId, VariableId);
      bool addPowerN(ConstraintId, VariableId, VariableId, int);

      bool addNodeForVariable(NodeId, VariableId);

      void activateAll();
      bool shrinkDone() { return p.done(); }
      bool shrink(bool probe);

      bool updateVariable(VariableId, const double, const double);

      SharedVar* getVariable(VariableId);
      double getLBVariable(VariableId);
      double getUBVariable(VariableId);

      bool hasChangedVariables() { return cv.size() > 0; }
      VariableId getNextChangedVariableId();
      double getUBChangedVariable();
      double getLBChangedVariable();

      bool hasNodes();
      NodeId getNextNodeId();
};

#endif
```

A.1.6 The ICSParser class header file

```
#ifndef ICSPARSER_H
#define ICSPARSER_H

#include "ICSTypes.h"
```

```
#include <vector>
#include <string>

class ICSParser
{
public:
      void parse(const char fName[], vector<Con>& c, int& noClusters,
            vector<Var>& v, vector<int>& varsToColl);
      ICSParser();
      ~ICSParser();

private:
      bool readVar(const string &s, Var &v);
      bool readCon(const string &s, Con &c);
      bool readVarsToColl(const string &s, vector<int> &v);
      void errorAtRead(const char* fName, int lineNo);

      int crtCluster;
      int crtId;
};

#endif
```

A.1.7 The ICSPartitioner class header file

```
#ifndef ICSPARTITIONER_H
#define ICSPARTITIONER_H

#include "ICSTypes.h"
#include <vector>
#include <algorithm>
#include <map>
#include <set>

class ICSPartitioner
{
public:
      void partition(vector<Con>& c, vector<Var>& v, int noPartitions);
      int cost();
      ICSPartitioner();
      ~ICSPartitioner();
private:
      struct SimpleVar
      {
            int id;
            set<int> partitions;// the partitions sharing this variable
            SimpleVar() {}
      };

      void assignVarToPartition(int vId, int partition);
      void addCon(Con c, int toPartition);
      void restore();
      void store();

      // the current state of the system;
```

```
	// used to calculate a possible new best state
	set<Con> c1;             // the set of constraints
	map<int,SimpleVar> v1;   // the set of variables

	// the best partitions found so far;
	// used to restore the state of the system
	set<Con> c0;             // the set of constraints
	map<int,SimpleVar> v0;   // the set of variables
};

#endif
```

A.1.8 The `ICSDistributor` class header file

```
#ifndef ICSDISTRIBUTOR_H
#define ICSDISTRIBUTOR_H

#include "ICSTypes.h"
#include <vector>

class ICSDistributor
{
public:
	ICSDistributor();
	~ICSDistributor();
	void distribute(vector<Var> &v, vector<Con> &c,
vector<NodeForVar> &nfv);
};

#endif
```

A.1.9 The `ICSTypes` header file header file

```
#ifndef ICSTYPES_H
#define ICSTYPES_H

#include "real.h"
#include <set>

struct Con
{
	int id;
	int type;
	int v1Id, v2Id, v3Id;
	int n;
	int cluster;
	int partition;

	Con() { id=type=v1Id=v2Id=v3Id=n=cluster=partition= -1; }
	Con(int id, int type, int v1Id, int v2Id, int v3Id, int n,
		int cluster, int partition)
	{
		this->id = id;
		this->type = type;
```

```
		this->v1Id =v1Id;
		this->v2Id =v2Id;
		this->v3Id =v3Id;
		this->n = n;
		this->cluster = cluster;
		this->partition = partition;
	}

	friend bool operator<(const Con& c1, const Con& c2) {
		return c1.id<c2.id;
	}
};

struct Var
{
	int id;
	Real val;
	bool isConst;
	set<int> partitions;

	Var() {}
};

struct NodeForVar
{
	int node;
	int var;
	int destNode;
	NodeForVar(int n, int v, int d) { node=n; var=v; destNode=d; }

#endif ICSTYPES_H
```

A.2 The active objects

This section contains the source code for the actor interfaces.

A.2.1 The `Monitor` actor interface

```
actor class Monitor
{
Start:
	message INIT(string m);
RegisterToMom:
	message register_as_monitor();
	message register_ack(bool succeded);
Subscribe:
	message subscribe(Worker w);
SetNodeAddrs:
	message set_node_addrs();
	message set_node_addr_no_ack(int wId);
	message set_node_addr_ack(int wId);
StartDistribution:
```

```
      message start_distribution();
      message start_distribution_ack();
WaitForEndOfDistribution:
      message distribution_done();
WatchPropagation:
      message distribution_done();
      message start_prop();
      message start_prop_no_ack(int wId);
      message start_prop_ack(int wId);
      message set_node_status(int wId, bool s);
      message propagation_done();
CollectResults:
      message start_collect_results();
      message start_collect_results_ack();
      message done_collect_results();
DismissWorkers:
      message done_collect_results();
      message dismiss_workers();
DeregisterAsMonitor:
      message deregister_as_monitor();
      message deregister_ack(bool succeded);
      message kill();
}
```

A.2.2 The Worker actor interface

```
actor class Worker
{
Start:
      message INIT(string m);
SearchMonitor:
      message lookup_monitor(int n);
      message lookup_ack(actor a);
SubscribeMonitor:
      message subscribe_to_monitor();
      message subscribe_ack(int i);
ReceiveConstraintsSystem:
      message set_node_addr(int nodeId, Worker addr);
      message addVar(int varId, real lb, real ub);
      message addSum(int conId, int v1Id, int v2Id, int v3Id);
      message addProd(int conId, int v1Id, int v2Id, int v3Id);
      message addLeq(int conId, int v1Id, int v2Id);
      message addEq(int conId, int v1Id, int v2Id);
      message addPowerN(int conId, int v1Id, int v2Id, int n);
      message add_node_for_variable(int nodeId, int varId);
      message start_prop();
Propagate:
      message start_prop();
      message propagate();
      message update_var(int vId, real lb, real ub);
      message send_node_status(bool s);
      message set_node_status_ack();
      message get_var(int vId);
      message kill();
}
```

A.2.3 The `ConstraintSystem` actor interface

```
actor class ConstraintsSystem
{
Start:
      message INIT();
GetNodeAddrs:
      message set_node_addr(int nodeId, Worker addr);
      message start_distribution();
DistributeNodeAddrs:
      message start_distribution();
      message distribute_node_addrs();
      message resend_node_addr(int i, int j);
      message set_node_addr_ack(int ackNo);
DistributeVariables:
      message distribute_variables();
      message variable_distributed_ack(int ackNo);
DistributeConstraints:
      message distribute_constraints();
      message constraint_distributed_ack(int ackNo);
DistributeNodesForVariables:
      message distribute_nodes_for_variables();
      message node_for_variable_distributed_ack(int ackNo);
DistributionDone:
      message distribution_done();
      message distribution_done_ack();
StartCollectResults:
      message start_collect_results();
      message kill();
CollectResults:
      message start_collect_results();
      message collect_var(int vId, int nodeId);
      message get_var_ack(int vId, real lb, real ub);
EndCollectResults:
      message done_collect_results();
      message done_collect_results_ack();
}
```

Appendix B `Propagator` usage

As described in section 5.2.3, the Propagator class has a modular implementation that allows to be used in either a parallel or a sequential constraint propagation. This appendix contains the source code that shows how the Propagator can be used in a simple C++ program to perform a sequential constraint propagation.

B.1 The problem

The following simple system is used to show the usage of the `Propagator` class.

$$\begin{cases} (x+3)*(y-2)=1 \\ x*y=1 \end{cases}$$

B.2 The source code

```
#include "Propagator.h"
#include "Variable.h"
#include "Constraint.h"
#include "flpt.h"
#include "real.h"
#include <cassert>
#include <set>
#include <iostream>
#include <iomanip>

int main()
{
      set<Variable*> changedVars;
      set<Variable*> vars;
      set<Variable*>::iterator i;
      set<Constraint*> cons;
      unsigned long iterations = 0;
```

```
Propagator p;

Variable *x, *y, *x1, *x2, *x3, *x4, *x5;

cout << setprecision(18);

x  = new Variable(0,0,flpt::PosInf, &p);
y  = new Variable(1, &p);
x1 = new Variable(2, &p);
x2 = new Variable(3, &p);
x3 = new Variable(4,3.0, &p);
x4 = new Variable(5,-2.0, &p);
x5 = new Variable(6,1.0, &p);
vars.insert(x);
vars.insert(y);

p.addConstraint(new Sum(0,x,x3,x1));
p.addConstraint(new Sum(1,y,x4,x2));
p.addConstraint(new Prod(2,x1,x2,x5));
p.addConstraint(new Prod(3,x,y,x5));

p.activateAll();

cout << "\nJust Propagation:";

p.store();
p.shrink(1000,iterations,changedVars);

for (i=vars.begin(); i!= vars.end(); i++)
{
      cout << "\n v[" << (*i)->getId() << "] = ["
           << (*i)->getLB()
           << ", " << (*i)->getUB() << "]";
}

cout << "\n Terminated in " << iterations << " iterations";
p.restore();
iterations = 0;

cout << "\n\nPropagation and probing:";
cout << "\n - probing failed; ! probing succeeded\n ";
do {
      unsigned long iter;
      if ( ! p.shrink(1, iter, changedVars) )
      {
            cout << "\nEMPTY solution\n";
            exit(0);
      }
      for (i = changedVars.begin(); i!=changedVars.end(); i++)
      {
            if ( p.probe(*i, false) )
                  cout << "!";
            else
                  cout << "-";
      }
      changedVars.clear();
      iterations++;
```

```
        } while ( ! p.done() );

        for (i=vars.begin(); i!= vars.end(); i++)
        {
              cout << "\n v[" << (*i)->getId() << "] = ["
                   << (*i)->getLB()
                   << ", " << (*i)->getUB() << "]";
        }
        cout << "\n Terminated in " << iterations << " iterations";
        cout << endl;
}
```

B.3 The result

The program produces the following output:

```
Just Propagation:
 v[0] = [0.4364916731037084, 0.436491673103708511]
 v[1] = [2.29099444873580538, 2.29099444873580582]
 Terminated in 45 iterations

Propagation and probing:
 - probing failed; ! probing succeeded
 ----!-!!!!!!!!!!!!!!!!!!!!!!!!!!-!---
 v[0] = [0.4364916731037084, 0.436491673103708511]
 v[1] = [2.29099444873580538, 2.29099444873580582]
 Terminated in 36 iterations
```

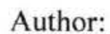
As can be noticed, probing helps at reducing the number of iterations required to reach the same result. An iteration represents the contraction of a single constraint.

University of Victoria Partial Copyright License

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain by the university of Victoria shall not be allowed without my written permission.

Title of Thesis:

Parallel Constraint Propagation

Author: 

Sinesie Calin Somosan

September 20, 2001