NEGOTIATING SOLUTIONS TO MULTIOBJECTIVE COMBINATORIAL OPTIMIZATION PROBLEMS

A Thesis Presented

by

Yönet A. Eracar

to

the Graduate School of Engineering

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Industrial Engineering

in the field of

Computer Systems Engineering

Northeastern University

Boston, Massachusetts

April 22, 2005

NORTHEASTERN UNIVERSITY

Graduate School of Engineering

Thesis Title: Negotiating Solutions to
Multiobjective Combinatorial Optimization Problems.

Author: Yönet A. Eracar.

Program: Computer Systems Engineering

Approved for Thesis Requirements of the Doctor of Philosophy Degree:

Thesis Advisor: Mieczyslaw M. Kokar	Date
Committee Member: Ken P. Baclawski	Date
Committee Member: Emanuel Melachrinoudis	Date
Committee Member: Robert Laddaga	Date
Chairman of the MIE Dept.: Hameed Metghalchi	Date

To my lovely wife Elif who was with me at every step of this adventure.

Abstract

This thesis addresses the problem of automatic synthesis of a satisficing self-controlling program for a multiobjective combinatorial optimization problem with respect to specification. Primary focus is on multiobjective optimization problems where the objectives conflict and there is no prior information on the relative importance or weights of the objectives. For such problems, a globally optimum solution may not exist or may be impossible to find.

In the absence of an algorithm that can find global optima, a satisficing technique is used. In this technique, first, the original multiobjective optimization problem is converted to many constraint satisfaction problems (CSP). Next, each CSP is solved individually by a CSP solver that uses a branch and bound algorithm. Then, the feasible solutions for each CSP are ordered by the respective objective functions. Finally, a negotiation algorithm is used to select one solution among these ordered solutions that will satisfy all the CSPs.

To support this technique, an agent-based software architecture is specified. For each objective, a different software agent is created. Each agent independently executes the branch and bound algorithm and reports the feasible solutions found to an entity called a “reconfigurer.” The reconfigurer mediates the negotiation among agents. Anytime a solution is demanded, the reconfigurer applies an imitative negotiation approach to choose the best-compromise solution among the ones that are reported up to that moment. The quality of the solutions improves as the system allows more time for the search.

UML/OCL is selected as the language in which problem specifications can be expressed. An automatic conversion mechanism for translating UML/OCL specifications to multiple CSPs is provided. A generic CSP code generator is implemented as a proof of concept for the conversion mechanism. For the proof of concept, OZ is selected as the target CSP programming language. The proof of concept system is tested against two experimental scenarios – a job scheduling problem and a fixture design problem – with each problem having conflicting objectives. For both problems, phase transition invariants are identified empirically and later used to generate hard problems that show phase transition behavior. The proof of concept system is executed against these hard problems, and the performance of system is measured for different control parameters and initial conditions. The critical control parameters, which the performance of the proof of concept is most sensitive to, are identified. The experiments with the proof of concept system show that satisficing programs can be automatically generated from specifications, that the complexity of the CSP solving algorithms can be controlled, and that the hard problems (phase transitions) can be detected with high probability and low probability of false alarm.

Acknowledgements

This Ph.D. thesis took a significant part of my life. Throughout my studies, I met many people, professors, friends, peers, and students that helped me with my work one way or another. I apologize for the ones I forgot to include here.

I like to thank,

Prof. Mitch Kokar, for his broad knowledge,
for his inputs from architectural decisions to implementation phase,
for always being positive, patient, encouraging, and
for guiding me to the right direction when I was about to loose focus,

Prof. Emanuel Melachrinoudis, for introducing the Operations Research field to me and for his timely corrections and comments on the thesis,
Dr. Robert Laddaga, for his help on the selection of the topic and the title for the thesis,
Prof. Ken Baclawski, for his invaluable feedback on the thesis,
Prof. Yaman Yener, for making my graduate studies possible,
from the application stage to the graduation date,

Evelyn Shore, for her help with the language and
the construction of the thesis,

Kivanç Aksoy and Askiner Güngör, for setting an example for the
presentation of a thesis,

Fikri Karaesmen, Ayse Kavusturucu, Sibel Tari, Zbigniew Korona, Stephen
Linder, Arda Aksu, and Erozan Kurtas, for showing me
how it should be done,

Aybek Korugan, Kendra Moore, Jennifer Black, Emre Karaoglu, Tunga
Ayik, for proving that it can be done, even after so many years,

Mehmet Dinçer and Omca Korugan, for their friendship,

Teresa Lopes, the Oracle, for convincing me to select the blue pill,
the XML technology,

Eugene Polyakov, for his brutal reviews,

Mike Goldstein, for introducing Evelyn Shore to me,

Tushar Gorel, for the printer,

all the friends at Teradyne, for their "emotional support",

and

Mom and Dad, for their love, prayers, the discipline they gave,
and anything that parents can provide for their child and more,

Thank you all with whole my heart.

Contents

Abstract **vi**

Acknowledgements **viii**

1 Introduction **1**

2 Literature Review **5**

2.1 Introduction 5

2.1.1 Multiobjective Optimization Problems 6

2.1.2 Combinatorial Optimization 9

2.1.3 Constraint Satisfaction Problems 10

2.1.4 Satisficing 16

2.1.5 CSP Solution Methods and Tools 21

2.1.6 Active Software 24

2.1.7 Negotiation Models 29

2.2 Analysis of Candidate Components 32

3 Proposed Solution **36**

3.1 Introduction 36

3.2 Defining Structural Constraints in UML 37

3.3 Defining Goals and Constraints in OCL 42

3.4 Ontology 43
3.4.1 New Types 43
3.4.2 New Operations 43
3.5 Translation of Goals and Constraints from UML / OCL to CSP Programming Language 47
3.5.1 CSP Code Generation 50
3.6 Architecture for Software Agents 55
3.6.1 Background 55
3.6.2 Components of the Architecture 57
3.6.3 Communication among Components 58
3.7 Formalization of the CSP Search Algorithm as a Dynamic System 63
3.7.1 A Branch and Bound Search Algorithm for Solving a CSP 63
3.7.2 General Dynamic Systems 67
3.7.3 Modeling the CSP Search Algorithm as a Dynamic System 68
3.7.4 Modifying the Model to Handle CSPs with Objective Functions 70
3.7.5 The Control of Search by an Agent 70
3.8 A Mediator-based Negotiation Algorithm 75

4 Verification and Demonstration **82**
4.1 Introduction 82
4.2 Job Scheduling System 84
4.2.1 Problem Statement 84
4.2.2 Problem Formulation 85

4.2.3 Goals and Constraints Presented in OCL 86
4.2.4 Empirical Calculation of the Phase Transition Regions 91
4.2.5 Experimental Setup 92
4.2.6 Analysis of the Results 94
4.2.7 Complete Results 101
4.3 Fixture Design Experiment 106
4.3.1 Problem Statement 106
4.3.2 Problem Formulation 106
4.3.3 Empirical Calculation of Phase Transition Regions 114
4.3.4 Experimental Setup 118
4.3.5 Analysis of the Results 118
4.3.6 Complete Results 120

5 Conclusions 124
5.1 Contributions 124
5.2 Conclusion 126
5.2.1 Future Research 129

A Source Code for the System 132
A.1 Agent.oz 132
A.2 Reconfigurer.oz 145
A.3 Template OCL File for Optimization Problems 162
A.4 Job Scheduling Benchmark Program 165
A.5 Fixture Design Benchmark Program 170
A.6 Source Code for the OCL Parser 173
A.6.1 Lex (.l) File for the OCL Parser 173

A.6.2 Keywords for the OCL Syntax 182

A.6.3 Yacc (.y) File for the OCL Parser 183

Bibliography **200**

List of Figures

3.1 Overview of the proposed solution 38
3.2 UML Class diagram for bank example 39
3.3 Translation of UML/OCL to the CSP programming language 47
3.4 Schema for OCLML 48
3.5 OCLML tree generated by the OCL Translator 51
3.6 Schema for the translation rules file 52
3.7 Self-controlling software model 56
3.8 Agent-based Self Controlling Software Architecture (SCSA) 57
3.9 State transition diagram for reconfigurer 59
3.10 Sequence diagram for the communication among a reconfigurer and its agents 61
3.11 Sequence diagram for the communication between an agent and the CSP solver 62
3.12 Flowchart for the CSP search algorithm 64
3.13 The structure of the general dynamic system 67
3.14 Extended CSP search algorithm modeled as a dynamic system 69
3.15 Search tree constructed by the branch and bound algorithm and the search parameters 71
3.16 PD control in SCSA 74

4.1 Class Diagram for the Job Scheduling System 84
4.2 Job scheduling experiment: PFA as a function of termination point and varying control parameters 96
4.3 Effects of changing goal values on PFA 99
4.4 Effects of changing the initial value κ_0 of the control variable on the values on PFA 100
4.5 Test results for the Job Scheduling experiment data set: $k_D = 0.3$, $k_P = 0.5$, $\kappa_0 = 0.5$, $goal = 1.0$ 102
4.6 Test results for the Job Scheduling experiment data set: $k_D = 0.1$, $k_P = 0.2$, $\kappa_0 = 0.0$, $goal = 1.0$ 103
4.7 Test results for the Job Scheduling experiment data set: $k_D = 0.025$, $k_P = 0.05$, $\kappa_0 = 0.0$, $goal = 1.0$ 104
4.8 Test results for the Job Scheduling experiment data set: $k_D = 0.175$, $k_P = 0.35$, $\kappa_0 = 0.0$, $goal = 1.0$ 105
4.9 Class Diagram for the Fixture for a Functional Board Test System 107
4.10 Effects of changing the channel card size on the complexity of the problem. 116
4.11 Fixture design experiment: PFA as a function of termination point and varying control parameters 119
4.12 Test results for the Fixture Design experiment data set: $k_D = 0.5$, $k_P = 1.0$, $\kappa_0 = 0.0$, $goal = 0.5$ 121
4.13 Test results for the Fixture Design experiment data set: $k_D = 0.175$, $k_P = 0.35$, $\kappa_0 = 0.0$, $goal = 0.5$ 122
4.14 Test results for the Fixture Design experiment data set: no control $\kappa_0 = 0.0$, $goal = 0.5$ 123

List of Tables

3.1 Listing for function GenerateCSPCode 53

3.2 Listing for function WalkOCLMLTree 53

3.3 Listing for function WalkUMLClass 54

3.4 Listing for function WalkOCLElement 54

3.5 Listing for function NegotiateSolution 75

3.6 Partial listing of the solutions for the optimization problem defined by $O_1 = 3 \cdot X + Y$ and $O_2 = X \cdot (10 - X) + Y$ . . 77

3.7 Listing for function FindNonDominatedSolutions 77

3.8 Listing for function TestSolutionForNonDominance 78

3.9 Non-dominated solutions for the optimization problem defined by $O_1 = 3 \cdot X + Y$ and $O_2 = X \cdot (10 - X) + Y$ 79

3.10 Listing for function CompareSolutions 80

3.11 Listing for function OrderNonDominatedSolutions 81

4.1 Probability of false alarm (PFA) as a function of termination point and varying control parameters: $goal = 1.0, \kappa_0 = 0.0, \tau = 500$ 97

4.2 Probability of false alarm (PFA) as a function of termination point and varying goal values: $k_P = 0.35, k_D = 0.175, \kappa_0 = 0.0, \tau = 500$ 98

4.3 Probability of false alarm (PFA) as a function of termination point and initial value of the control variable; κ_0, $k_P = 0.35, k_D = 0.175, goal = 1.0, \tau = 500$ 98

4.4 Empirical results for Phase Transition in the Fixture Design System problem 115

4.5 Probability of false alarm (PFA) as a function of termination point and varying control parameters: $goal = 0.5, \kappa_0 = 0.0, \tau = 500$ 120

Chapter 1

Introduction

Multiobjective optimization is a core area in engineering, business practice, and research. Application areas of multiobjective optimization include resource allocation, transportation, logistics, distribution, investment decisions, business planning with uncertain information, and others. Multiobjective optimization problems are formulated in terms of performance criteria (objective functions) and constraints. Optimization problems divide into two categories; those with continuous variables and those with discrete variables, which are also called *combinatorial optimization* problems.

Two examples of combinatorial optimization problems are design of a system and order negotiation in the manufacturing context. In design, components and a structure must be selected which "fit together" while the resulting system delivers the desired performance. The system must be energy efficient as well as relatively inexpensive. In order negotiation, the goal is to select the products to be manufactured, the amounts of each product, and the prices necessary to satisfy goals of both the manufacturers and the customers. While the manufacturer wants to maximize the profit and lead times, the customer is interested in minimizing the cost and the lead time. The constraints include manufacturing plant capabilities, storage capabilities, and other necessary items.

Finding a solution requires finding an instantiation of the problem variables that not only satisfy constraints, but also meet high value results in performance criteria. A way to search for an optimal solution is first converting the original problem into a number of constraint satisfaction problems (CSP) and then selecting a solution that is best from the point of view of optimality. When a CSP problem is NP-complete, full algorithmic optimal solutions cannot be expected. Moreover, since multiple performance criteria are involved, any solution requires tradeoffs among them. One has to settle for less than optimality. Conceptualizations of this kind of problem formulation are known as *satisficing* solutions or *good-enough-soon-enough* solutions. Solutions of this kind of problem typically use search as one of the components. However, since a generic search algorithm does not guarantee finding solutions within a finite time, the search incorporates features specific to a particular problem. In other words, the search is domain and problem specific.

As a consequence, a special program needs to be developed for any specific problem/domain. What if the problem formulation changes? Either a new program will need to be developed or the original program will need to have built-in mechanisms for adapting to changes. While the former applies to non-parametric changes in constraints and objective functions, the latter approach is appropriate when the changes in the constraints are parametric.

To solve the CSP problem described above, off-the-shelf CSP solvers could be used, but all of them require coding in a CSP language. A user-oriented and high-level language that will hide the implementation details related to the CSP solver is desirable [108]. The Unified Modeling Language (UML) is a combination of such abstract and graphical languages. The use of an abstract language like UML/Object Constraint Language (OCL) for domain modeling brings the necessity of a translator and code generator that will read the abstract language and generate the code for the target CSP solver. However, when automatic translation of problem specifications into CSP code replaces the human programmer, the automatically generated code may never terminate due to the complexity of CSP search. Therefore, the replacement of manual CSP coding by a UML/OCL interface requires a solution to the handling of the complexity of the search for a satisficing solution.

This thesis addresses the specific problem just mentioned above. How can a satisficing program for a multiobjective combinatorial optimization problem be automatically synthesized with respect to specification such that it can monitor and control its complexity? In order to develop such a program, a number of questions must be answered. What is the language in which problem specifications can be expressed? What algorithm or system can be used to search for solutions to multiobjective satisficing problems? What algorithms should be used to determine the aspiration levels (good-enough) for the objective functions in satisficing problems? How does the program monitor and control its computational complexity? What architecture should be used to implement the control of complexity?

This research investigates the use of UML to capture the structural aspects of the user's view of problem formulations; OCL to capture quantitative and global constraints; mapping of multiobjective optimization problems to an agent-based architecture in which each objective (goal) is "represented" by an agent; a general CSP solver to implement search; negotiation as a mechanism to determine the aspiration levels for each objective function; *phase transition invariants* for the purpose of identifying for computation intensive regions; and Self-Controlling Software Architecture to control the complexity of the search.

To evaluate the approach, an experimental system is implemented and tested using two scenarios, a fixture design utility and a job-scheduling system. Problem formulations are specified using the UML representation. Formulations are automatically translated into the selected CSP programming language and then used by the system to find satisficing solutions. The goal is to provide a proof-of-concept for developing such a self-controlling satisficing program that (1) is applicable to various multiobjective optimization problems and (2) has the ability to control its own complexity.

In both experimental scenarios, the system tests both hard and easy regions in order to show their ability to control their own complexity. The phase transition behavior is investigated. A model for phase transition (complexity as a function of the parameter characterizing the amount of change) is developed and then used to generate test data to test the proposed system. The quality of solutions to multiobjective optimization problems is also evaluated against known benchmark approaches.

Chapter 2

Literature Review

2.1 Introduction

The search for solutions starts with a short description of multiobjective optimization problems. The challenges in finding one solution that optimizes all objectives are discussed first. Combinatorial optimization problems, as a subcategory of general optimization problems, are introduced next. *Constraint satisfaction problems* (CSP) in general follow where an overview of existing CSP approaches are given. Since CSP algorithms are outside the scope of this thesis, focus rests on the CSP solvers, the evaluation of the performance of CSP solvers, and the *phase transition* phenomena. Next, the concept of satisficing as an alternative to optimization is discussed, a practice originating from Herbert Simon's [125] ideas on the conversion of optimization problems to constraint satisfaction problems when it becomes evident that the extra cost of finding an optimum solution exceeds the benefits of finding one. The satisficing approaches – *approximate reasoning*, *meta-reasoning* and *bounded optimality* – are presented, after which a list of CSP solution methods and tools are given. A discussion of *self adaptive* *software* and negotiation follows since it is related to the architecture proposed in this thesis. A comparison of the approach to the solution with a review of literature in the field closes the search.

2.1.1 Multiobjective Optimization Problems

Decision making for single-objective optimization has been well-studied. The problem becomes more difficult when a consideration of several conflicting objectives is required. In many cases, it is unlikely that different objectives would be optimized by the same choices of decision variables. Therefore, a trade-off between the objectives is needed to ensure a satisfactory solution. This type of problem is known as either a multiobjective or multi-criteria optimization problem (MOOP). Examples of multiobjective optimization are seen as early as in nineteenth-century economics [26, 99].

A multiobjective optimization problem has a number of objective functions which are to be minimized or maximized. The problem usually has a number of constraints which any feasible solution (including the optimal solution) must satisfy. In the following formula, the multiobjective optimization problem is stated in its general form:

$$\left.\begin{array}{lll}\text{Minimize/Maximize} & f_m(x), & m = 1, 2, \ldots, M; \\ \text{subject to} & g_j(x) \geq 0, & j = 1, 2, \ldots, J; \\ & h_k(x) = 0, & k = 1, 2, \ldots, K; \\ & x_i^L \leq x_i \leq x_i^U, & i = 1, 2, \ldots, N. \end{array}\right\} \tag{2.1}$$

The last set of constraints is called variable bounds. These restrict each decision variable x_i to take a value within a lower bound x_i^L and an upper bound x_i^U. A solution x is a vector of N decision variables: $x =$

$(x_1, x_2, \ldots, x_N)^T$. The values of decision variables bounded by these limits constitute a *decision space* D. There are J inequality and K equality constraints that are associated with the problem above. The terms $h_k(x)$ and $g_j(x)$ are called constraint functions. Instead of "$\leq$" type inequality constraints, "$\geq$" type inequality constraints can be used. If a value of x satisfies all of the $(J + K)$ constraints and all of the $2N$ variable bounds, it is called a *feasible solution.* There are M objective functions $F(x) = (f_1(x), f_2(X), \ldots, f_M(x))^T$ in the above equation. Each objective function can be either minimized or maximized. Multiobjective optimization is sometimes referred to as *vector optimization*, because an M-tuple of objectives is optimized. The space in which the objective vectors belong is called the *objective space.*

Some of the well-known methods to solve multiobjective optimization problems are,

- *weighting objectives*,
- ϵ-constraint method,
- *goal programming*,
- *hierarchical optimization*,
- *global criterion*,
- *distance functions*,
- *min-max optimum*,
- and, *trade-off* methods [31, 28].

All of the classical methods listed above suggest a way to convert a multiobjective optimization problem into a single-objective optimization problem. For such a conversion, the methods above require more knowledge about the problem. For example, in weighting objectives method, the individual weights should be known and assumed to be constant.

Most multiobjective optimization methods use a concept called *domination.* In these methods, two solutions are compared on the basis of whether or not one solution dominates the other solution. In [22], domination is defined as the following:

Definition 2.1.1 (Strictly Better). *A solution x^2 is strictly better than solution x^1 for a given objective function f_m, if*

- $f_m(x^1) < f_m(x^2)$*, when the objective is to maximize* f_m*,*
- $f_m(x^1) > f_m(x^2)$*, when the objective is to minimize* f_m

Definition 2.1.2 (Better solution). *Between two solutions x^1 and x^2, $x^1 \triangleleft x^2$ denotes that the solution x^1 is better than the solution x^2 on a particular objective.*

Definition 2.1.3 (No worse than). *Between two solutions x^1 and x^2, $x^1 \ntriangleleft x^2$ denotes that the solution x^2 is no worse than the solution x^1 on a particular objective.*

Definition 2.1.4 (Domination). *A solution x^1 is said to dominate the other solution x^2 (or mathematically $x^1 \preceq x^2$), if both conditions 1 and 2 are true:*

1. *The solution x^1 is no worse than x^2 in all objectives, or $f_m(x^2) \ntriangleleft f_m(x^1)$ for all $m = 1, 2, \ldots, M$.*

2. The solution x^1 is better than x^2 in at least one objective, i.e., $\exists m' \in \{1, 2, \ldots, M\}$ such that $f_{m'}(x^1) \lhd f_{m'}(x^2)$).

Definition 2.1.5 (Non-dominated Set). *Among a set of solutions X, the non-dominated set of solutions X' are those that are not dominated by any member of the set X.*

When the set X is the entire search space, the resulting non-dominated set $X' \subset X$ is called the *Pareto optimal set*.

Definition 2.1.6 (Globally Pareto-optimal Set). *The non-dominated set of the entire decision space D is the globally Pareto-optimal set.*

In reaction to the weaknesses of the classical methods, a new set of methods like *genetic algorithms* were designed. For a more complete list of optimization algorithms, the reader can refer to [22].

However, as it is stated in [130, 47, 124], there are two major concerns or weaknesses in these methods. The first weakness is that finding an optimal solution may not be feasible with the given resources (e.g., computational power, time, cost, and knowledge limitations, etc.). The second weakness is that optimization fails to describe how decisions are often made in natural settings. Requirements for reliability, functionality, and robustness in uncertain and changing environments can conflict with optimal performance requirements. Therefore, new research areas have emerged for exploring concepts of decision making that do not depend on the principle of optimality.

2.1.2 Combinatorial Optimization

Combinatorial optimization problems are a subcategory of the general optimization problems where the values of some or all of the decision variables are restricted to be integer. The word “combinatorial” is used to present the fact that only a finite number of alternative feasible solutions exists. Combinatorial optimization problems are often referred to as *integer programming* problems. A survey of related applications and algorithms of combinatorial optimization can be found in [49, 98, 27].

Some classical combinatorial optimization problems are; 1) *knapsack problem* [83] which is important for cryptography, computer file protection, electronic transfer of funds, and electronic mail, 2) *network and graph problems* [4], 3) *travelling salesman problem* which has applications in routing, scheduling, large scale circuitry design, and strategic defense [6, 96], 4) *minimum cut problem* which has important implications for the reliability of large systems, and 5) *rule-based scheduling problems* [50].

Finding an optimal solution to combinatorial problems can be difficult due to the fact that unlike linear programming whose feasible region is a convex set, in combinatorial problems, a lattice of feasible points or a set of disjoint half-lines or line segments need to be searched to find an optimal solution [56].

Three of the well-known approaches for solving integer programming problems are; 1) enumerative techniques (e.g. branch and bound) [67], 2) relaxation [35] and decomposition techniques [51], and 3) cutting planes approaches based on polyhedral combinatorics [45].

2.1.3 Constraint Satisfaction Problems

A *constraint satisfaction problem* (CSP) consists of a finite set of variables, each associated with a domain of values, and a set of constraints. A solution is reached when there is an assignment of a value to each variable from its domain that satisfies all the constraints. Typical constraint satisfaction problems are; (1) to determine whether a solution exists, (2) to find one or all solutions, and (3) to find an optimal solution relative to a given cost function. In a way, MOOP defined in Equation 2.1 is a special instance of CSPs. The absence of a cost function makes the CSPs easier to solve than the classical MOOPs.

Two well known examples of constraint satisfaction problems are k-colorability and SATisfiability. In k-colorability, the task is to color a given graph with k colors so that any two adjacent nodes have different colors.

In SATisfiability, the task is to find the truth assignment to propositional variables in boolean expressions in *conjunctive normal form* (CNF), e.g. Equation 2.2, so that all clauses in boolean expressions are satisfied. While a *clause* is a boolean sum of variables or their negations, a boolean expression in CNF is a product of many clauses.

$$(A \vee B \vee C) \wedge (\bar{C} \vee D \vee A) \wedge (\bar{D} \vee E) \tag{2.2}$$

The classical CSP framework had been introduced formally at the beginning of the 70's [87]. ([89] gives a brief history of the studies on CSP). The general constraint satisfaction problem is of high-complexity (*NP*-complete or worse) and provides no efficient algorithm to solve it. Therefore, one of the main research topics has been finding fast preprocessing algorithms that can make the search for a solution efficient in important practical cases.

A widely used method for solving the CSP is *backtracking*. In this method, variables are instantiated sequentially. As soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial instantiation violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has alternatives available. However, the backtracking method suffers trashing ([42]); i.e., search in different parts of the space keeps failing for the same reasons. For example, suppose the variables are instantiated in the order $V_1, V_2, \ldots, V_i, \ldots, V_j, \ldots, V_N$. Suppose further that the binary constraint between V_i and V_j is such that for $V_i = a$, the binary constraint disallows any value of V_j. In the backtrack search tree, whenever V_i is instantiated to “a”, the search will fail while trying to instantiate V_j. This failure will be repeated for each possible combination that the variables $V_k (i < k < j)$ can take. The cause of this kind of trashing is referred as to as lack of *arc consistency.*

The following definitions from [73] can be helpful to understand arc-consistency:

Definition 2.1.7 (n-ary constraint). *An n-ary constraint is a constraint in which the number of variables relevant to the constraint is n or less.*

Definition 2.1.8 (n-ary CSP). *An n-ary CSP is a CSP, in which each constraint is an n-ary constraint.*

Definition 2.1.9 (Binary CSP). *A binary CSP is a CSP, in which each constraint is unary or binary.*

It is possible to convert an n-ary CSP to another equivalent binary CSP [109]. A binary CSP can be depicted by a constraint graph in which each node (V_i) represents a variable and each arc ($Arc(V_i, V_j)$) represents a constraint between variables represented by the end points of the arc. A unary constraint is represented by an arc originating and terminating at the same node.

Definition 2.1.10 (Arc consistency). *$Arc(V_i, V_j)$ is arc consistent if for every value x in the current domain of V_i there is some value y in the domain of V_j such that $V_i = x$ and $V_j = y$ is permitted by the binary constraint between V_i and V_j.*

The concept of arc-consistency is directional; i.e., if an arc (V_i, V_j) is consistent, then it does not automatically mean that (V_j, V_i) is also consistent. Trashing due to arc-inconsistency can be avoided if, before the search starts, each arc (V_i, V_j) of the constraint graph is made consistent.

Algorithms for eliminating arc-inconsistency are only a subset of preprocessing algorithms that are used to increase the efficiency of the search algorithms. Some of such preprocessing algorithms are described in [33, 80, 87, 38, 23, 39, 145].

Later, a new research field has emerged that focuses on finding classes of constraint satisfaction problems where the preprocessing algorithms could find a solution by themselves. These classes of problems are referred to as "islands of tractability". Some identified classes are tree-structured problems, problems generated by graph grammars [88], and problems with a certain relationship between their graph structure and their level of consistency [24].

Sometimes a constraint satisfaction problem can be over-constrained. When there is no solution that will satisfy all constraints, some of the constraints can be relaxed to make the problem solvable. In such a case, the algorithm needs to find that the problem is unsolvable first, and then to determine the constraints that can be relaxed. To determine the constraints that will be relaxed, the algorithm can assign a level of importance to each constraint. Hierarchical CLP (HCLP) [11] is such a CLP language and a CSP solving algorithm that satisfies the constraints by the order of their importance.

The classical CSP model assumes a well-defined and stable constraint satisfaction problem, and the main task is to find either one or all solutions. However, in [89], Montanari and Rossi state that the constraints are dynamic in many real-life problems, that there is a need for more interactive constraint satisfaction systems, and that they provide assistance in the modeling phase and support dynamic changes in the constraints.

Phase Transitions

Comparing the performance of different search algorithms is important. Some measures are: the number of consistency checks, the number of nodes visited, CPU time, the number of permanent search no-goods (the values that are eliminated from variable domains permanently to establish the arc consistency), and the number of temporary search no-goods (variable values that are discarded for a particular search step). Theoretical evaluation of constraint satisfaction algorithms is accomplished primarily by worst-case analysis or by dominance relationships [72].

All these measures are useful only if the algorithm terminates. For NP-complete problems, however, a different approach is needed. In the past, untractable problems were avoided. But, then it was recognized that for NP-complete problems, solutions can be found, except for some input regions called "phase transitions".

The idea of *phase transitions* was first introduced in statistical physics. From [60]: "Studies in statistical mechanics have shown that despite the apparent diversity in the composition and underlying structure of these systems, phase transitions take place with universal quantitative characteristics, independent of the detailed nature of the interactions between individual components. This means the singular behavior of observables near the transition point is identical for many systems when appropriately scaled, defining universality classes that only depend on the range of interaction of the forces at play and the dimensionality of the problem. One of these common characteristics is rapidly increasing correlation lengths between parts of the system as the transition is approached, giving rise to a change from a disordered to an ordered state and particularly large variances. It is these so-called *critical phase transitions* that are most relevant to computational search."

In [14], Cheeseman showed that, for many NP-complete problems, one or more "order parameters" can be defined, and that hard instances occur around particular critical values of these order parameters (or invariants). Such critical values form a boundary that separates the space of problems into two regions or *phases*. While one region is under-constrained and easy to find a solution, the other region is over-constrained and unlikely to contain a solution. Really hard problems occur on the boundary between these two regions where the probability of finding a solution is low but not negligible.

Currently, a normal approach is to evaluate a proposed algorithm empirically on a set of randomly generated instances taken from the relatively "hard" *phase transition* region [122]. As Cheeseman indicated, there is a need to produce phase transition diagrams for particular problem domains to help in identifying hard problems and predicting the existence of solution, such as shown in [103]. Phase transitions in constraint satisfaction problems have also been studied by [101, 128, 43, 121, 61, 58, 59, 147]. At this point, the challenge is to identify an order parameter for a particular problem domain.

2.1.4 Satisficing

In [125, 126, 127], Herbert Simon proposed that searching for an optimal solution could be terminated when an option was identified that met the the decision maker's *aspiration level*, the point where the cost of further searching for alternatives exceeded the expected benefit of continuing the search. Instead of using an optimal program, which maximizes a pay-off function, he suggested determining a threshold (aspiration level) that the payoff must exceed. The payoff requirement would be represented as an additional inequality that needed to be satisfied. The aspiration level is a term borrowed from psychology, where it represents a dynamic, context-dependent criterion typically acquired by experience. While Simon's *satisficing* idea inspired many other theories, the seemingly *ad hoc* nature of determining an aspiration level was criticized as arbitrary [131]. In [150], the weaknesses and open questions of Simon's definition of satisficing were summarized as follows:

- The aspiration level tells the designer nothing about the problem solving technique.
- What is a "good enough" solution?
- How can a computer measure that?
- Should a satisfactory solution be reached directly or by iterative refinement?
- How is the performance of a satisficing agent evaluated ?

Current approaches to satisficing are approximate reasoning, meta-reasoning, bounded optimality, and a combination of the above. These approaches lead to different agent-based designs and performances, although optimal meta-reasoning and bounded optimality are favored over other approaches. Papers are beginning to emerge regarding the problems of satisficing multiple-agent decision making [123]. However, they are mainly interested in the algorithms, and little has been done to formalize a multi-agent satisficing concept.

Approximate Reasoning

An approximate model of the problem domain can be used to find a solution. A solution which is optimal to the simplified problem, is not necessarily optimal for the original problem. Tractable models can be created by abstraction or by making simplifying assumptions. Some of the work in this area were *Bayesian belief networks*, *reasoning with approximate theories* [111] (or knowledge compilation as it is called in [120]), and *fuzzy logic*.

Meta-Reasoning

Perfect Rationality (or Type I Rationality) is the classical notion of rationality in economics and philosophy. A perfectly rational agent acts at every instant in a way that maximizes its expected utility, given the information it has acquired from the environment. Since action selection requires significant computation time, perfectly rational agents do not exist for non-trivial environments.

Meta-reasoning, also called Type II rationality by I. J. Good [46], utilizes some sort of *metalevel architecture*. Metalevel architecture is a design concept for intelligent agents that divides an agent into two (or more) levels. The first level, called *object level* carries out the computations in the problem domain. The second level, called *metalevel*, is a second decision making process whose application domain consists of the object level computations themselves and the computational objects and states that they affect. The basic idea is that object-level computations are actions with costs and benefits. A rational metalevel selects computations according to their expected utility.

Anytime algorithms are used for a general class of meta-reasoning problems. Anytime algorithms are the algorithms whose quality of results improve gradually as computation time increases. They offer a tradeoff between resource consumption and output quality [20, 62, 21]. There are two types of anytime algorithms, interruptible and contract. An interruptible algorithm can be interrupted at any time to produce results whose quality is described by its performance profile, where a performance profile is a probabilistic description of the dependency of output quality on computation time. In a contract algorithm the total time allocated for computation needs to be known in advance. Therefore, interruptible algorithms are more flexible than contract algorithms although they are more complicated to construct for three reasons. Such an algorithm gives a solution before the deadline independently of the quality of the best solution it knows at the time. In other words, the "soon-enough" requirement has higher priority than the "good-enough" requirement. First, the designer has to ensure the interruptibility of the composed system, that the system as a whole can respond to immediate demands for output [113]. Second, a mechanism has to allocate the available computation optimally among the components to maximize the throughput and the total output quality. While the problem can be solved in time linear in program size when the call graph of the components is tree-structured [114], the problem is NP-hard for the general case. Third, almost all metalevel reasoning systems to date have adopted a *myopic* strategy – a greedy, depth-first search at the metalevel. Research has started to develop programming tools for composition and monitoring of anytime algorithms [149, 48, 91].

Bounded Optimality

A bounded optimal agent behaves as well as possible given its computational resources [112]. The following definitions allow better understanding of the concept of bounded optimality. Let O be the set of percepts that the agent can observe at any instant, and A be the set of possible actions the agent can carry out in the external world (including the action of doing nothing). Then;

Definition 2.1.11 Agent function $f : O^* \rightarrow A$ *defines how an agent behaves under all circumstances.*

Assume $Agent(l, M)$ is the agent function implemented by the program l running on machine M, L_M is the finite set of all programs that can be run on M, $\mathbf{E}$ is the environment class in which the agent operates, and U is the performance measure which evaluates the sequence of states through which the agent drives the actual environment. Finally, $V(f, \mathbf{E}, U)$ denotes the expected value according to U obtained by any agent function f in environment class $\mathbf{E}$. Then;

Definition 2.1.12 (Bounded optimality) *The bounded optimal program* l_{opt} *is defined as*

$$l_{opt} = argmax_{l \in L_M} V(Agent(l, M), \boldsymbol{E}, U) \qquad (2.3)$$

In [115], the steps to construct a provably bounded optimal agent are specified as follows:

- Specify the properties of the environment in which actions will be taken.
- Specify a class of machines on which the programs are to be run.
- Specify a construction method.
- Prove that the construction method succeeds in building bounded optimal agents.

As Zilberstein points out, bounded optimality represents a well-defined optimization problem, but, it actually shifts the intractable computational task from the agent to its designer. While desirable, it is hard to achieve. The approach described in [106] is one of many that combine multilevel reasoning and bounded optimality.

Satisficing Equilibria

In terms of grammar, there are three degrees of comparison: (1)*Superlative* (or highest) degree is founded on the notion of being "best" and requires rank-ordering preferences for the consequences associated with the solutions [57]. (2) *Positive* (or lowest) degree is founded on the notion of being "good" and requires no explicit preference orderings or comparisons. (3) *Comparative* degree (or paradigm), is founded on the notion of being "better" and tries to fill the gap between the superlative and positive degrees. Under the comparative paradigm, a set of utilities are used to provide rankings of attributes for each solution.

In [131], Stirling and Goodrich have introduced the notion of *satisficing equilibria.* Given a set of solutions in a decision making problem, instead of making one global decision with respect to the entire collection of solutions, the *comparative paradigm* requires a separate local decision to be made for each solution.

Definition 2.1.13 (Satisficing equilibria) *A solution is in a state of satisficing equilibrium if*

S-1 *The benefits derived from adopting it at least compensate for the costs incurred.*

S-2 *No other solution provides more benefits without also costing more, or costs less without also providing less benefit.*

Condition S-1 provides a weak notion of adequacy. Condition S-2 applies the domination principle to the cost-benefit framework to eliminate options that needlessly sacrifice performance or incur expense. In general, the set of solutions in a state of satisficing equilibrium will not be a singleton. Further elimination will be required before action can be taken.

2.1.5 CSP Solution Methods and Tools

Constraint Logic Programming

Application developers took advantage of the constraint-solving methods and used constraint-related techniques successfully in applications such as assignment problems, CAD, decision-making systems, graphics, network management, robotics, scheduling, typesetting and VLSI. This led to the design and implementation of a number of constraint-based programming languages [17]. SKETCHPAD [134], CONSTRAINT [133], and ThingLab [9] were some of the earlier constraint programming languages.

Another group of programming language designers recognized that logic programming was an appropriate language for stating combinatorial search problems. Its relational form made it easy to express constraints while its non-determinism removed the need for programming a search procedure. However, traditional logic programming languages (e.g., Prolog) could be inefficient, causing trashing, repeated failure due to the same reason, or having to do redundant work during backtracking– all due to their passive use of constraints to test potential values instead of reducing the search space actively [143]. In addition, defining rich data structures and operations on these structures were not possible.

Earlier constraint logic programming (CLP) languages like CHIP [139], CLP($\Re$) [64], Prolog II, and Prolog III tried to preserve the advantages of logic programming without being affected by its limitations. Later, the CLP scheme [63] generalized the fundamental idea behind these constraint logic programming languages by defining a family of programming languages based on their semantic properties. The scheme could be instantiated to produce a specific constraint logic programming language by defining a constraint system. It was later generalized into the *concurrent constraint* `CC` framework of concurrent constraint programming to enable issues such as concurrency, control, and extensibility to be addressed at the language level.

Today's well known constraint programming languages, in addition to CHIP and Prolog III [15], are Eclipse [29], OZ [116, 129, 84], CIAO [55], AKL [53], Prolog IV [16], HAL [25] and Salsa [74]. Their constraint vocabulary and solvers perform beyond traditional linear and non-linear constraints and support logical and global constraints. OZ, CIAO, and AKL use CC framework and implement distributed and concurrent systems. However, these languages mostly target the computer scientists and have weaker abstractions for algebraic and set manipulation.

Helios language [140] and Numerica [141], on the other hand, have been designed to solve non-linear constraint systems using interval analysis techniques while CLP toolkits, like QOCA [82], EaCL [137] and Ultraviolet [10], implement graphical user interfaces to monitor the progress of the constraint solver and provide the user with a mechanism to interact with the solver at run-time.

Modeling Languages for CSP

Mathematical modeling languages are another kind of tool used in optimization. Modeling languages like AMPL [36], GAMS [3], Claire [77, 78], CML [5], and VISUAL SOLVER [142] provide high-level algebraic and set notations to express mathematical problems that can then be solved using the solvers just mentioned. In the case of CML, the models written by this language are later translated to CHIP.

There are also a new set of optimization programming languages, like OPL [8], XPRESS-MP, and Modeler++ [86], which aim to unify modeling and constraint programming languages.

In addition, there are negotiation [40], machine learning [110] and constraint query languages [68, 13] where CLP and database technologies were integrated as well as other techniques that have been applied to constraint problems to help the user to model the system.

2.1.6 Active Software

In [104], Laddaga states that there are three principle interrelated problems facing the software development:

- Escalating complexity of application functionality
- Insufficient robustness of the applications
- The need for autonomy

To deal with these problems, Laddaga proposes an approach called *active software*. A system that follows this approach must be responsible for its own robustness and manage its own complexity. To accomplish this, the system must incorporate the representations of its goals, methods, alternatives, and environment. The collection of available technologies under active software are *tolerant software*, *physically grounded software*, *self adaptive software*, and *negotiated coordination*. All of these technologies incorporate the knowledge of requirements, designs, structure, I/O sources in the running software and can be used together. Tolerant software is software that can tolerate non-critical variations from nominal specification. Physically grounded software is software that takes explicit account of its environment and other physical factors in the context of embedded systems. A discussion of *self adaptive software* and *negotiated coordination* follows in the following sections.

Self Adaptive Software

Self-adaptive software is defined in [75]: "Self-adaptive software evaluates its own behavior and changes behavior either when the evaluation indicates that it is not accomplishing what the software is intended to do or when better functionality or performance is possible." To accomplish constant performance evaluation and behavior change when the performance drops below criteria, the runtime code needs to include (1) descriptions of the software goals, design, and program structure and (2) a collection of alternative implementations and algorithms.

Early research in self-adaptive software has concentrated on three paradigms: *dynamic planning systems*, *self-controlling systems*, and *self-aware systems*. Dynamic planning systems plan their actions first [44, 92]. After evaluating and confirming the effectiveness of their actions, they start the execution. Planning includes scheduling and configuration of resources such as hardware, communication capacity, and software components.

In self-controlling systems [70, 71, 30, 41, 7] (detailed in Section 3), the runtime software behaves like a plant, with inputs and outputs monitored and controlled by separate monitoring and controlling units. There are three levels of control: *feedback*, *adaptation*, and *reconfiguration*.

In a self-aware system [136, 69, 12, 107], the key factor is a self-modeling approach. The application is built to contain knowledge of its operation; and it uses this knowledge to evaluate performance and to configure and adapt to changing circumstances.

In [76], Laddaga points to the following problems and unsolved issues in existing self-adaptive work:

- Evaluation of the functionality and the performance at runtime [85].
- Dynamism and software architecture representations for self-adaptive software: more introspective languages, better debug-ability, better process descriptions, better structural descriptions [100].
- Runtime performance while evaluating outcomes of computations

and determining if expectations are met.

- Effort required to create software capable of evaluating and reconfiguring itself.

- Lack of adequate metrics for degree of robustness and adaptation.

- Advances in computer hardware.

Two of the implementations of self-adaptive software are SAFER and ASC [79].

Negotiated Coordination

Negotiation is defined as "a process by which a joint decision is made by two or more parties. The parties first verbalise contradictory demands and then move towards agreement by a process of concession making or search for new alternatives [102]". Negotiated coordination is defined as "the coordination of independent software entities via mutual rational agreement on exchange conditions" [104].

There are advantages to using negotiation among agents in a software system because it is inherently distributed, multi-dimensional, and robust against the changes in the environment.

Negotiated coordination, another approach covered under the active software paradigm, was supported by a number of government and research organization programs. One of these programs is called the ANTs (Autonomous Negotiating Teams) program sponsored by AFRL and DARPA. The objective of ANTs program [2] is to provide technology that enables the development of information systems that autonomously negotiate the assignment and customization of resources to tasks in real-time, distributed allocation systems. Under the ANTs program, a series of research projects have started:

- CAMERA (Coordination and Management of Environments for Responsive Agents), a joint project executed by ISI/University of Southern California and Vanderbilt University. Its functionality is scheduling of pilots against tasks and planes, self-monitoring and reporting of the negotiating agents, and self-correcting negotiations that will force the agents to work collectively.

- ATTEND (Analytical Tools To Evaluate Negotiation Difficulty), maps the resource management problem that the system is trying to solve by negotiations into a CSP. ATTEND uses: ideas like satisficing decision making, control over real-time performance, and complexity reduction via phase-transition aware partitioning of task space; management of resource contention facilitated by SAT encoding of complex allocation problems.

- MARBLES [37], a definition and comparison of cooperative negotiation schemes for distributed resource allocation; assigns values (or prices) to the tasks and allocates the higher value resources first.

- DEALMAKER utilizes agents that select the best sources of supply to fill orders [135]; uses a flexible XML-based representation for the contracts and has an interactive user interface to enter contracts online and enter rules to govern the contracts.

- The MICANTS (Model Integrated Computing and Autonomous Negotiating Teams for Autonomous Logistics), executed by Vanderbilt

University; seeks to develop efficient negotiation protocols for distributed problem solving in the logistics domain.

- MAPLANT (Maintenance Planning Tool) [138], part of MICANTS project; program goal to create a schedule for airplane maintenance activity; scheduling problem first transformed to a finite domain constraint problem, then the CSP code manually written in OZ; input data provided in XML format. Like DEALMAKER, MAPLANT has a graphical user interface that helps the user interact with the search process.

- ADEPT (Advanced Decision Environment for Process Tasks) , a multi-agent system based on an approach suitable for implementing systems to manage business processes [65]; system and agent architectures designed to ensure maximum flexibility to adapt as the business process changes; provides a method for designing agent-oriented business process management system, agent implementation suitable for operation within such a system, and a technology for solving the problem of integrating an enterprise in the performance of a business process [66].

Other current research on negotiation can be found in [132, 40, 148, 19, 90, 34].

2.1.7 Negotiation Models

Bilateral Negotiation Model

This model discussed in this section is the *two parties, multiple issues* value scoring model defined in [105]. It is a model for bilateral negotiations about a set of quantitative variables. In a two-party negotiation sequence called a *negotiation thread*, offers and counter-offers are generated by linear combinations of simple functions, called *tactics.* Tactics generate an offer and counter-offer considering a single criterion such as time or resources. To achieve flexibility in negotiation, agents may wish to change their ratings of the importance of the different criteria, and their tactics may vary. Through *Strategy* an agent changes the weights of tactics over time. Strategies combine tactics depending on the negotiation history.

Let $i \in \{a, b\}$ represent the negotiating agents and $j \in \{1, \cdots, n\}$ the issues under negotiation. Let $x_j \in [min_j, max_j]$ be a value for issue j. Each agent has a scoring function $V_j^i : [min_j, max_j] \rightarrow [0, 1]$ that gives the score agent i assigns to a value of issue j in the range of acceptable values. For convenience, scores are kept in the interval [0,1]. w_j^i is the importance of issue j for agent i. The weights for all agents are normalized, i.e. $\sum_{1 \le j \le n} w_j^i = 1$, for all i in $\{a, b\}$. An agent's scoring function for a *contract* - that is for a value $x = (x_1, \cdots, x_n)$, is defined as:

$$V^i(x) = \sum_{1 \le j \le n} w_j^i V_j^i(x_j) \tag{2.4}$$

A Service-oriented Negotiation Model

In service-oriented negotiations, the agents undertake two possible roles that are in conflict, the *client* and the *server*. Roman letters $c, c_1, c_2, \cdots$ are used to represent client agents and $s, s_1, s_2, \cdots$ are used for server agents.

The negotiating agents may have conflicting interests. While a client agent wants a service as soon as possible with a low price, the server agent desires a higher price. Besides, the server agent's schedule may not allow an early service date. Therefore, in terms of negotiation values, the scoring functions of the client agent and the server agent show opposite tendencies; for issue j, if $x_j, y_j \in [min_j, max_j]$ and $x_j \leq y_j$, then $V_j^s(x_j) \leq V_j^s(y_j) \Longleftrightarrow V_j^c(y_j) \leq V_j^c(x_j)$.

Once the agents have determined the set of variables over which they will negotiate, the negotiation process between two agents consists of an alternate succession of offers and counter offers of values for those variables. This continues until an offer or a counter offer is accepted or an agent terminates negotiation.
In the following definitions, $x_{a\to b}^t$ represents the value of the offer proposed by agent a to agent b, and $x_{a\to b}^t[j]$ represents the value of issue j proposed from a to b at time t.

Definition 2.1.14 (Negotiation thread). *A negotiation thread between agents $a, b \in Agents$, at time $t \in Time$, noted $x_{a\to b}^t$ or $x_{b\to a}^t$, is any finite sequence of the form*
$\{x_{d_1\to e_1}^{t_1}, x_{d_2\to e_2}^{t_2}, \cdots, x_{d_n\to e_n}^{t_n}\}$ *where:*

1. *$e_i = d_{i+1}$, proposals alternate between both agents,*

2. *$t_k \leq t_l$ if $k \leq l$, ordered over time,*

3. $d_i, e_i \in \{a, b\}$, the thread contains only proposals between agents a and b,

4. $d_i \neq e_i$, the proposals are between agents, and

5. $x^{t_i}_{d_i \rightarrow e_i} \in [min^{d_i}_j, max^{d_i}_j]$ or is one of $\{accept, reject\}$.

Assume $x^{t'}_{b \rightarrow a}$ is the contract that agent a would offer to agent b at the time of the interpretation t', and t^a_{max} is a constant that represents the time which agent a must have completed the negotiation.

Definition 2.1.15 (Offer). *The interpretation by agent a of an offer $x^t_{b \rightarrow a}$ sent at time $t < t'$, can be formalized as follows:*

$$I^a(t', x^t_{b \rightarrow a}) = \begin{cases} reject, & \text{If } t' \geq t^a_{max} \\ accept, & \text{If } V^a(x^t_{b \rightarrow a}) \geq V^a(x^{t'}_{a \rightarrow b}) \\ x^{t'}_{b \rightarrow a}, & otherwise \end{cases} \tag{2.5}$$

In order to prepare a counter offer the following families of tactics are defined [32]:

Time-dependent: If an agent has a time deadline by which an agreement must be reached, these tactics model the fact that the agent is likely to concede more rapidly as the deadline approaches.

Resource-dependent: These tactics model the pressure in reaching an agreement that limited resources (e.g. money, labor, raw material, or any other) and the environment (e.g. number of clients, number of servers, or economic parameters) impose upon the agent's behavior.

Imitative: In situations where the agent is not under great pressure to reach an agreement, the choice may be to use imitative tactics that protect the agent from being exploited by other agents. In this case the counter offer depends on the behavior of the negotiation opponent.

Mediator-based Negotiation

While observing the benefits of a decentralized agent-based systems, applications show that agents which try to maximize their goals all the time may not reach an agreement if their goals are conflicting. In these situations a mediator can be utilized for conflict resolution. Examples of mediator-based negotiation approach are seen in [18, 119].

2.2 Analysis of Candidate Components

The kind of problems addressed in this thesis are multiobjective combinatorial optimization problems. In the following, an analysis of the candidate solutions described in Section 2 is provided. The goal is to come up with a solution that satisfies all the requirements of the problem stated in Section 1.

The first candidate for a component of the solution is the conversion of a multiobjective optimization problem into a single-objective optimization problem by combining all the objective functions into one. This is described in Section 2.1.1. An example of the combination of the functions into one consists of assigning weights to each objective and then adding the weighted values of the objective functions. The combination method is based on some additional information about the problem. Other methods, similar to this one, also use some additional information about the problem. The problem formulation described in Section 1, however, does not include this kind of information. Since it is not assumed that this kind of information is available, this approach cannot be used in the solution to the problem.

The next candidate is the satisficing approach. However, as it is stated in Section 2.1.4, this approach is based on the assumption that an "aspiration level" for each objective is known. Again, this kind of an assumption is not part of the problem formulation. Therefore, a straight forward application of the satisficing approach cannot be used. Instead, a mediator-based negotiation mechanism is proposed to establish aspiration levels dynamically. To achieve this, an agent-based approach is proposed. In this approach, as in the MAPLANT project [138], each objective and the related constraints are transformed to a separate constraint satisfaction problem. Each objective is represented by an agent after which the agents negotiate solutions with the help of a mediator. Each agent uses a separate CSP solver to find a solution that will satisfice its objective (cf. [65]).

The ATTEND project [1] also utilizes the satisficing approach. The system monitors the performance of the negotiation to avoid phase transition regions. However, the problem domain for the ATTEND project is limited to SAT problems. Consequently, the ATTEND system monitors the invariants specific to the SAT problem, i.e., the ratio of the number of clauses to the number of variables. Like the ATTEND project, specialized components to monitor the performance of the CSP solvers and to detect phase transition regions are used. However, the problem domain in this thesis is not limited to SAT, and the same kind of invariants can not be used. Instead, one research goal is to find invariants related to the experimental scenarios selected to demonstrate the merits of the proposed approach. In addition, the proposed solution in this thesis differs from the ATTEND approach such that it terminates the search gracefully upon the detection of a phase transition behavior in the problem and returns the best of the available solutions up to that point. In this regard, the proposed solution acts as an anytime algorithm. At any time during the search, one can stop the proposed system and get a solution. Similar to anytime algorithms, the quality of the solution returned by the proposed system improves as it is allowed more time for the search.

In this thesis, the architecture used for the software agents is a metalevel architecture (see Section 2.1.4). This architecture, called *Self-Controlling Software Architecture* (SCSA), is a variation of the self-controlling software model [70]. In its full implementation it includes three metalevels (also called *loops*): the feedback loop, the adaptation loop, and the reconfiguration loop. In this thesis only the feedback loop is implemented in each agent.

One of the goals stated in Section 1 is to choose a language for specifying multiobjective combinatorial optimization problems. One possibility would be to use CML [5]. For this language, translation rules exist that could be used for translating a model expressed in CML to the constraint logic programming language CHIP [139] manually. Because the expressiveness of this language is limited, it is not possible to express structural aspects of the user's view of the problem formulation. Since this feature is needed for the problem formulated in this thesis, a more expressive language is required. Using UML/OCL instead is proposed for this purpose. The solution is further extended by the idea of translation and provides an automated way of translating the problem represented in a high-level modeling language to the CSP programming language. Automated translation hides the implementation details related to the CSP programming language and allows a change in the target CSP solver with minimum effort.

Other modeling language candidates could include Claire [77, 78] and VISUAL SOLVER [142]. While these languages have more expressive power than CML, as well a graphical tool support, the problem is that these tools are closed proprietary systems. On the other hand, UML, which is a de-facto standard language for specifying software specifications, provides a standardized XMI output, which then can be used as input to a CSP code generator. Using such a standard modeling language allows the utilization of off-the-shelf products like Rational Rose, Rhapsody or any graphical UML tool.

Chapter 3

Proposed Solution

3.1 Introduction

To achieve the goals outlined in the previous sections, an experimental system (see Figure 3.1) was implemented that consists of the solution elements described in Section 2.2. The system automates the synthesis of a satisficing program from specifications so that the program can control its complexity, can detect the presence of a phase transition behavior, and terminate the search gracefully.

The following tools were implemented as part of the experimental system:

- An ontology for multiobjective optimization. The ontology is specified in UML/OCL and supports the user in the specification of optimization problems (detailed in Sections 3.2 and 3.3).

- A parser that converts constraints expressed in OCL to an intermediate XML form.

- A generic code generator that converts the constraints expressed in the intermediate XML form and the structural constraints in XMI (from the UML tool) to a target CSP programming language (see Appendix 3.5 for details).
- Translation rules for the code generator for the OZ target CSP programming language.
- Templates for creating software agents.
- Specification for an agent-based Self-Controlling Software Architecture (SCSA). Section 3.6 describes SCSA. An OZ implementation for SCSA is provided.
- A mediator-based negotiation algorithm is developed. Section 3.8 details the algorithm.
- Phase transition invariants for the experimental multiobjective optimization problems are defined. Experiments are performed to assess the appropriateness of the invariants for defining phase transition regions. The invariants are used to generate problem instances showing phase transition behavior to test the experimental system. See Sections 4.2.4 and 4.3.3 for the discovery for phase transition invariants.

3.2 Defining Structural Constraints in UML

Domain specific knowledge about the problem area is expressed in this step. Class diagrams, as defined in the Unified Modeling Language (UML)

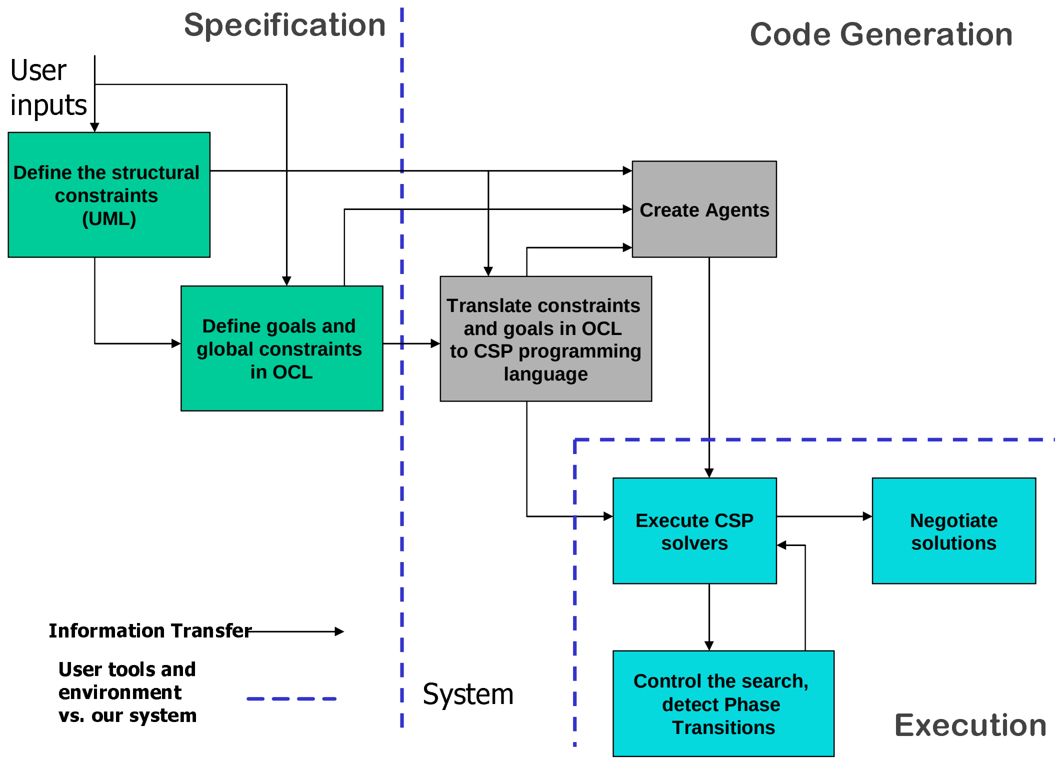

Specification
Code Generation
User inputs
Define the structural constraints (UML)
Create Agents
Define goals and global constraints in OCL
Translate constraints and goals in OCL to CSP programming language
Execute CSP solvers
Negotiate solutions
Control the search, detect Phase Transitions
Information Transfer
User tools and environment vs. our system
System
Execution

Figure 3.1: Overview of the proposed solution

Specification [95], are used to define structural constraints of the problem. Today, there are many commercial graphical UML tools available to create these diagrams. Rational Rose, I-Logix Rhapsody, Project Technology's BridgePoint and Kennedy-Carter's iUML are some of the well-known and widely used tools. These tools can export the information captured in the class diagrams in XML Metadata Interchange (XMI) form [94]. XMI specification supports the interchange of any kind of metadata that can be expressed using the Meta Object Facility (MOF) specification, including both model and metamodel information. MOF is the OMG's adopted technology for defining metadata and representing it as CORBA (Common Object Request Broker Architecture) [93] objects. In these specifications, *metadata* is a general term for data that describes information. The MOF supports any kind of metadata that can be described using object modeling techniques. Figure 3.2 is an example class diagram.

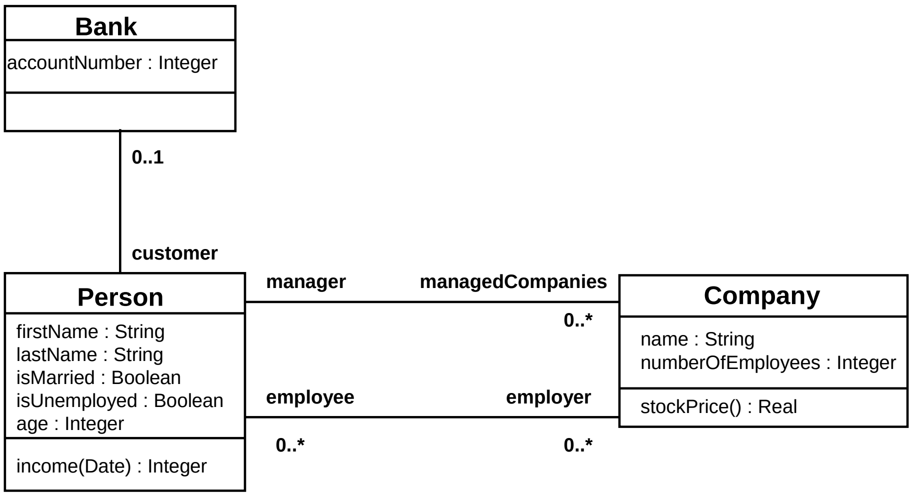


Figure 3.2: UML Class diagram for bank example

The following is a part of XMI file that is generated for the class diagram in Figure 3.2.

```
<?xml version="1.0" encoding="UTF-8"?> <XMI xmi.version="1.0">
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>Novosoft UML Library</XMI.exporter>
      <XMI.exporterVersion>0.4.19</XMI.exporterVersion>
    </XMI.documentation>
    <XMI.metamodel xmi.name="UML" xmi.version="1.3"/>
  </XMI.header>
  <XMI.content>
    <Model_Management.Model xmi.id="xmi.1" xmi.uuid="-8000">
```

```
<Foundation.Core.ModelElement.name>BankExample
                  </Foundation.Core.ModelElement.name>
<Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
<Foundation.Core.GeneralizableElement.isRoot xmi.value="false"/>
<Foundation.Core.GeneralizableElement.isLeaf xmi.value="false"/>
<Foundation.Core.GeneralizableElement.isAbstract
      xmi.value="false"/>
<Foundation.Core.Namespace.ownedElement>
  <Foundation.Core.Class xmi.id="xmi.2" xmi.uuid="-7ffe">
    <Foundation.Core.ModelElement.name>Bank
                  </Foundation.Core.ModelElement.name>
    <Foundation.Core.ModelElement.isSpecification
             xmi.value="false"/>
    <Foundation.Core.GeneralizableElement.isRoot
             xmi.value="false"/>
    <Foundation.Core.GeneralizableElement.isLeaf
              xmi.value="false"/>
    <Foundation.Core.GeneralizableElement.isAbstract
              xmi.value="false"/>
    <Foundation.Core.Class.isActive xmi.value="false"/>
    <Foundation.Core.ModelElement.namespace>
      <Foundation.Core.Namespace xmi.idref="xmi.1"/>
    </Foundation.Core.ModelElement.namespace>
    <Foundation.Core.Classifier.feature>
      <Foundation.Core.Attribute xmi.id="xmi.3"
           xmi.uuid="-7ffc">
        <Foundation.Core.ModelElement.name>accountNumber
```

```
                        </Foundation.Core.ModelElement.name>
              <Foundation.Core.ModelElement.visibility
                  xmi.value="public"/>
              <Foundation.Core.Feature.ownerScope
                  xmi.value="instance"/>

            ...

            </Foundation.Core.Attribute>
          </Foundation.Core.Classifier.feature>
        </Foundation.Core.Class>

       ...

      </Foundation.Core.Namespace.ownedElement>
    </Model_Management.Model>
  </XMI.content>
</XMI>
```

Models saved in machine readable XMI form will be input to the OCL parser (see Section 3.5). In the current form of XMI, there is no standard way to store the actions (an action language) that happen in different states of dynamic objects. Because defining an action language is outside the scope of this thesis, if there is a need for expressing state actions in any experiments, it will be programmed in the target CSP programming language.

3.3 Defining Goals and Constraints in OCL

The Object Constraint Language (OCL) Specification is described as follows. “In object-oriented modeling, a graphical model, like a class model, is not enough for a precise and unambiguous specification. There is a need to describe additional constraints about the objects in the model.” In this thesis, version 2.0 of the OCL Specification which is a part of UML version 1.5 has been used. OCL is used:

- To specify invariants on classes and types in the class model.
- To describe pre and post conditions on operations and methods.
- To specify constraints on operations.
- To specify the objective functions for the optimization problem.

The standard OCL comes with pre-defined types (basic or complex) and operations on these types. Collection types, like *Set*, *Bag*, and *Sequence*, and set operations, like *forAll*, *exists*, *union*, and *intersection*, are part of the standard OCL.

In this thesis, an ontology has been defined to extend the pre-defined types and operations in OCL to cover concepts such as

- New data types
- Minimize, maximize functions
- List operations (lists of integers)
- Search operations

- Templates for constraints and objective functions

Section 3.4 provides a complete list of the types and the operations that are introduced by the ontology. A template OCL file for an easy start to specify an optimization problem can be found in Appendix A.3.

3.4 Ontology

3.4.1 New Types

```
FD_VAR -- A finite domain integer variable that can be used in a
constraint satisfaction problem.

ONT_PROC -- A reference to a procedure, like a pointer to a
procedure in C programming language.

ONT_RECORD -- A structured data type

ONT_FIELD -- A labeled sub-part of an ONT_RECORD
```

3.4.2 New Operations

```
-- Creates a new finite domain integer decision variable
-- and initializes to the input..

context ONTOLOGY::ONT_NewInt(I : Integer) : FD_VAR
    post:
        let R: FD_VAR = I
    in
```

```
        result = R

-- Creates a new finite domain integer decision variable list
-- and sets the domain of all the list members to between 0
-- and the input. The input needs to be a positive integer.

context ONTOLOGY::ONT_NewList(Size:Integer, Max : Integer) :
                    Sequence(FD_VAR)
    pre: I > 0
    post:
        let L: Sequence(FD_VAR)
    in
        result = R

-- Creates a new array of integers and initializes the elements to
-- 0. The input needs to be a positive integer.

context ONTOLOGY::ONT_NewArray(Size:Integer) : Sequence(Integer)
    def:
        let L: Sequence(Integer)
    pre: Size > 0 and L->size = Size and L->iterate(I | I = 0)
    post:
        result = L

context ONTOLOGY::ONT_PutArray(Array: Sequence(Integer),
        Index:Integer, Value:Integer)
    pre: Index > 0 and Index <= Array->size
    post: Array->at(Index) = Value

 -- Every member of the list is mutually distinct.

context ONTOLOGY::ONT_Distinct(L : Sequence(Integer))
    post: L->forAll(p1,p2 |  p1 <> p2)

-- Returns the size of the input list.

context ONTOLOGY::ONT_Size(L : Sequence(Integer)) : Integer
    post: result = L->size

-- Increments the value of the input integer decision variable.

context ONTOLOGY::ONT_Inc(A: FD_VAR, B: Integer)
    post: A = @A + B

-- Introduces an inequality constraint for the input integer
-- decision variables.

context ONTOLOGY::ONT_NotEqual(A: FD_VAR, B: FD_VAR)
    post: A <> B

-- Introduces a "less than or equal" constraint for the input
-- integer decision variables.

context ONTOLOGY::ONT_LessThanEqual(A: FD_VAR, B: FD_VAR)
```

```
    post: A <= B

-- Introduces a "greater than" constraint for the input integer
-- decision variables.

context ONTOLOGY::ONT_GreaterThan(A: FD_VAR, B: FD_VAR)
    post: A > B

-- Sets the distribution strategy that will be used in the search
-- algorithm for the constraint satisfaction problem. The input
-- is a list of decision variables that will be determined as
-- part of the constraint satisfaction problem.

context ONTOLOGY::ONT_Dist(L: Sequence(FD_VAR))

-- Starts the search algorithm for the constraint satisfaction
-- problem specified by the procedure ConstraintProcedure. An
-- optional second procedure OrderProcedure is used if it is an
-- optimization problem.

context ONTOLOGY::ONT_Solve(ConstraintProcedure: ONT_PROC,
                            OrderProcedure: ONT_PROC)

-- Returns a record with same label and arity as R1, whose fields
-- are computed by applying the binary procedure P to all fields of
-- R1.
```

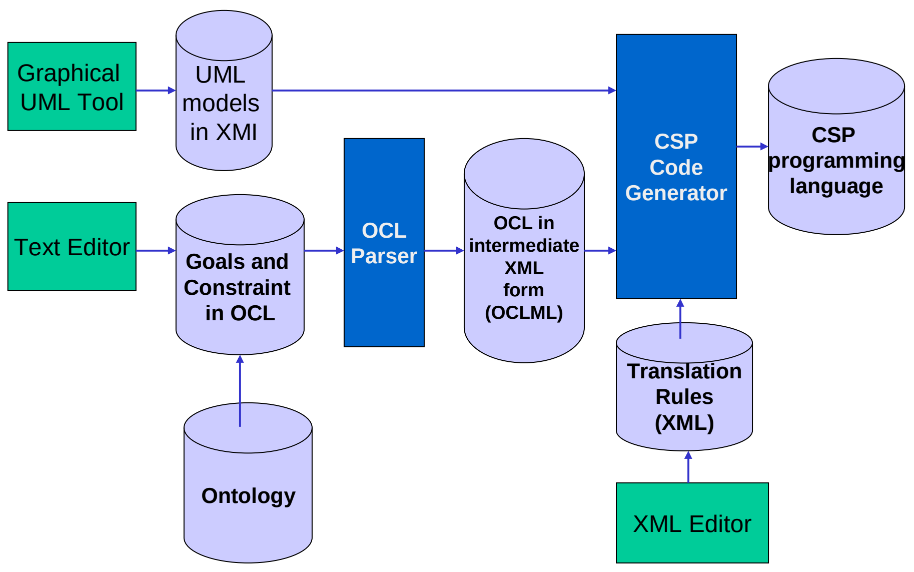


Figure 3.3: Translation of UML/OCL to the CSP programming language

```
context ONTOLOGY::ONT_Map(R1 : RECORD, P : ONT_PROC) : ONT_RECORD
    post:
    R1->iterate(F | P(F))
```

3.5 Translation of Goals and Constraints from UML / OCL to CSP Programming Language

Translation of goals and constraints to CSP programming language is done in two steps (see Figure 3.3). The OCL translator is made up of two separate components: the OCL parser and the generic CSP code generator. First, the OCL parser parses the OCL file and converts the domain-specific constraints into an intermediate XML form, which is called OCL Markup

Language (OCLML) in this thesis.

The following is an example for a constraint written in OCL:

```
package Bank
   context c:Company
      inv: c.numberOfEmployees > 50
endpackage
```

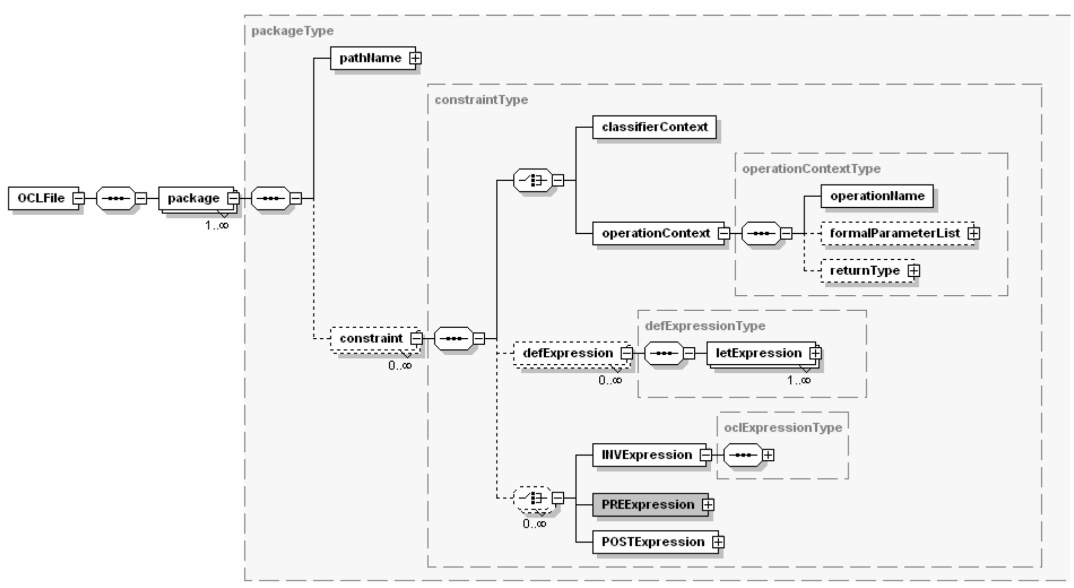


Figure 3.4: Schema for OCLML

And, the following is the same constraint after parsed and transformed to OCLML:

```
<OCLFile>
    <package Value="Bank">
        <constraint>
            <classifierContext Value="c">
                <secondaryName Value="Company"/>
            </classifierContext>
            <INVExpression Value="">
                <binaryOperator Operation="GREATER">
                    <leftOperand>
                        <unaryOperator Operation="">
                            <postfixExpression>
                                <primaryExpression>
                                    <propertyCall Value="c"/>
                                </primaryExpression>
                                <propertyCallList>
                                    <propertyCall
                                        Value="numberOfEmployees"
                                         Type="DOT"/>
                                </propertyCallList>
                            </postfixExpression>
                        </unaryOperator>
                    </leftOperand>
                    <rightOperand>
                        <unaryOperator Operation="">
                            <postfixExpression>
```

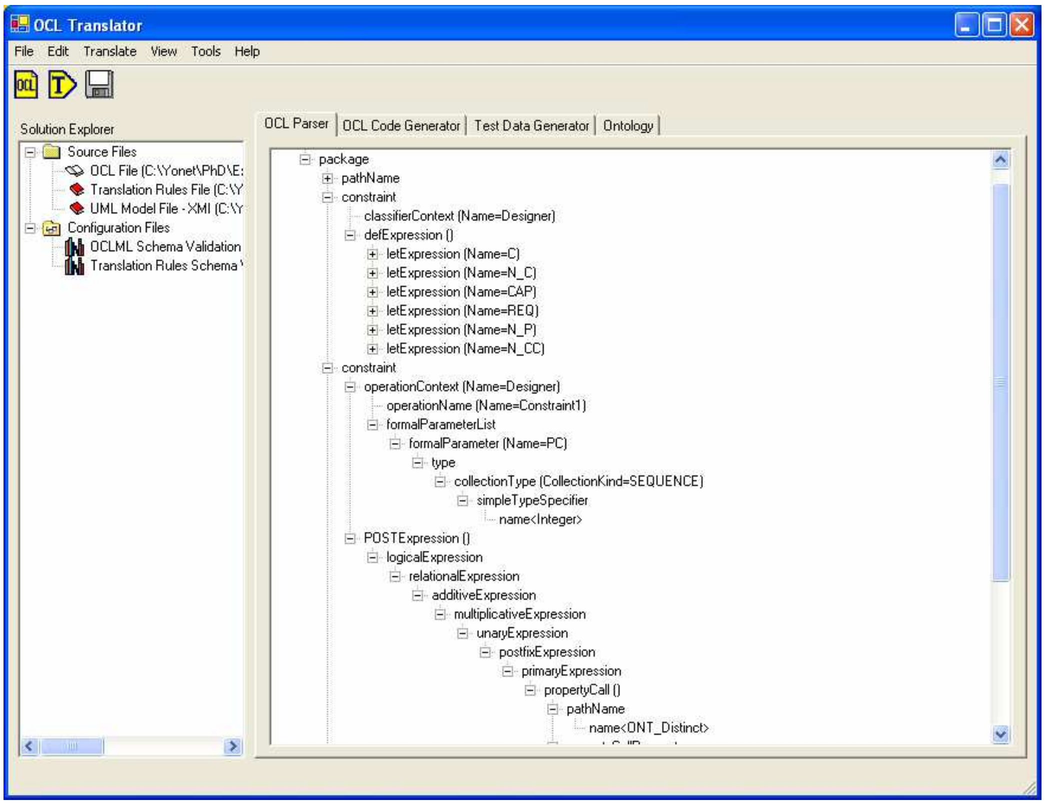
```
                                        <primaryExpression>
                                            <oclObject Value="50"/>
                                        </primaryExpression>
                                    </postfixExpression>
                                </unaryOperator>
                            </rightOperand>
                        </binaryOperator>
                    </INVExpression>
                </constraint>
            </package>
        </OCLFile>
```

A schema is provided to validate the intermediate files in OCLML (See Figure 3.4).

Lex (version 3.3.3.136) and Yacc (version 3.3.3.138) utilities from Mortice Kern Systems, Inc. were used to generate the OCL parser. Appendix A.6.1 includes a complete listing of the Lex (lexical analyzer) file and the reserved keywords for OCL. Appendix A.6.3 includes a complete listing of the Yacc (production rules) file.

3.5.1 CSP Code Generation

After the constraints and the objective functions that are expressed in OCL are parsed, the generic CSP code generator translates the constraints in OCLML form to the target CSP programming language using a translation rules file for the target language (see Figure 3.6). The separation of the parser from the code generator and the use of translation rules file make the OCL Translator flexible enough to generate CSP code for different target



```
package
  pathName
  constraint
    classifierContext (Name=Designer)
    defExpression ()
      letExpression (Name=C)
      letExpression (Name=N_C)
      letExpression (Name=CAP)
      letExpression (Name=REQ)
      letExpression (Name=N_P)
      letExpression (Name=N_CC)
  constraint
    operationContext (Name=Designer)
      operationName (Name=Constraint1)
      formalParameterList
        formalParameter (Name=PC)
          type
            collectionType (CollectionKind=SEQUENCE)
              simpleTypeSpecifier
                name<Integer>
    POSTExpression ()
      logicalExpression
        relationalExpression
          additiveExpression
            multiplicativeExpression
              unaryExpression
                postfixExpression
                  primaryExpression
                    propertyCall ()
                      pathName
                        name<ONT_Distinct>
```

Figure 3.5: OCLML tree generated by the OCL Translator

CSP programming languages.

In this thesis, the translation rules file for OZ language is provided for demonstration. OZ has been selected as the target CSP programming language, because;

- It supports concurrency.
- It has an open architecture and the source code is publicly available.
- It supports integration with other programs written in different programming languages, e.g. C, C++.

- It comes with a run-time debugging environment called Mozart.
- It provides a graphical user interface that helps to monitor the progress of execution.

Table 3.1 shows the high-level function `GenerateCSPCode`, that generates the CSP code, given an XMI file, an OCLML file, and a translation rules file. Table 3.2 shows the main function `WalkOCLMLTree`, that walks the OCLML tree and generates code for each UML class in the optimization problem. Before walking the OCLML tree, the code generator extracts the member variables of the UML classes and the associations among the UML classes by walking through the XMI tree. Function `WalkUMLAssociation` updates the UML classes in the OCLML tree with the member variables and the associations, such as inheritance and containment relationships, obtained from the XMI tree.

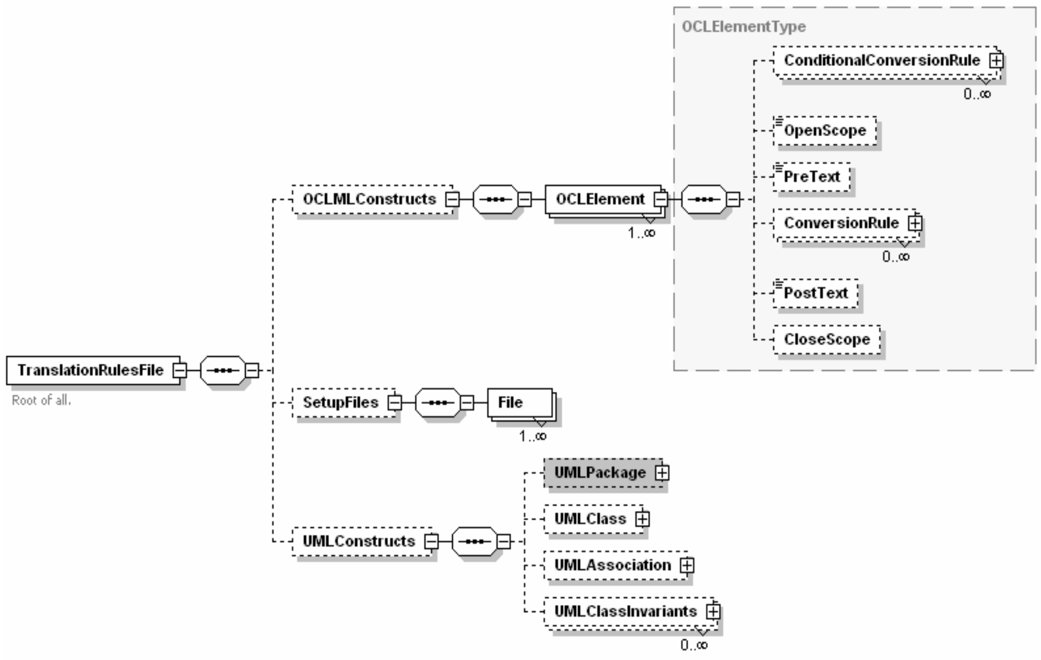


Figure 3.6: Schema for the translation rules file

Table 3.1: Listing for function GenerateCSPCode

```
proc GenerateCSPCode(fileNameXMI, fileNameOCLML,
    fileNameTransRules)
    XMITree = LoadXMIFile(fileNameXMI)
    OCLMLTree = LoadOCLMLFile(fileNameOCLML)
    Rules = LoadTranslationRulesFile(fileNameTransRules)

    WalkOCLMLTree(OCLMLTree, Rules, XMITree))
end
```

Table 3.2: Listing for function WalkOCLMLTree

```
proc WalkOCLMLTree(OCLMLTree, TranslationRules, XMITree)
    foreach association in XMITree do
        WalkUMLAssociation(association, OCLMLTree, TranslationRules)
    endfor

    foreach class in OCLMLTree do
        WalkUMLClass(class, TranslationRules)
    endfor
end
```

Table 3.3 lists function `WalkUMLClass` which creates a source file and walks through the sub-elements of the UML class. For each sub-element, it calls function `WalkOCLElement` which performs the actual code generation. Table 3.4 lists function `WalkOCLElement`. First, `WalkOCLElement` finds the translation rule for the given `OCLElement`. Next, it calls function `WalkConditionalConversionRule` to check whether or not the OCLElement requires special handling. For example, the translation rules file provided for the OZ language handles ontology members defined in Section 3.4, such as `ONT-Solve`, differently than any regular procedure call defined in OCL. If there is special code to be generated for the `OCLElement`, then `WalkConditionalConversionRule` handles it and returns `true`. If not, `WalkOCLElement` generates the code that will come before the element.

Table 3.3: Listing for function WalkUMLClass

```
proc WalkUMLClass(UMLClass, TranslationRules)
   OutputFile = OpenFile(UMLClass.name)

   foreach element in UMLClass.children do
      WalkOCLElement(element, TranslationRules, OutputFile)
   endfor

   CloseFile(OutputFile)
end
```

Table 3.4: Listing for function WalkOCLElement

```
proc WalkOCLElement(OCLElement, TranslationRules, OutputFile)
   Rule = FindTranslationRuleInfo(OCLElement.name)

   if WalkConditionalConversionRule( Rule.ConditionalConversionRules,
                              OCLElement, OutputFile) == true then
      return
   endif

   WalkConversionRule(Rule.OpenScope, OCLElement, OutputFile)
   foreach element in OCLElement.children do
      WalkOCLElement(element, TranslationRules, OutputFile)
   endfor
   WalkConversionRule(Rule.CloseScope, OCLElement, OutputFile)
end
```

Then, it calls itself recursively for each sub-element of OCLElement. Finally, `WalkOCLElement` generates the code that will come after the element.

3.6 Architecture for Software Agents

3.6.1 Background

In the proposed solution, a separate *software agent* represents each objective in the multiobjective optimization problem. "Agent" is a concept coined in artificial intelligence. The agent concept is used as Maes has defined in [81]: "An agent is a computational system which is long-lived, has goals, sensors and effectors, decides autonomously which actions to take in the current situation to maximize the progress towards its changing goals." The main goals in using software agents are to de-centralize the computation and to improve the performance in search by utilizing the adaptive and autonomous nature of agents.

In this thesis, a variation of the self-controlling software model [70] is used as the architecture for software agents. (See Figure 3.7 for self-controlling software model.) The self-controlling software model regards the software system as a plant to be controlled and models the behavior of the plant and the environment as a *dynamic system*.

A dynamic system is a physical system whose state changes or evolves from one moment of time to the next. The essence of a dynamic system is that its output depends on the system's state. Self-controlling software identifies measurable inputs to the plant and classifies them as *control inputs* which control the plant's behavior. In addition, self-controlling software identifies *disturbances* which alter the plant's behavior unpredictably; includes a *controller* subsystem for changing the values of the control inputs to the plant; and adds, if necessary, a *quality of service* (QoS) subsystem for computing feedback. The controller uses this feedback to control the plant. A self-controlling software includes three loops, each of which represents a

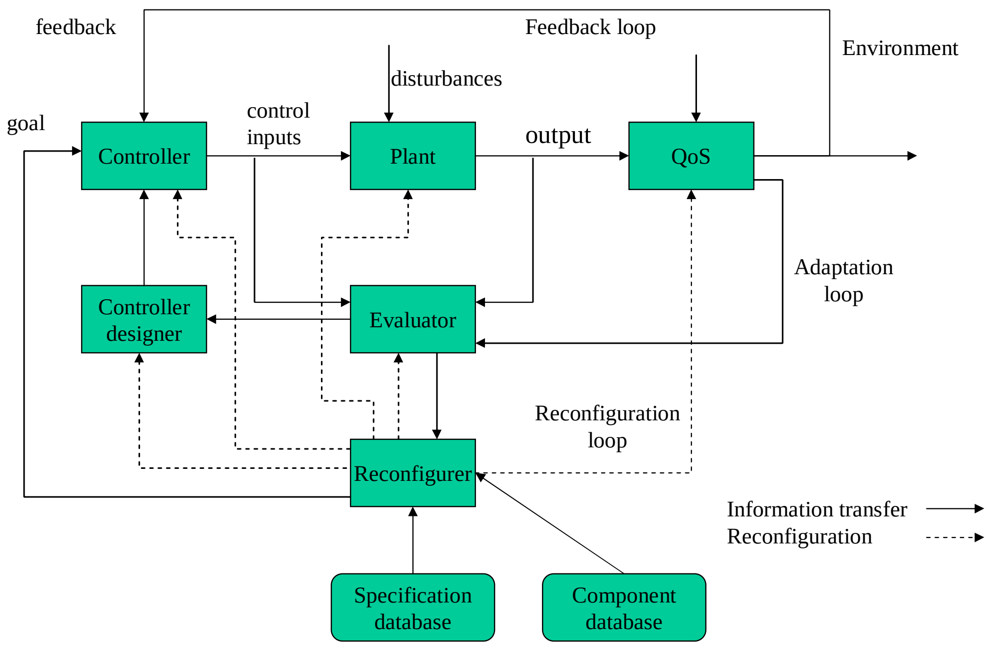


Figure 3.7: Self-controlling software model

different timescale for control activity.

In a *feedback loop*, the controller computes the control inputs for the plant based on the goal received from the reconfigurer and feedback received from the QoS subsystem. In an *adaptation loop*, an *evaluator* subsystem evaluates the behavior and performance of the plant to determine whether the plant's model is appropriate. If not, the evaluator uses a *controller designer* subsystem to modify the behavior of the controller. In a *reconfiguration loop*, a *reconfigurer* subsystem initiates structural changes in the QoS subsystem, evaluator, controller, controller designer, or even the plant. The reconfigurer uses a *specification database* for decision making and a *component database* to assemble various system elements.

3.6.2 Components of the Architecture

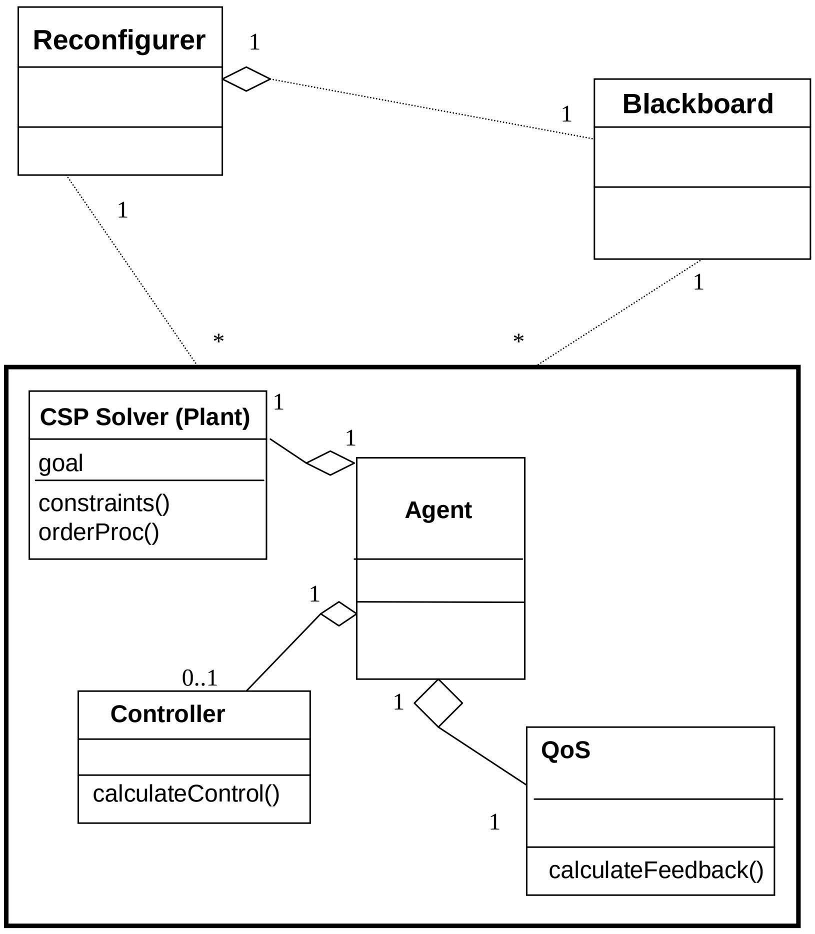


Figure 3.8: Agent-based Self Controlling Software Architecture (SCSA)

The agent-based Self-Controlling Software Architecture (SCSA) proposed in this thesis is a simplified version of the self-controlling software model described above. Figure 3.8 shows a class diagram for SCSA. SCSA contains a reconfigurer, a blackboard, and as many agents as the number of objective functions. The reconfigurer instantiates the agents and supports the communication among the agents. Section 3.6.3 describes this communication in detail. During the *negotiation* among the agents for a solution for the optimization problem, the reconfigurer acts as a mediator. The reconfigurer uses a *blackboard* to record and process the solutions obtained from agents. Blackboard is an architecture that was developed in the artificial intelligence community [54].

Each agent in SCSA is responsible for one objective function of the multiobjective optimization problem. Every agent utilizes a feedback loop described in self-controlling software model and contains a plant, a quality of service (QoS) subsystem, and a controller subsystem. The plant is a constraint satisfaction program that is auto-generated by the CSP code generator implemented in this research. QoS module monitors the performance of the constraint satisfaction program within the plant and returns a feedback to the controller subsystem. The controller subsystem uses this feedback to calculate the control inputs. These control inputs are used to adjust the behavior of the CSP solver that executes the constraint satisfaction program. Section 3.7 provides a formalization for the control in the constraint satisfaction program. The QoS subsystem also warns the plant if it detects a phase transition behavior in the current optimization problem. Appendix A.1 and A.2 lists the OZ implementation for agent and reconfigurer components of the system that were used in this thesis.

3.6.3 Communication among Components

The state transition diagram for the reconfigurer is shown in Figure 3.9.

The sequence diagram in Figure 3.10 demonstrates the sequence of events taking place while the system executes for a multiobjective optimization problem with two objectives. First, the reconfigurer instantiates

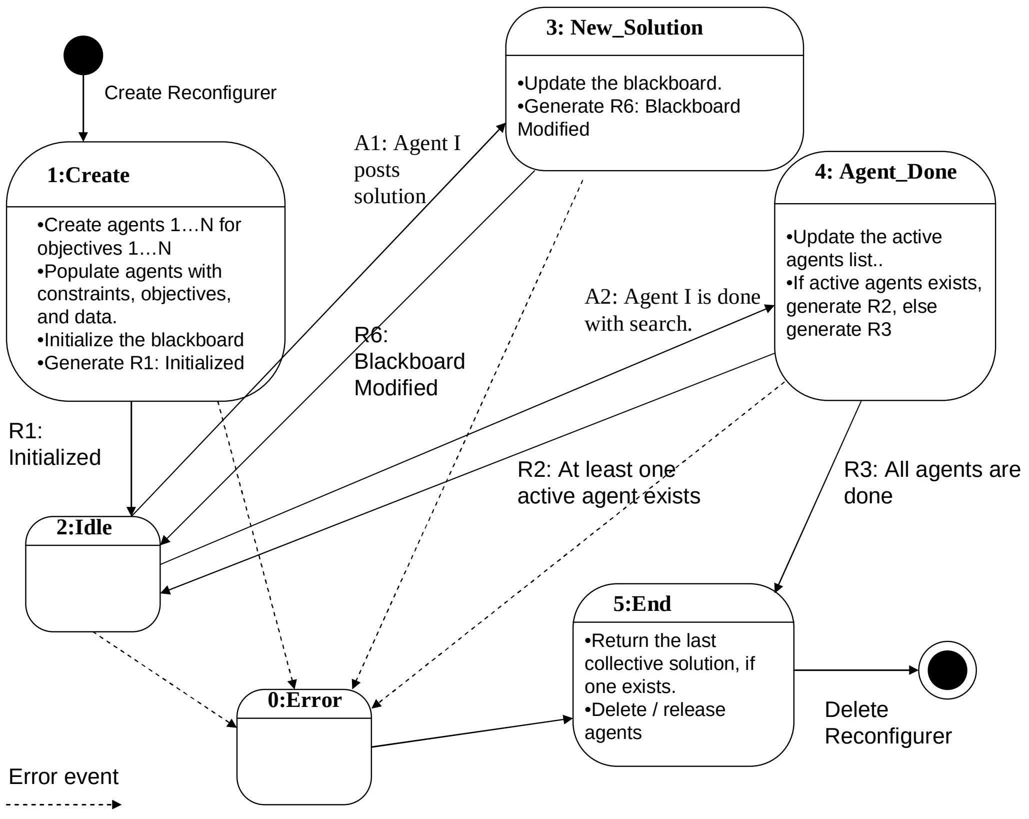

Create Reconfigurer
1:Create
•Create agents 1…N for objectives 1…N
•Populate agents with constraints, objectives, and data.
•Initialize the blackboard
•Generate R1: Initialized
R1: Initialized
2:Idle
A1: Agent I posts solution
3: New_Solution
•Update the blackboard.
•Generate R6: Blackboard Modified
R6: Blackboard Modified
A2: Agent I is done with search.
4: Agent_Done
•Update the active agents list..
•If active agents exists, generate R2, else generate R3
R2: At least one active agent exists
R3: All agents are done
0:Error
5:End
•Return the last collective solution, if one exists.
•Delete / release agents
Delete Reconfigurer
Error event

Figure 3.9: State transition diagram for reconfigurer

two agents, Agent1 and Agent2, and sends `setObjective` messages to get the agents to create and setup their respective CSP solvers. The reconfigurer and the agents run independently and utilize message ports for passing asynchronous messages back and forth. After the agents create and setup their CSP solvers, they send the `setupDone` message to the reconfigurer and wait for the next message from the reconfigurer. Upon getting the `setupDone` messages from the agents, the reconfigurer sends the `soln` messages to the agents to initiate their search for the optimization problem. In response to this message, every agent activates its CSP solver by sending the `next` message. If the CSP solver finds a solution that satisfies all the constraints of the optimization problem, it informs its parent agent with the `nextSoln` message, where the solution is encoded in the message. In return, the agent passes the solution back to the reconfigurer with the `newSoln` message. Reconfigurer sends the agent another `soln` message after recording the agent's ID and the solution in its blackboard. If the CSP solver cannot find a solution, it returns the `noSoln` message to the agent, and the agent informs the reconfigurer with the the `agentDone` message and terminates the CSP solver and itself. This loop continues until the agent traverses the whole search space for the optimization problem or the agent visits a pre-determined number of decision nodes that was set by the reconfigurer at the beginning which is called the *termination point.* When the reconfigurer gets the `agentDone` messages from all the agents, it selects the best solution according to the negotiation algorithm described in Section 3.8 and returns best solution to the main program.

Figure 3.11 shows the communication between an agent and its CSP solver in detail. An agent uses one port to communicate with the reconfigurer and another one to communicate with its CSP solver. In SCSA

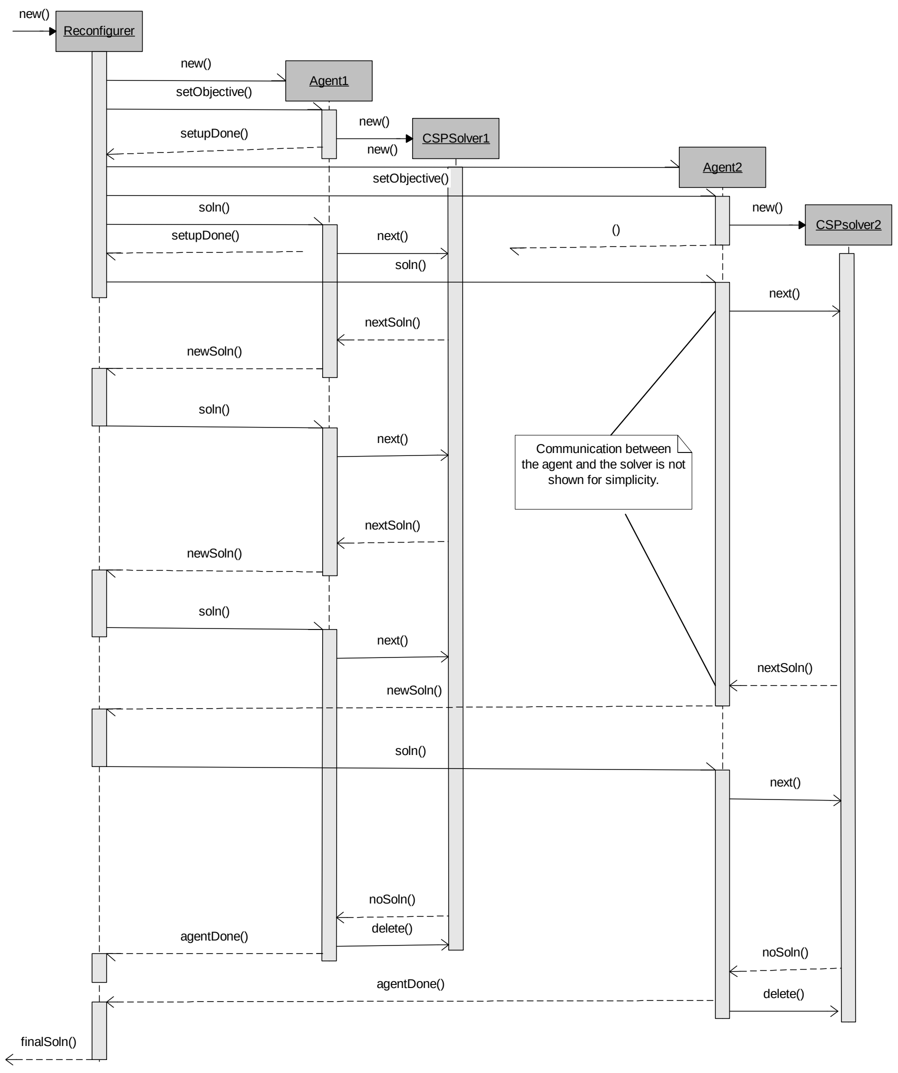


Figure 3.10: Sequence diagram for the communication among a reconfigurer and its agents

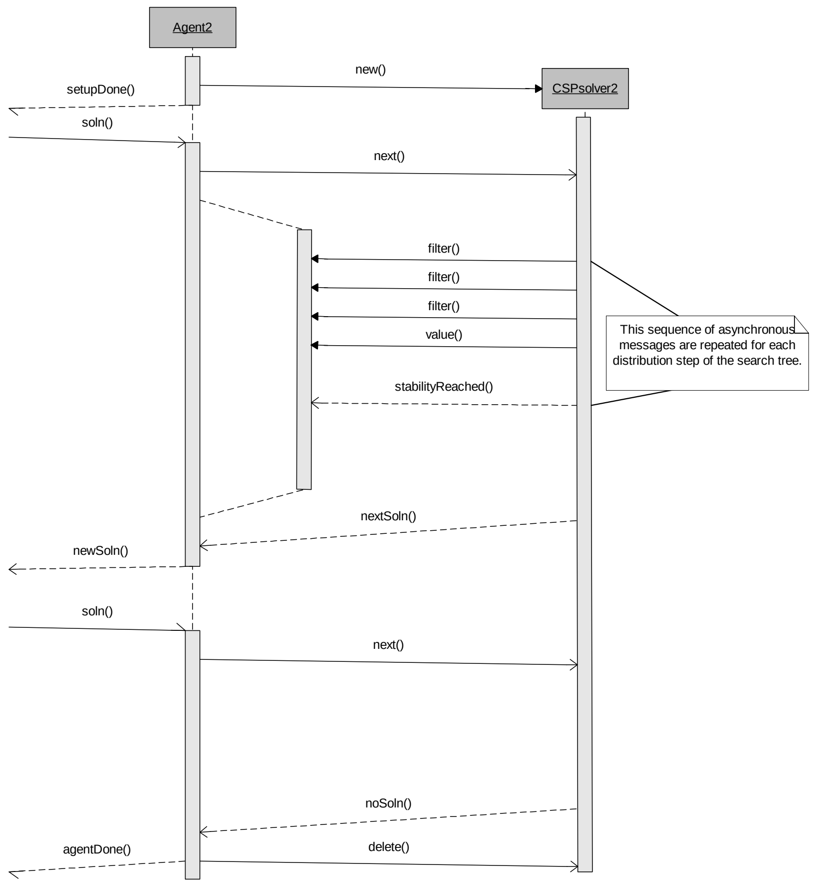


Figure 3.11: Sequence diagram for the communication between an agent and the CSP solver

the ports are implemented by different threads. After an agent starts the search in CSP solver with the `next` message, it establishes a port between itself and the CSP solver to monitor and control the search as it is described in Section 3.7.5. The CSP solver sends a sequence of messages to its agent to inform it about certain events. In addition, the CSP solver passes a number of statistics, so that the agent can adjust its control parameters and change the execution and direction of the search if it is necessary.

3.7 Formalization of the CSP Search Algorithm as a Dynamic System

As was mentioned earlier in this thesis, our goal is to use a control metaphor and architecture to control the time complexity of the search. Towards this aim we need to first conceptualize the plant as a dynamic system.

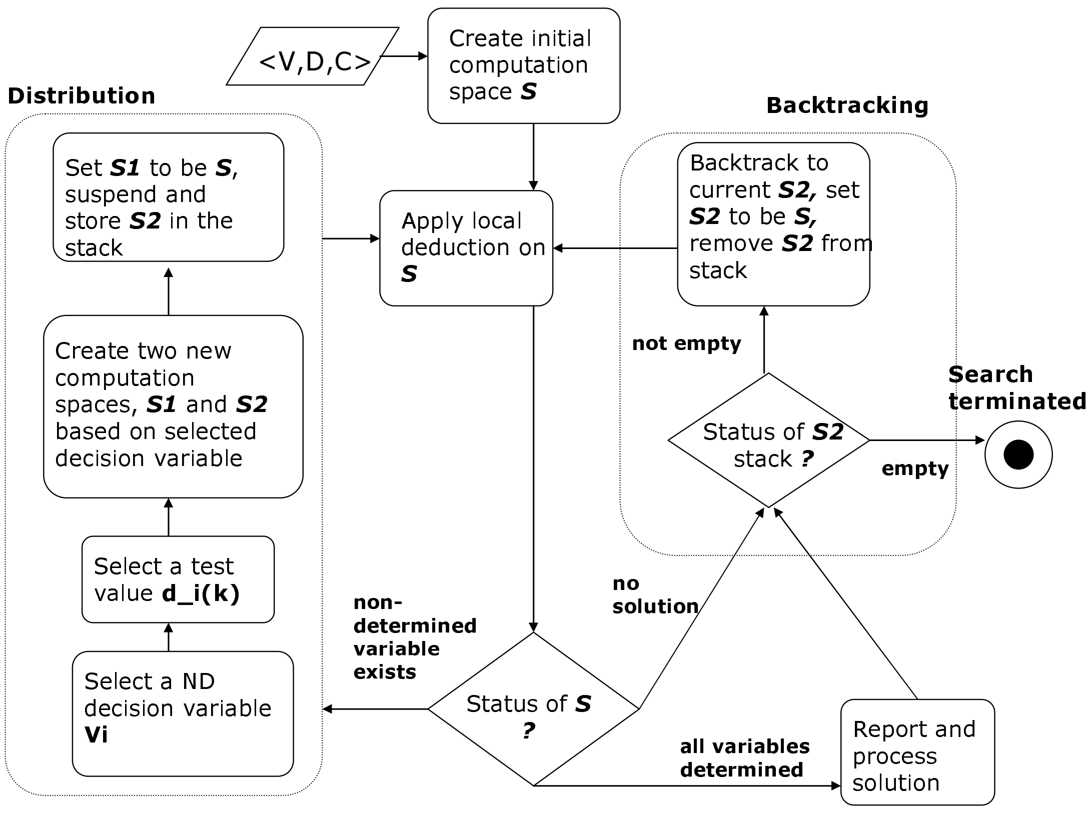
A CSP is a 3-tuple, $< V, D, C >$, where V is the set of decision variables, D is the set of possible values for the decision variables, and C is the set of constraints $\{C_1, \ldots, C_t\}$ on the decision variables [101]. A constraint C_i contains [24] a subset of the variables $var(C_i) = \{V_{i_1}, \ldots, V_{i_{j(i)}}\}$ and a relation rel_i, defined on this subset:

$C_i = rel_i(V_{i_1}, \ldots, V_{i_{j(i)}}) \subseteq D_{i_1} \times \ldots \times D_{i_{j(i)}}$.

A finite domain is a finite set of nonnegative integers. A finite domain problem FDP is specified in terms of a finite set of constraints without quantifiers (like $\forall, \exists$), such that FDP contains a domain constraint for every variable occurring in a constraint of FDP [118]. A variable assignment is a function mapping variables to integers. A solution of a FDP is a variable assignment that satisfies every constraint in FDP. *Constraint propagation* is an inference rule for finite domain problems that narrows the domains of variables.

3.7.1 A Branch and Bound Search Algorithm for Solving a CSP

In this thesis, automatically generated CSP code (in OZ) executes a search engine [117, 144] which is one of the system modules of the Mozart Programming system. The search engine can solve CSPs with and without



<V,D,C>
Create initial computation space S
Distribution
Backtracking
Set S1 to be S, suspend and store S2 in the stack
Apply local deduction on S
Backtrack to current S2, set S2 to be S, remove S2 from stack
not empty
Create two new computation spaces, S1 and S2 based on selected decision variable
Search terminated
Status of S2 stack ?
empty
Select a test value d_i(k)
no solution
non-determined variable exists
Select a ND decision variable Vi
Status of S ?
all variables determined
Report and process solution

Figure 3.12: Flowchart for the CSP search algorithm

objective functions and implements a modified version of the branch and bound algorithm introduced in Section 2.1.2.

The computation steps of the search engine, that is shown in Figure 3.12 are:

1. $(k = 0)$ **Create a computation space** S where,

   $S =< V, D, C >$

   This is the root of the search tree. k indicates the depth of the node within the search tree that is created and traversed by the search engine.

2. ($k = k + 1$) **Local deduction:** Use the updated constraints C to narrow the domains of decision variables via local deduction.

3. **Check the status of computation space** S:

   (a) If all the decision variables are determined (assigned a value), report the variable assignments. At this point, the algorithm is at a leaf node of the search tree. If more than one solution is queried or if this is an optimization problem, continue Step 5.

   (b) If there are still non-determined decision variables, continue Step 4.

   (c) If there are no possible solutions left consistent with the constraints in C, continue Step 5.

4. **Distribution:**

   (a) **Select a non-determined (ND) decision variable:** Select a decision variable $V_i \in V^{ND}$, where V^{ND} is the set of non-determined decision variables and $D_j(k)$ is the current domain of the decision variable V_j at depth k, using a selection function Sel:

   $Sel : 2^{V^{ND}} \rightarrow V^{ND}$ where,
   $V_i = Sel(2^{V^{ND}})$
   $V^{ND} = \{V_j \in V | card(D_j(k)) > 1\}$

   (b) **Select a test value:** Select a value or a domain specification $d_i(k)$ within the current domain $D_i(k)$ of the selected decision variable V_i, such that $d_i(k) \subset D_i(k)$, using a distribution strategy $U(D_i(k))$:

$U(D_i(k)) : 2^{D_i(k)} \rightarrow D_i(k)$ or $U(D_i(k))(2^{D_i(k)}) = d_i(k)$
$d_i(k)$ is a set of a single integer, $card(d_i(k)) = 1$.

(c) **Create a choice point for the selected decision variable** V_i:

i. Create two new computation spaces by adding new constraints,

$S_1(k) =< V, D(k), C(k) \bigcap (V_i \in d_i(k)) >$
$S_2(k) =< V, D(k), C(k) \bigcap (V_i \in (D_i(k) \setminus d_i(k))) >$
where,
$C(k)$ is the intersection of the initial constraints in C and the constraints added at each choice point before this state. This will extend the search tree one more level.

ii. Continue Step 2 with the new computation space $S_1(k)$.

$S = S_1(k)$

iii. Store the new computation space $S_2(k)$ in a computation space stack.

5. **Backtracking**: Backtrack the search tree by removing the first computation space from the computation space stack and making it S.

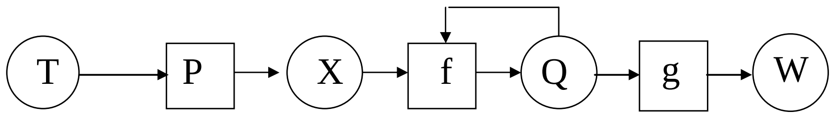


Figure 3.13: The structure of the general dynamic system

If the stack is empty, it means that the entire search tree has been traversed for a solution and the search is over.

3.7.2 General Dynamic Systems

A general dynamic system [97] GDS is an 8-tuple

$$GDS = (T, X, W, Q, P, f, g, \leq) \tag{3.1}$$

where

- T is time set with an order relation $\leq$,
- X and W are the input and output sets, respectively,
- Q is the set of inner states q,
- P are the input processes, functions, $p : T \rightarrow X$,
- f is the state transition function, $f : T \times Q \times p \rightarrow Q$, and
- g is the output function, $g : Q \rightarrow W$.

Figure 3.13 represents the structure of a general dynamic system. In this figure, the circles represent the sets, T, X, Q, and W. The rectangles represent the functions and processes, P, f, and g.

The state transition function f must satisfy the following properties:

1. (consistency) $f(t_0, t_0, q_0, p) = q_0$
2. (semigroup) $f(t_2, t_1, f(t_1, t_0, q_0, p), p) = f(t_2, t_0, q_0, p)$
3. (causality) $f(t, t_0, q, p) = f(t, t_0, q, p_1)$, if $p(\tau) = p_1(\tau)$, for $t_0 < \tau \leq t$.

3.7.3 Modeling the CSP Search Algorithm as a Dynamic System

The branch and bound search algorithm used in this thesis can be formulated as a dynamic system:

- The search algorithm is a discrete system and it is time-invariant.
- The input X is the test value $d_i(k)$ at each distribution step at depth k, where $d_i(k)$ is a set of a single integer.
- W is the count of non-determined decision variables.
- Q is the set of internal states, q, of the dynamic system, s.t.
  - $q \subset D^N = D_1 \times D_2 \times \ldots \times D_N$, where N is the number of decision variables in the CSP,
  - D_i is the initial domain $D_i(k = 0)$ for the decision variable V_i,
  - The state q can also be represented by an intersection of three sets of constraints; a) C_{D_i}, constraints that specify the initial domains of the decision variables V_i, b) C_l, original constraints that are parts of C, and c) $C(k)$, constraints that are added to the CSP at each choice point.
    $q(k) = C_{D_1} \bigcap C_{D_2} \bigcap \ldots \bigcap C_{D_N} \bigcap C_1 \bigcap C_2 \bigcap \ldots \bigcap C_M \bigcap$
    $C(1) \bigcap C(2) \bigcap \ldots \bigcap C(k-1)$

- P, the input processes are, decision variable selection function Sel and the distribution strategy $U(D_i(k))$.
- g is the output function that returns the value assignments for the decision variables,
  $g(q(k)) = D_1(k) \times D_2(k) \times \ldots \times D_N(k)$
- Since the search algorithm is time-invariant, state transition function f reduces to:
  $f : Q \times X \rightarrow Q$

$$q(k+1) = \begin{cases} q(k) \bigcap C(k), & \text{If search is moving down a choice point} \\ q(k-1), & \text{If backtracking} \end{cases} \tag{3.2}$$

where,

$$C(k) = \begin{cases} V_i \in d_i(k), & \text{If } S_1 \text{ is selected at the choice point} \\ V_i \in (D_i(k) \setminus d_i(k)), & \text{If } S_2 \text{ is selected at the choice point} \end{cases} \tag{3.3}$$

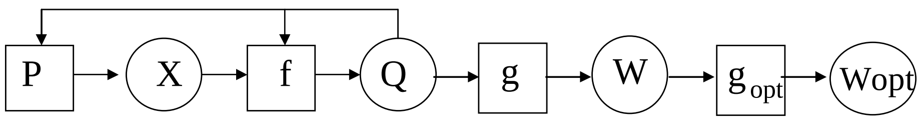


Figure 3.14: Extended CSP search algorithm modeled as a dynamic system

3.7.4 Modifying the Model to Handle CSPs with Objective Functions

If the CSP is extended with an objective function, which ranks the solutions if there are more than one, the change is made in Step 3a of the search algorithm defined in Section 3.7.1.

Figure 3.14 shows the modified version of the dynamic system model defined in Section 3.7.3 to accommodate the CSP search algorithm with an objective function:

$$Search = (X, W, W_{Opt}, Q, P, f, g, g_{opt}) \tag{3.4}$$

where,

- g_{opt} is the objective function
- W_{opt} is the value of the objective function for a given value assignment

Since the search algorithm is time-invariant, there is no time set T. However, input processes P take the internal state Q as an input and the arrow from the state circle to the input processes rectangle represents that relation.

3.7.5 The Control of Search by an Agent

This section describes the control strategy used in each agent of the SCSA to control the complexity of the satisficing program. It starts with an investigation of search parameters that affect the direction of search algorithm within the search tree. Next, the results of the investigation are used to formally define a control strategy with the control variable, the feedback,

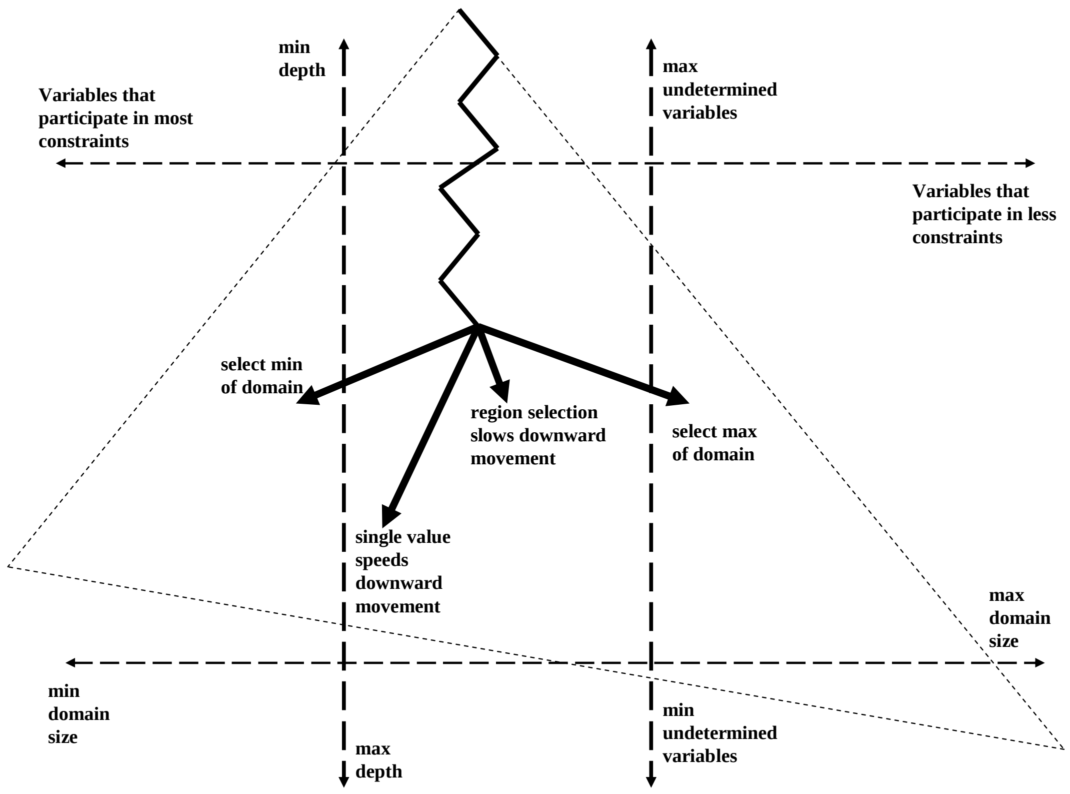

min depth
Variables that participate in most constraints
max undetermined variables
Variables that participate in less constraints
select min of domain
region selection slows downward movement
select max of domain
single value speeds downward movement
max domain size
min domain size
max depth
min undetermined variables

Figure 3.15: Search tree constructed by the branch and bound algorithm and the search parameters

and the control equation. The discussion ends with a brief explanation of the control implementation used in SCSA.

The complexity of the satisficing program is a function of the size of the search tree that is constructed by the branch and bound search algorithm described in Section 3.7.1. If it is desired to reach the sections of the search tree concentrated with feasible solutions without traversing the entire tree, then one needs to find a set of search parameters that controls the direction of the search and a measure or indicator that points to the sections with concentrated feasible solutions.

Figure 3.15 shows a search tree constructed by the branch and bound algorithm and the effects of some of the search parameters on the direction of the search. In the figure the triangle with the dashed perimeter represents the search tree with the root node being at the top. The primary factor in changing the search direction is the selection function Sel that picks the next decision variable to be distributed among the non-determined decision variables at the current decision node. Different selection functions cause the search to move side to side horizontally. Two of the widely used selection functions in the branch and bound algorithm are; 1) *MinimumSuspensions*, the function that selects the non-determined decision variable that participates in most of the constraints and 2) *MinimumDomain*, the function that selects the non-determined decision variable with the smallest current domain $D_i(k)$.

The secondary factor in changing the search direction is the distribution strategy U, which is a function of the current domain $D_i(k)$ of the selected non-determined decision variable V_i at depth k. The effect of U is local relative to the selection function Sel. However, U causes changes in speed of the search in addition to the direction. If U returns a single integer as the test value for V_i, then search becomes a depth-first search and the vertical progress in the search tree gets faster. On the contrary, if U returns a subset of $D_i(k)$, then the search becomes a breadth-first search and the vertical progress slows down. In the case of returning a single integer, the relative position of the selected integer within the current domain, which is an ordered collection of integers, is called the *split point.* The split point κ takes a value between 0.0 (associated with the smallest integer within the current domain) and 1.0 (associated with the largest integer within the current domain) and contributes to the horizontal move.

Experimentation showed that `MinimumSuspensions` as the selection function and returning a single integer in the distribution function resulted in faster search results. Therefore, SCSA primarily uses the `MinimumSuspensions` function. When more than one decision variable participate in maximum number of constraints, SCSA uses `MinimumDomain` function to break the tie condition.

However, there is not one magic value for the split point κ that shows an improvement on the performance for all search trees. Therefore, κ is selected as a variable to be controlled by the control strategy used in SCSA.

Next, a feedback variable is defined. Feedback is used as a measure that points to the sections with concentrated feasible solutions to the satisficing problem. The goal is to steer the search towards the sections of the search tree, where the percentage of feasible solution leaves are higher than the percentage of non-solution leaves. As a result, the feedback is defined as in Equation 3.5, where S is the number of feasible solution leaves and F is the number of non-solution leaves visited in the last control cycle. Here, feedback can take any value between 1.0 (all feasible solutions) and -1.0 (no feasible solutions).

$$feedback = \frac{S - F}{S + F} \tag{3.5}$$

After feedback, a control equation is defined for the calculation of the control variable κ. Figure 3.16 demonstrates the Proportional-Derivative (PD) control used in SCSA. Equation 3.6 details the calculation of the control variable κ, where $\Delta = goal - feedback$, K_P and K_D are the proportional and derivative control constants respectively:

$$\kappa_{n+1} = \kappa_n + K_P \times \Delta + K_D \times \frac{\delta feedback}{\delta n} \tag{3.6}$$

The control strategy is implemented in each agent. The selected CSP solver needs to be modified to take the control variable κ as a dynamic

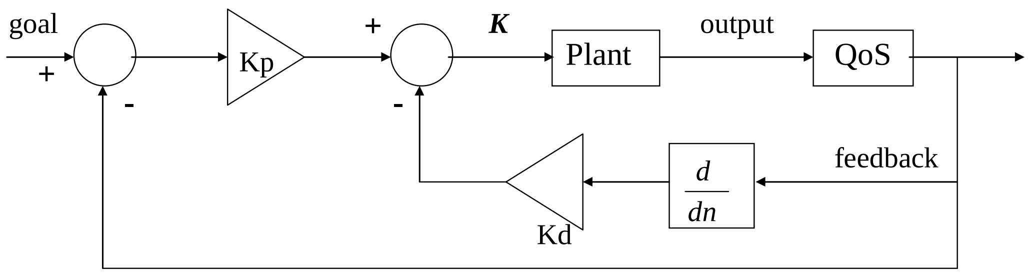


Figure 3.16: PD control in SCSA

input and needs to provide access to vital search statistics. Some of the important search statistics are the number of decision points, the number of feasible solutions and the number of non-solutions visited in the last control cycle, and the depth of the current decision point. As part of this thesis, `Search.oz` file in the Mozart system library was modified to implement the control strategy in OZ.

Finally, a termination policy for monitoring the status of the CSP solver and detecting phase transition behavior in potentially hard problems is defined. The search statistics obtained from the CSP solver is used to calculate the number of decision points Δdp visited since the last significant event. The significant event can be either the beginning of the search, if no feasible solutions are found so far, or the last feasible solution, if there is one found. If any point in the search Δdp exceeds a *termination point* set by the reconfigurer, the agent returns an `agentDone` message to the reconfigurer and terminates the search.

The results from the experimental systems described in Sections 4.2 and 4.3 showed that the selection of the initial value κ_0 for the control variable is very important for the performance of the search.

Table 3.5: Listing for function NegotiateSolution

```
func NegotiateSolution(LIST blackboard)
   LIST agentStacks[nObjectives], nonDominatedSolns

   if blackboard.count == 0 then
      return NO-SOLUTION
   endif

   agentStacks = SplitSolutionsByAgent(blackboard)
   nonDominatedSolns = FindNonDominatedSolutions(agentStacks)

   return OrderNonDominatedSolutions(agentStacks,nonDominatedSolns)
end
```

3.8 A Mediator-based Negotiation Algorithm

In this thesis, negotiation is used to determine a value assignment of the decision variables of the optimization problem when the objective functions conflict with each other. The goal of the negotiation is to find the best-compromise non-dominated solution from the non-dominated solutions set as it is defined in Definition 2.1.5 based on a negotiation criterion.

In SCSA the reconfigurer is the mediator for the negotiation. The reconfigurer records the solutions that are returned by all agents in a list called `blackboard`. Every record contains a solution and the ID of the agent that has returned this solution. If an agent has stopped its search for a solution prematurely because of reaching the termination point, the last record in the stack for that agent contains a premature termination flag instead of a solution. The negotiation algorithm does not rely on all the agents to be done with the search. The reconfigurer can select a solution on demand of the main program as long as there is at least one solution recorded in the blackboard.

Table 3.5 lists the function `NegotiateSolution`, which is the high level function that implements the negotiation algorithm. First, the algorithm splits the solutions listed in the `blackboard` into separate lists, `agentStacks` in this function, based on the agent that has returned them. The solutions in each list in `agentStacks` are ordered from best to worst by the value of the objective function that is implemented by that agent. Next, the negotiation algorithm finds the non-dominated solutions among the ones listed in `agentStacks`. Finally, the algorithm orders and selects the best-compromise non-dominated solution according to the negotiation criterion.

The following sample multiobjective optimization problem will help demonstrating the details of the algorithm:

$$\begin{array}{lll} \text{Max} & O_1 & = 3 \cdot X + Y \\ \text{Max} & O_2 & = X \cdot (10 - X) + Y \\ \text{s.t.} & X & \neq Y \\ & X & \in \{1 \ldots 9\} \\ & Y & \in \{1 \ldots 9\} \end{array}$$

Table 3.6 displays a partial list of the solutions found for the multiobjective optimization problem defined above. This is the `blackboard` list that is passed to function `NegotiateSolution` as an input. The solution at the top of the table represents the last solution returned by any agent.

After the solutions in `blackboard` are split into separate lists, the new lists are passed to function `FindNonDominatedSolutions` to find the non-dominated solutions. Table 3.7 shows a listing of `FindNonDominatedSolutions`. `FindNonDominatedSolutions` calls function `TestSolutionForNonDominance`

Table 3.6: Partial listing of the solutions for the optimization problem defined by $O_1 = 3 \cdot X + Y$ and $O_2 = X \cdot (10 - X) + Y$

Agent ID	Soln[X,Y]	$O_1 = 3 \cdot X + Y$	$O_2 = X \cdot (10 - X) + Y$
1	9,8	35	17
1	9,7	34	16
2	5,9	24	34
2	4,9	21	33
1	8,9	33	25
1	8,7	31	23
2	4,8	20	32
2	4,7	19	31
1	7,9	30	30
2	3,9	18	30
1	7,8	29	29
2	3,8	17	29
1	6,9	27	33

Table 3.7: Listing for function FindNonDominatedSolutions

```
func FindNonDominatedSolutions(LIST agentStacks)
    LIST nonDominatedSolns

    for I = 1 to nObjectives do
        for S = 1 to agentStacks[I].count do
            TestSolutionForNonDominance(agentStacks[I,S], nonDominatedSolns)
        endfor
    endfor

    return nonDominatedSolns
end
```

Table 3.8: Listing for function TestSolutionForNonDominance

```
proc TestSolutionForNonDominance(sol, LIST nonDominatedSolns)
  BOOL isNDsolution = true
  if nonDominatedSolns.count == 0 then
    nonDominatedSolns = sol
  elseif sol != prematurelyTerminated then
    for S = 1 to nonDominatedSolns.count do
      comparisonRes = CompareSolutions(nonDominatedSolns[S], sol)
      if comparisonRes == BETTER then
        isNDsolution = false
      elseif comparisonRes == WORSE then
        ListRemove(nonDominatedSolns, nonDominatedSolns[S])
      endif
    endfor
    if isNDsolution == true then
      ListAppend(nonDominatedSolns, sol)
    endif
  endif
end
```

to compare a given solution `agentStacks[I,S]`, which is the S^{th} best solution returned by the I^{th} agent, against all other solutions that are found to be non-dominated up to that point. If the current solution `agentStacks[I,S]` is found to be a non-dominated solution by `TestSolutionForNonDominance`, then the current solution is added to the `nonDominatedSolns` list as well.

Table 3.8 shows a listing of function `TestSolutionForNonDominance`. Two inputs to `TestSolutionForNonDominance` are `sol`, the next solution that is tested to be non-dominated or not and `nonDominatedSolns`, the list of solutions that are considered to be non-dominated up to this point. In `TestSolutionForNonDominance`, `sol` is compared against each solution in `nonDominatedSolns` using function `CompareSolutions`. (See Table 3.10 for a listing of `CompareSolutions`.) If `sol` is better than a solution (`nonDominatedSolns[S]`) in `nonDominatedSolns`, then `nonDominatedSolns[S]` is removed from `nonDominatedSolns`. If none of the solutions in `nonDominatedSolns` are better than `sol`, then `sol` is added the `nonDominatedSolns` list.

Table 3.9 displays the complete list of the non-dominated solutions found by `FindNonDominatedSolutions` for the multiobjective optimization problem defined above.

After the non-dominated solutions $NonDom$ are determined, function `OrderNonDominatedSolutions` (listed in Table 3.11) selects the best-compromise solution among them. `OrderNonDominatedSolutions` uses an *imitative* tactic as a selection criterion. (See Section 2.1.7 for a brief definition for imitative tactics.) Equation 3.7 summarizes the selection criterion used by `OrderNonDominatedSolutions`, where $OV_i(S)$ is the value of the i^{th} objective function for solution $S = [X_1, \ldots, X_N]$, O is the number of objectives, and S_{i_B} is the solution that gives the best value for OV_i. Using this algorithm, `OrderNonDominatedSolutions` selects a solution where each objective function OV_i compromises the least from its best possible value. This is similar to the *Utopian Approach* [146], where the agents' best-compromise solution is defined as the agents' most preferable solution that is closest to the utopian point and the utopian point is

Table 3.9: Non-dominated solutions for the optimization problem defined by $O_1 = 3 \cdot X + Y$ and $O_2 = X \cdot (10 - X) + Y$

Agent ID	Soln[X,Y]	$O_1 = 3 \cdot X + Y$	$O_2 = X \cdot (10 - X) + Y$
1	9,8	35	17
1	8,9	33	25
1	7,9	30	30
1	6,9	27	33
2	5,9	24	34

Table 3.10: Listing for function CompareSolutions

```
func CompareSolutions(sol1, sol2)
    BOOL allBetter = true, allWorse = true
    for I = 1 to nObjectives do
        if GetObjectiveValue(I, sol1) $\geq$ GetObjectiveValue(I, sol2) then
            allWorse = false
        else
            allBetter = false
        endif
    endfor
    if allBetter = true then return BETTER
    elseif allWorse = true then return WORSE
    else return NON-DOMINATED
    endif
end
```

defined as $z^* = [S_{1_B}, S_{1_B}, \ldots, S_{O_B}]$.

$$\begin{aligned} \min \quad SumOfSquares \quad &= \sum_{i=1}^{i=O} \left(\frac{OV_i(S) - OV_i(S_{i_B})}{OV_i(S_{i_B})}\right)^2 && (3.7) \\ \text{s.t.} \quad S \quad &\in NonDom && (3.8) \end{aligned}$$

If each agent returns a single solution and they are different, then the algorithm picks the solution that arrived last.

In the example problem above, `OrderNonDominatedSolutions` selects the solution $[X = 7, Y = 9]$ which gives objective values of $OV_1([X = 7, Y = 9]) = 30$ and $OV_2([X = 7, Y = 9]) = 30$, as the best solution.

Table 3.11: Listing for function OrderNonDominatedSolutions

```
func OrderNonDominatedSolutions(LIST agentStacks,
                                LIST nonDominatedSolns)
   INT bestSolIdx = 1
   REAL bestSumOfSquares = $\infty$
   for S = 1 to nonDominatedSolns.count do
      REAL sumOfSquares = 0.0
      for I = 1 to nObjectives do
         REAL bestValue = GetObjectiveValue(I, agentStacks[I,1])
         REAL newValue = GetObjectiveValue(I, nonDominatedSolns[S])
         REAL percentChange = (bestValue - newValue)/ bestValue
         sumOfSquares = sumOfSquares + percentChange * percentChange
      endfor
      if bestSumOfSquares < sumOfSquares then
         bestSumOfSquares = sumOfSquares
         bestSolIdx = S
      endif
   endfor

   return nonDominatedSolns[bestSolIdx]
end
```

Chapter 4

Verification and Demonstration

4.1 Introduction

An experimental system was implemented and tested on two scenarios to evaluate the approach specified in this thesis; a job scheduling problem (see Section 4.2) and a fixture design utility problem (see Section 4.3). Problem formulations were specified using the UML/OCL representation. The formulations were automatically translated to the selected Constraint Satisfaction Problem (CSP) language OZ and then used by the system to find satisficing solutions. The goal was to provide a proof-of-concept of automating the development of such a self-controlling satisficing program that (1) was applicable to various multiobjective optimization problems and (2) had the ability to control its own complexity by controlling the search direction and detecting phase transition regions to terminate the search gracefully. The first requirement was satisfied by demonstrating that the same system can be used in two different problem domains. The second requirement was satisfied by testing the experimental system on both hard and easy regions in order to show its ability to detect phase transition regions. In each of the scenarios, the phase transition phenomenon was investigated and a model for phase transition was developed. Later, these models were used to generate test data for the experimental system. Known benchmark approaches implemented in Mozart Programming System version 1.3.1 were used as a reference.

4.2 Job Scheduling System

4.2.1 Problem Statement

This experiment deals with a job scheduling problem (see Figure 4.1). Each job consists of one or more tasks and has a due date which all the tasks for the job need to be completed by. Each task requires a single resource and takes a certain time to be completed. There is only one of each resource type and a resource can not be shared by more than one task at the same time. The objectives are to complete all the jobs as soon as possible and to prevent any given job to starve or to wait for other jobs to be completed.

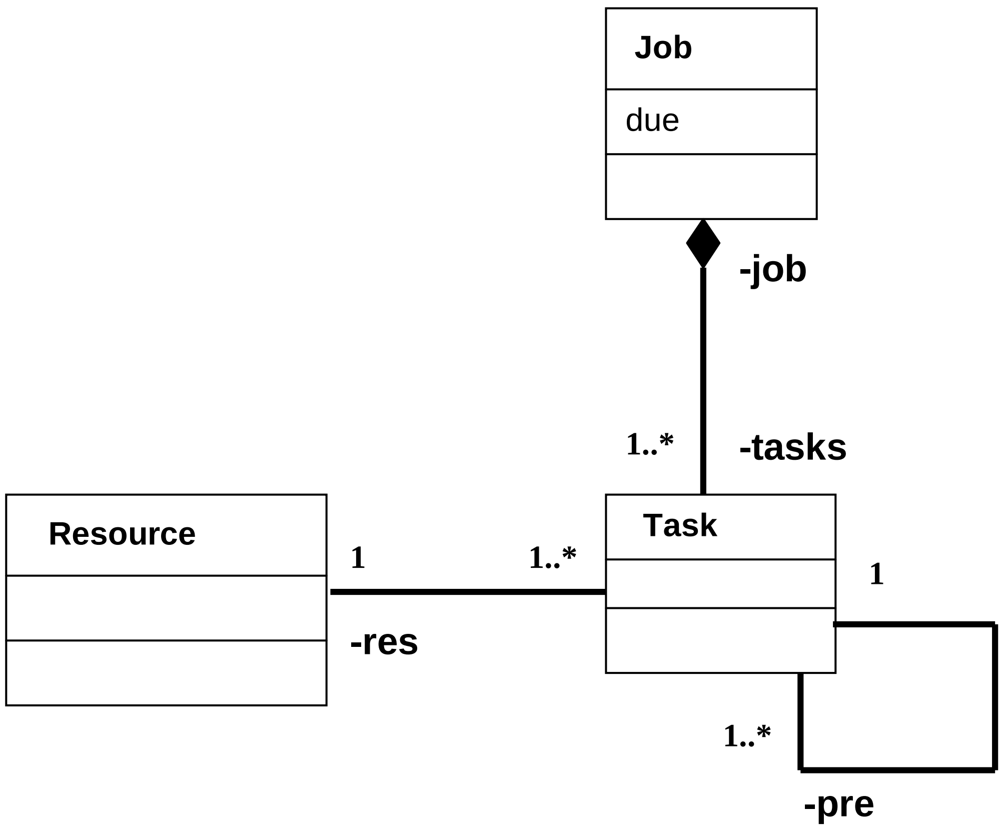


Figure 4.1: Class Diagram for the Job Scheduling System

4.2.2 Problem Formulation

To formalize the objective functions and the constraints used in the job scheduling experiment, we introduce the following notation:

- T - the set of all tasks
- J - the set of all jobs
- R - the set of all resources
- $Dur : T \rightarrow N$ - a function that returns the duration of a given task. All durations are expressed in terms of natural numbers.
- $Pre : T \rightarrow 2^T$ - a function that returns the set of pre-requisite tasks for a given task.
- $Due : J \rightarrow N$ - a function that returns the due date or deadline for the completion of a given job. All time values are expressed in terms of natural numbers.
- $LastTask : J \rightarrow T$ - a function that returns the last task required for the completion of a given job.
- $Res : T \rightarrow R$ - a function that maps a task to a resource. For the simplicity of the problem, a task depends on a single resource.
- $Start : T \rightarrow N$ - a function that returns the starting date for a given task. In our problem, starting dates for the tasks are the decision variables.

Objective 1: Minimize the end date for the last job that is completed.

$$\begin{array}{lll}
\text{Min} & \text{EndDate} & = \max_{t \in T}(\mathit{Start}(t) + \mathit{Dur}(t)) \\
\text{s.t.} & \forall t \in T, \forall p \in \mathit{Pre}(t): & \mathit{Start}(p) + \mathit{Dur}(p) \leq \mathit{Start}(t) \\
 & \forall j \in J, p = \mathit{LastTask}(j): & \mathit{Start}(p) + \mathit{Dur}(p) \leq \mathit{Due}(j) \\
 & \forall t_1, t_2 \in T: & \text{if } \mathit{Start}(t_1) \leq \mathit{Start}(t_2) \\
 & & \text{and } \mathit{Start}(t_2) \leq \mathit{Start}(t_1) + \mathit{Dur}(t_1) \\
 & & \Rightarrow \mathit{Res}(t_1) \neq \mathit{Res}(t_2)
\end{array}$$

Objective 2: Minimize the average time to complete a job.

$$\begin{array}{lll}
\text{Min} & \text{AvgEndDate} & = \frac{\sum_{j \in J} \mathit{Start}(\mathit{LastTask}(j))}{|J|} \\
\text{s.t.} & \forall t \in T, \forall p \in \mathit{Pre}(t): & \mathit{Start}(p) + \mathit{Dur}(p) \leq \mathit{Start}(t) \\
 & \forall j \in J, p = \mathit{LastTask}(j): & \mathit{Start}(p) + \mathit{Dur}(p) \leq \mathit{Due}(j) \\
 & \forall t_1, t_2 \in T: & \text{if } \mathit{Start}(t_1) \leq \mathit{Start}(t_2) \\
 & & \text{and } \mathit{Start}(t_2) \leq \mathit{Start}(t_1) + \mathit{Dur}(t_1) \\
 & & \Rightarrow \mathit{Res}(t_1) \neq \mathit{Res}(t_2)
\end{array}$$

4.2.3 Goals and Constraints Presented in OCL

```
package JobSchedule


context Scheduler def:
    let Tasks : Sequence(Task) = Data::tasks
    let Jobs : Sequence(Product) = Data::jobs
    let DurList : Sequence() = Tasks->iterate(T : Task;
            AccD : Sequence() = Sequence{} |
        AccD->append( ONT_Field(ONT_Label(T), T.dur) ) )
```

```
-- Group the tasks based on the resource that they use.

context Scheduler::GetTasksOnResource(Ts : Sequence(Task)) :
           ONT_RECORD
    def:
    let D = ONT_NewDictionary()
    pre: Ts->iterate(T |
        if ONT_HasField(T, res)
        then ONT_DictionaryPut(D, T.res, ONT_Append(ONT_Label(T),
            ONT_DictionaryGet(D, T.res, nil)))
        endif)
    post: ONT_DictionaryToRecord(tor, D)

--  Allocate memory for the decision variables.

context Scheduler::GetStart(Ts : Sequence(Task)) : ONT_RECORD
    post:
    let TaskNames = ONT_Map(Ts, Label)
    in ONT_NewRecord(start, TaskNames, ONT_Field(0,FD.sup))

--  List of all constraints

context Scheduler::Constraints (Start : Sequence(Integer)) :
                ONT_PROC
    def:
    let DurRecord : ONT_RECORD = ONT_ListToRecord(dur, DurList)
    pre: PC = GetStart(Tasks)
    pre: Constraint1(DurRecord, PC)
    pre: Constraint2(DurRecord, PC)
    pre: Constraint3(DurRecord, PC)
    post: ONT_Dist(PC)

-- Presedence constraints.

context Scheduler::CheckPreTasks(PreTasks : Sequence(Task),
            L : String, DurRecord : ONT_RECORD, Start : ONT_RECORD)
    post:
    PreTasks->iterate(P |
        ONT_LessThanEqual(Start.P + DurRecord.P, Start.L))

context Scheduler::Constraint1(DurRecord : ONT_RECORD, Start :
                  ONT_RECORD)
    post: Tasks->iterate(T |
        if ONT_HasField(T, pre)
        then CheckPreTasks(T.pre, ONT_Label(T), DurRecord, Start)
        endif)

-- Duration and job due date constraints

context Scheduler::CheckDueDate(DurRecord : ONT_RECORD,
        Start : ONT_RECORD,
        J : Job)
    def:
    let P : String = J.job
    post: ONT_LessThanEqual(Start.P + DurRecord.P, J.due)

context Scheduler::Constraint2(DurRecord : ONT_RECORD, Start :
               ONT_RECORD)
    post: Jobs->iterate(J  |
        CheckDueDate(DurRecord, Start, J))

context Scheduler::Constraint3(DurRecord : ONT_RECORD, Start :
               ONT_RECORD)
    def:
    let TasksOnResources : ONT_RECORD = GetTasksOnResource(Tasks)
    post: ONT_SerializedDisjoint(TasksOnResources, Start, DurRecord)

--  Find the completion time of the last job. --      Assumption
is that the input is a record of integer completion dates.

context Scheduler::obj1Value(Ts : ONT_RECORD) : Integer
    def:
    let TaskList = ONT_RecordToList(Ts)
    post:
    let Max = TaskList->iterate(TItem; AccM : Integer = 0 |
        ONT_Max(AccM , TItem))
    in Max

-- First objective function that minimizes the completion date
-- of tasks.
```

```
context Scheduler::objective1()
    post: ONT_Minimize(obj1Value)

-- Find the average completion time for all jobs.

context Scheduler::JobCompletionDate(Ts : ONT_RECORD, Jname :
                 String) : Integer
    post: Ts.Jname

context Scheduler::obj2Value(Ts : ONT_RECORD) : Integer
    post:
    let Total = Jobs->iterate(J; AccM : Integer = 0  |
        ONT_Inc(AccM, JobCompletionDate(Ts, J.job)))
    in Total / Jobs->size

-- Second objective function that minimizes the average
-- completion job for all jobs.

context Scheduler::objective2()
    post: ONT_Minimize(obj2Value)

endpackage
```

4.2.4 Empirical Calculation of the Phase Transition Regions

The search for phase transition variables, or *order parameters*, as Cheeseman refers to, for the job scheduling problem started with an analysis of the test parameters; number of jobs, number of tasks, duration of tasks, due date for jobs, number of resources, number of tasks contained in each job. Analysis and experimentation show that computation time to find lack of a solution or at least one solution depends on the size of the problem as well as the combination of certain test parameters.

The upper bound of the overall size of the search space for the job scheduling problem can be expressed by Equation 4.1, where $Tasks(j) : J \rightarrow 2^T$, a function that returns the subtasks of a job j.

$$\overline{Space} = \prod_{j \in J} (\prod_{t \in Tasks(j)} (Due(j) - \sum_{p \in Pre(t)} Dur(p) - Dur(t))) \tag{4.1}$$

This upper bound can be further lowered, if the resources used by the tasks are limited. In the job scheduling problem there is one of each resource type and a resource can not be shared by multiple resources. This limitation on the resources is a critical factor for the computation time. Equation 4.2 shows the initial candidate variable V_{R_C} that was tested to be the phase transition variable, where $Res^{-1}(r) : R \rightarrow 2^T$ is a function that returns all the tasks that use the resource r and R_C is the critical resource, which is used by most of the tasks.

$$V_{R_C} = \frac{\sum_{t \in Res^{-1}(R_C)} Dur(t)}{Due(j)} \tag{4.2}$$

When $V_{R_C} > 1.0$, there is no solution. However, the definition of V_{R_C} does not count for the prerequisite tasks of the tasks that use R_C. Even if $V_{R_C} \leq 1.0$, it is not guaranteed that a solution exists. Considering the existence of prerequisite tasks, the definition of V_{R_C} was modified as in Equation 4.3, where t_C is the critical task, which uses the resource R_C and has the prerequisite tasks with the longest durations and j_C is the critical job, which has the earliest due date.

$$V^p_{R_C}(J,T,R) = \frac{\sum_{t \in Res^{-1}(R_C)} Dur(t) + \sum_{t' \in Pre(t_C)} Dur(t')}{Due(j_C)} \tag{4.3}$$

When the experimental system ran on a set of test data generated with $V^p_{R_C} \sim 1.0$, it started to show phase transition behavior.

4.2.5 Experimental Setup

A test data generator was implemented based on the empirical results obtained in Section 4.2.4. The data generator required six test parameters; 1) the number of jobs, 2) the due date for each job, 3) the maximum number of tasks involved in a single job, 4) the maximum duration of each task, 5) the maximum number of different types of resources, and 6) the maximum number of resources in each type.

The test data generator was configured to use fixed values for the number of jobs (4), the maximum number of tasks involved in a job (7), the maximum duration of each task (10), the maximum number of resources in each type (1), and the number of different types of resources (5). This case wasn't as hard as the well-known 10 jobs on 10 machines job-shop scheduling problem, but it was sufficient enough to demonstrate the phase transition regions. Next, 50 problem instances were selected among many instances of the job scheduling problem that were auto-generated by the test data generator. The results of a benchmark CSP solver program that utilized the Explorer utility in the Mozart environment were used as a reference. (See Appendix A.4 for a complete listing of the benchmark program). The 50 problem instances included; i) the instances that have at least one solution, ii) the instances that do not have a solution that satisfies all the constraints, and iii) the instances that demonstrate a phase transition behavior.

The benchmark program, similar to this experimental system, used a generic distribution strategy. (See Section 3.7.1 for a brief description of various distribution strategies in search). The order for the next decision variable to be distributed was determined based on the number of constraints that depended on that decision variable. If more than one decision variable had the maximum number of constraints, that depended on them, then the benchmark program selected the decision variable whose domain was minimal. Next, the benchmark program selected the minimum value of the domain of the decision variable as the test value.

A PC with Intel Centrino 1.4 GHz microprocessor and 512 MB RAM was used to run the experiments. A problem instance that caused a virtual memory error in the PC due to extensive computation was regarded as a hard case with phase transition behavior. A typical hard case gave a virtual memory error after the search engine traversed 1.3M decision points or distribution steps.

4.2.6 Analysis of the Results

The primary goal of the job scheduling experiments was to measure the performance of the experimental system in terms of the number of computation steps to find at least one solution. To eliminate the time variation related to the PC environment (speed, memory), distribution steps, which are discussed in Section 3.7.1, were used as a measure for the complexity of the problem. To measure the performance of the experimental system, *probability of false alarm* (PFA), a concept from Receiver Operating Characteristic (ROC) curves [52], was used as a metric.

In this thesis, PFA is defined as the probability of the experimental system declaring a problem instance as a problem with phase transition behavior, while the benchmark program has found a solution or has been able to search the entire search space (and found no solutions). Equation 4.4 shows the calculation of PFA, where S is the number of problem instances with at least one solution, N is the number of problem instances with no solutions, and $PT|(S,N)$ is the number of problem instances, which the experimental system declared as problems with phase transition behavior given that they are known to have a solution or the space was searchable by the benchmark problem.

$$PFA = \frac{PT|(S,N)}{S+N} \tag{4.4}$$

To declare that a problem instance shows phase transition behavior, the experimental system needed to reach a termination point, which has been set prior to the execution, before completing the search in the entire search space and couldn't find a solution up to the termination point.

The first set of experiments demonstrated the effects of changing control parameters k_P and k_D, which were used to calculate the control variable κ, while keeping the goal value fixed. The control variable κ was used to calculate the test value for the decision variable that the search space was distributed on. Initial value for the control variable κ_0 was set to be 0.0, which means the minimum value within the current domain of the decision variable. The control variable κ was re-calculated every 500 distribution steps. Table 4.1 shows the results for $goal = 1.0$, $\kappa_0 = 0.0$, and $\tau = 500$. The experimental system was run on each problem instance multiple times with termination point varying between 250 and 100000. The results show that control parameters have minimum effect for low termination points, since the controller does not have enough cycles to change the control variable to make significant change in the direction of the search. Low termination point means low patience for the experimental system and results in inaccurate declaration of phase transition behavior. For high termination points PFA values are lower, since it means that the experimental system has traversed a high percentage of the search space by then. The use of control or the change of the direction of search makes a difference, an improvement over non-controlled execution, $k_P = 0.0$ and $k_D = 0.0$. (See Figure 4.2 for a comparison of the use of different control parameters). Another observation is that the PFA values for different values of control parameters converge and do not change much above a certain value of termination point.

The second set of experiments aimed at discovering the effects of different goal values on the search results. Table 4.2 shows the results obtained by using control parameters values of $k_P = 0.35$ and $k_D = 0.175$, and an initial value of $\kappa_0 = 0.0$ for the control variable κ. Although different goal

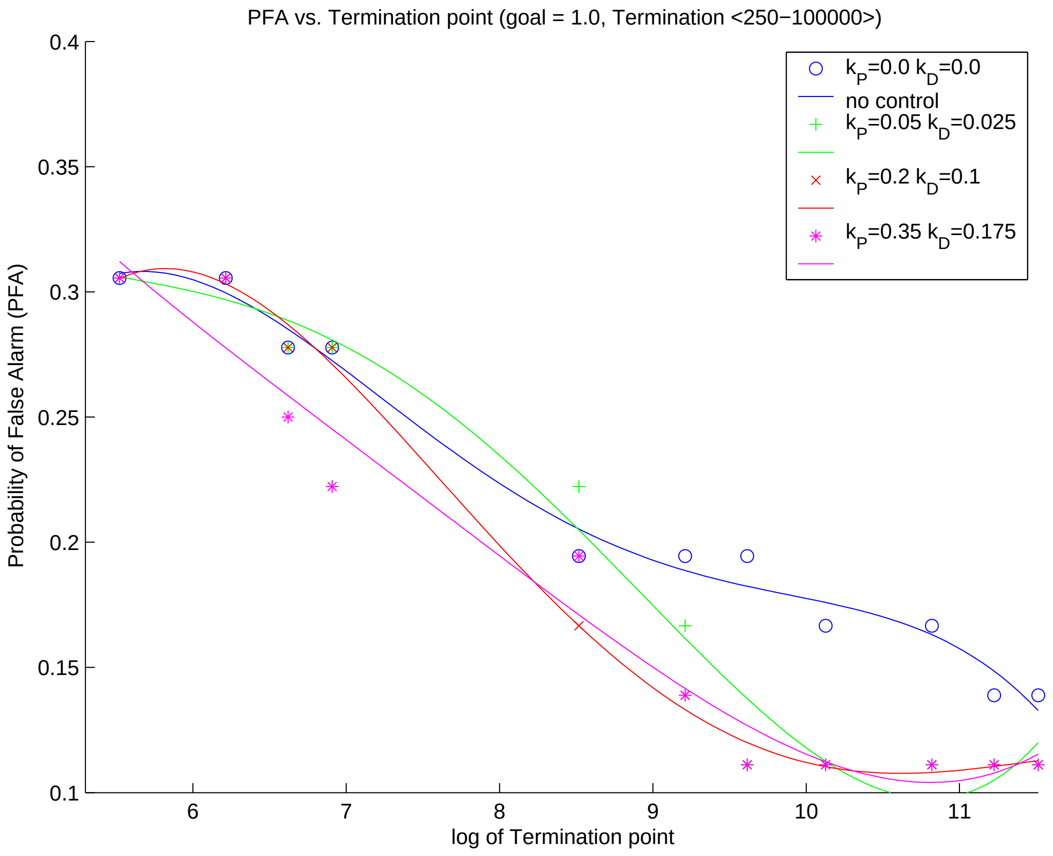


Figure 4.2: Job scheduling experiment: PFA as a function of termination point and varying control parameters

Table 4.1: Probability of false alarm (PFA) as a function of termination point and varying control parameters: $goal = 1.0, \kappa_0 = 0.0, \tau = 500$

Termination Point	$k_D = 0.175$ $k_P = 0.35$	$k_D = 0.1$ $k_P = 0.2$	$k_D = 0.025$ $k_P = 0.05$	$k_D = 0.0$ $k_P = 0.0$
250	0.31	0.31	0.31	0.31
500	0.31	0.31	0.31	0.31
750	0.25	0.29	0.29	0.29
1000	0.22	0.29	0.29	0.29
5000	0.19	0.17	0.22	0.19
10000	0.14	0.14	0.17	0.19
15000	0.11	0.11	0.11	0.19
25000	0.11	0.11	0.11	0.17
50000	0.11	0.11	0.11	0.17
75000	0.11	0.11	0.11	0.14
100000	0.11	0.11	0.11	0.14

values, all being between 1.0 and 0.0, resulted in different PFA values for low termination points, the results converged for higher termination points. Since the feedback, calculated by $(\Delta S - \Delta N)/(\Delta S + \Delta N)$, was always -0.1 for hard problem instances, the controller was updating the value of the control variable κ until it reached 1.0 (the maximum).

The third set of experiments listed in Table 4.3 were targeted towards the selection of the initial value of the control variable κ. Figure 4.4 shows that the selection of the initial value κ_0 for the control variable is more significant on the results than the selection of the control parameter values.

Overall the experiments showed that the use of control improved the results in terms of PFA. The selection of an initial value for the control variable was very important and after certain values of termination point, which are related to the size of the problem in hand, the change of the value of PFA was insignificant. Potential improvements for the experimental system can be; 1) to select the initial value of the control variable based

Table 4.2: Probability of false alarm (PFA) as a function of termination point and varying goal values: $k_P = 0.35, k_D = 0.175, \kappa_0 = 0.0, \tau = 500$

Termination Point	$goal = 1.0$	$goal = 0.5$	$goal = 0.25$
250	0.31	0.33	0.31
500	0.31	0.33	0.31
750	0.25	0.29	0.29
1000	0.22	0.29	0.29
5000	0.19	0.17	0.22
10000	0.14	0.14	0.17
15000	0.11	0.11	0.11
25000	0.11	0.11	0.11
50000	0.11	0.11	0.11
75000	0.11	0.11	0.11
100000	0.11	0.11	0.11

Table 4.3: Probability of false alarm (PFA) as a function of termination point and initial value of the control variable; κ_0, $k_P = 0.35, k_D = 0.175, goal = 1.0, \tau = 500$

Termination Point	$\kappa_0 = 0.0$	$\kappa_0 = 0.5$
250	0.31	0.60
500	0.31	0.58
750	0.25	0.49
1000	0.22	0.44
5000	0.19	0.36
10000	0.14	0.33
15000	0.11	0.33
25000	0.11	0.31
50000	0.11	0.31
75000	0.11	0.29
100000	0.11	0.29

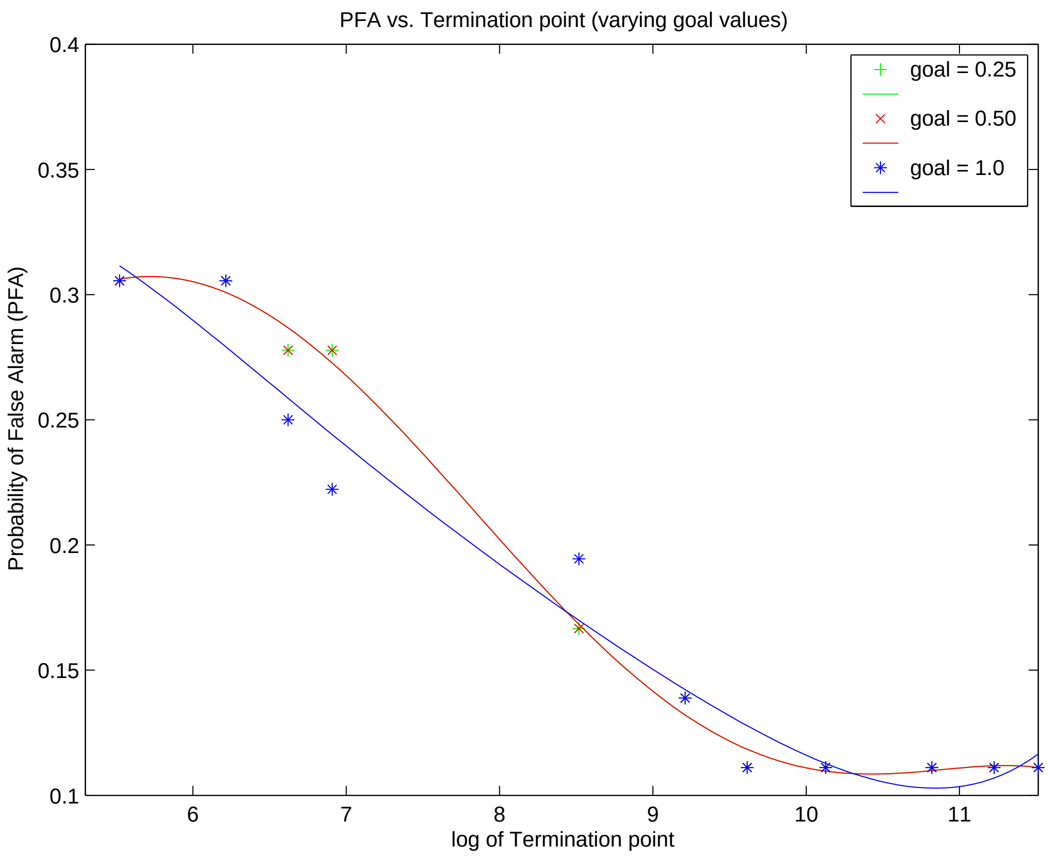


Figure 4.3: Effects of changing goal values on PFA

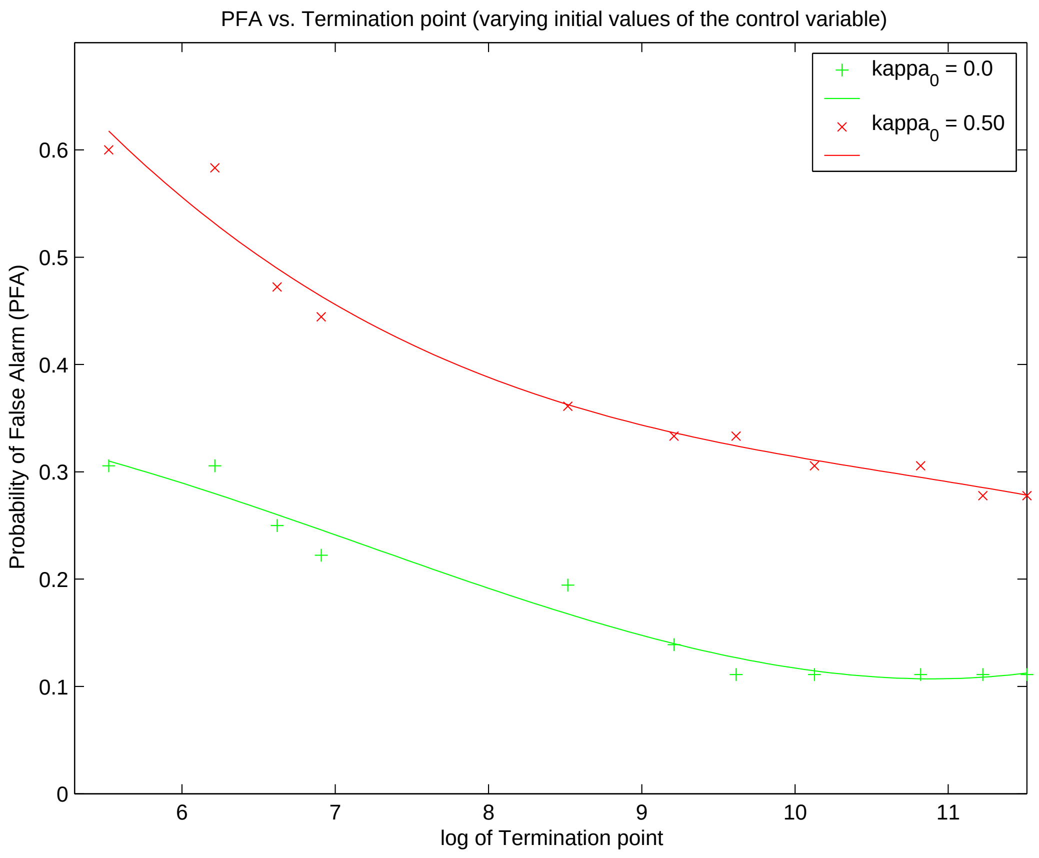


Figure 4.4: Effects of changing the initial value κ_0 of the control variable on the values on PFA

on the problem formulation, 2) to determine the value of the termination based on the problem size, and 3) to update the search engine so that the change in the search direction is not only valid for the sub-trees of the current branch of the search tree.

4.2.7 Complete Results

The following tables show the complete listing of the results obtained from the job scheduling experiments. In a results table each row corresponds to a different problem instance and each column corresponds to a different termination point. Problem instances are indexed 0 through 49. Each cell indicates the result of one execution of the problem. There can be 3 results; *S*, which means that there is at least one solution, *N*, which means that the entire search tree has been traversed and there are no solutions, and *PT*, which means that the search engine has reached the termination point before completing the traversal of the search tree and it couldn't find a solution up to the termination point.

Data Set	Expected Result	Term 100000	Term 75000	Term 50000	Term 25000	Term 15000	Term 10000	Term 5000	Term 1000	Term 750	Term 500	Term 250
0	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
1	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
2	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
3	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
4	S	S	S	S	S	S	S	S	S	S	S	S
5	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
6	S	S	S	S	S	S	S	S	S	S	PT	PT
7	S	S	S	S	S	S	S	S	S	S	S	S
8	S	S	S	S	S	S	S	S	PT	PT	PT	PT
9	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
10	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
11	S	S	S	S	S	S	S	S	S	S	S	S
12	S	S	S	S	S	S	S	S	S	S	S	S
13	N	N	N	N	N	N	N	PT	PT	PT	PT	PT
14	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
15	S	S	S	PT	PT	PT	PT	PT	PT	PT	PT	PT
16	N	N	N	N	N	N	N	N	N	N	N	N
17	N	N	N	N	N	N	N	N	PT	PT	PT	PT
18	S	S	S	S	S	S	S	S	S	S	S	S
19	S	S	S	S	S	S	S	S	S	S	S	S
20	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
21	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
22	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
23	S	S	S	S	S	S	S	S	S	S	S	S
24	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
25	S	S	S	S	S	PT	PT	PT	PT	PT	PT	PT
26	S	S	S	S	S	S	S	S	S	S	S	S
27	S	S	S	S	S	S	S	S	S	S	S	S
28	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
29	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
30	S	S	S	S	S	S	S	S	S	S	PT	PT
31	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
32	S	S	S	S	S	S	S	S	S	PT	PT	PT
33	S	S	S	PT	PT	PT	PT	PT	PT	PT	PT	PT
34	S	S	S	S	S	S	S	S	S	S	S	S
35	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
36	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
37	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
38	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
39	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
40	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
41	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
42	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
43	S	S	S	S	S	S	S	S	S	S	S	S
44	S	S	S	S	S	S	S	S	S	S	S	S
45	S	S	S	S	S	S	S	S	S	S	S	S
46	S	S	S	S	S	S	S	S	S	S	S	S
47	N	N	N	N	N	N	N	N	N	N	PT	PT
48	N	N	N	N	N	N	N	N	N	N	PT	PT
49	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT

Figure 4.5: Test results for the Job Scheduling experiment data set: $k_D = 0.3$, $k_P = 0.5$, $\kappa_0 = 0.5$, $goal = 1.0$

Data Set	Expected Result	Term 100000	Term 75000	Term 50000	Term 25000	Term 15000	Term 10000	Term 5000	Term 1000	Term 750	Term 500	Term 250
0	S	S	S	S	S	S	S	S	PT	PT	PT	PT
1	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
2	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
3	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
4	S	S	S	S	S	S	S	S	S	S	S	S
5	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
6	S	S	S	S	S	S	S	S	S	S	S	S
7	S	S	S	S	S	S	S	S	S	S	S	S
8	S	S	S	S	S	S	S	S	PT	PT	PT	PT
9	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
10	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
11	S	S	S	S	S	S	S	S	S	S	S	S
12	S	S	S	S	S	S	S	S	S	S	S	S
13	N	N	N	N	N	N	N	PT	PT	PT	PT	PT
14	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
15	S	S	S	S	S	S	S	S	S	S	S	S
16	N	N	N	N	N	N	N	N	N	N	N	N
17	N	N	N	N	N	N	N	N	PT	PT	PT	PT
18	S	S	S	S	S	S	S	S	S	S	PT	PT
19	S	S	S	S	S	S	S	S	S	S	S	S
20	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
21	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
22	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
23	S	S	S	S	S	S	S	S	S	S	S	S
24	S	S	S	S	S	S	S	S	S	S	S	S
25	S	S	S	S	S	S	S	S	S	S	PT	PT
26	S	S	S	S	S	S	S	S	S	S	S	S
27	S	S	S	S	S	S	S	S	S	S	S	S
28	S	S	S	S	S	S	PT	PT	PT	PT	PT	PT
29	S	S	S	S	S	S	S	S	PT	PT	PT	PT
30	S	S	S	S	S	S	S	S	S	S	S	S
31	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
32	S	S	S	S	S	S	S	S	S	S	S	S
33	S	S	S	S	S	S	S	S	S	S	S	S
34	S	S	S	S	S	S	S	S	S	S	S	S
35	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
36	S	S	S	S	S	S	S	S	S	S	S	S
37	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
38	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
39	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
40	S	S	S	S	S	S	S	S	S	S	S	S
41	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
42	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
43	S	S	S	S	S	S	S	S	S	S	S	S
44	S	S	S	S	S	S	S	S	S	S	S	S
45	S	S	S	S	S	S	S	S	S	S	S	S
46	S	S	S	S	S	S	S	S	S	S	S	S
47	N	N	N	N	N	N	N	N	N	N	N	N
48	N	N	N	N	N	N	N	N	N	N	N	N
49	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT

Figure 4.6: Test results for the Job Scheduling experiment data set: $k_D = 0.1$, $k_P = 0.2$, $\kappa_0 = 0.0$, $goal = 1.0$

Data Set	Expected Result	Term 100000	Term 75000	Term 50000	Term 25000	Term 15000	Term 10000	Term 5000	Term 1000	Term 750	Term 500	Term 250
0	S	S	S	S	S	S	S	PT	PT	PT	PT	PT
1	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
2	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
3	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
4	S	S	S	S	S	S	S	S	S	S	S	S
5	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
6	S	S	S	S	S	S	S	S	S	S	S	S
7	S	S	S	S	S	S	S	S	S	S	S	S
8	S	S	S	S	S	S	S	PT	PT	PT	PT	PT
9	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
10	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
11	S	S	S	S	S	S	S	S	S	S	S	S
12	S	S	S	S	S	S	S	S	S	S	S	S
13	N	N	N	N	N	N	PT	PT	PT	PT	PT	PT
14	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
15	S	S	S	S	S	S	S	S	S	S	S	S
16	N	N	N	N	N	N	N	N	N	N	N	N
17	N	N	N	N	N	N	N	N	N	N	PT	PT
18	S	S	S	S	S	S	S	S	PT	PT	PT	PT
19	S	S	S	S	S	S	S	S	S	S	S	S
20	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
21	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
22	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
23	S	S	S	S	S	S	S	S	S	S	S	S
24	S	S	S	S	S	S	S	S	S	S	S	S
25	S	S	S	S	S	S	S	S	S	S	S	S
26	S	S	S	S	S	S	S	S	S	S	S	S
27	S	S	S	S	S	S	S	S	S	S	S	S
28	S	S	S	S	S	S	PT	PT	PT	PT	PT	PT
29	S	S	S	S	S	S	S	S	PT	PT	PT	PT
30	S	S	S	S	S	S	S	S	S	S	S	S
31	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
32	S	S	S	S	S	S	S	S	S	S	S	S
33	S	S	S	S	S	S	S	S	S	S	S	S
34	S	S	S	S	S	S	S	S	S	S	S	S
35	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
36	S	S	S	S	S	S	S	S	S	S	S	S
37	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
38	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
39	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
40	S	S	S	S	S	S	S	S	S	S	S	S
41	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
42	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
43	S	S	S	S	S	S	S	S	S	S	S	S
44	S	S	S	S	S	S	S	S	S	S	S	S
45	S	S	S	S	S	S	S	S	S	S	S	S
46	S	S	S	S	S	S	S	S	S	S	S	S
47	N	N	N	N	N	N	N	N	N	N	N	N
48	N	N	N	N	N	N	N	N	N	N	N	N
49	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT

Figure 4.7: Test results for the Job Scheduling experiment data set: $k_D = 0.025$, $k_P = 0.05$, $\kappa_0 = 0.0$, $goal = 1.0$

Data Set	Expected Result	Term 100000	Term 75000	Term 50000	Term 25000	Term 15000	Term 10000	Term 5000	Term 1000	Term 750	Term 500	Term 250
0	S	S	S	S	S	S	S	PT	PT	PT	PT	PT
1	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
2	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
3	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
4	S	S	S	S	S	S	S	S	S	S	S	S
5	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
6	S	S	S	S	S	S	S	S	S	S	S	S
7	S	S	S	S	S	S	S	S	S	S	S	S
8	S	S	S	S	S	S	S	S	PT	PT	PT	PT
9	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
10	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
11	S	S	S	S	S	S	S	S	S	S	S	S
12	S	S	S	S	S	S	S	S	S	S	S	S
13	N	N	N	N	N	N	N	PT	PT	PT	PT	PT
14	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
15	S	S	S	S	S	S	S	S	S	S	S	S
16	N	N	N	N	N	N	N	N	N	N	N	N
17	N	N	N	N	N	N	N	N	N	N	PT	PT
18	S	S	S	S	S	S	S	S	S	S	PT	PT
19	S	S	S	S	S	S	S	S	S	S	S	S
20	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
21	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
22	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
23	S	S	S	S	S	S	S	S	S	S	S	S
24	S	S	S	S	S	S	S	S	S	S	S	S
25	S	S	S	S	S	S	S	S	S	S	S	S
26	S	S	S	S	S	S	S	S	S	S	S	S
27	S	S	S	S	S	S	S	S	S	S	S	S
28	S	S	S	S	S	S	PT	PT	PT	PT	PT	PT
29	S	S	S	S	S	S	S	S	S	PT	PT	PT
30	S	S	S	S	S	S	S	S	S	S	S	S
31	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
32	S	S	S	S	S	S	S	S	S	S	S	S
33	S	S	S	S	S	S	S	S	S	S	S	S
34	S	S	S	S	S	S	S	S	S	S	S	S
35	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
36	S	S	S	S	S	S	S	S	S	S	S	S
37	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
38	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
39	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
40	S	S	S	S	S	S	S	S	S	S	S	S
41	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
42	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
43	S	S	S	S	S	S	S	S	S	S	S	S
44	S	S	S	S	S	S	S	S	S	S	S	S
45	S	S	S	S	S	S	S	S	S	S	S	S
46	S	S	S	S	S	S	S	S	S	S	S	S
47	N	N	N	N	N	N	N	N	N	N	N	N
48	N	N	N	N	N	N	N	N	N	N	N	N
49	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT

Figure 4.8: Test results for the Job Scheduling experiment data set: $k_D = 0.175$, $k_P = 0.35$, $\kappa_0 = 0.0$, $goal = 1.0$

4.3 Fixture Design Experiment

4.3.1 Problem Statement

In this section we describe the second experimental multi-objective optimization scenario used in this thesis. It is a design problem of finding connections (channel layout) between one or more integrated circuit board(s) (or Unit Under Test - UUT) and a functional board tester (FBT). More specifically, the goal is to find a mapping between the edge connector pins of the UUT and the digital channels of the FBT while keeping the cost of the system low by using a small number of assets (channels).

The inputs to the system are the edge connector pin requirements (we will call them "pin requirements") for one or more Unit Under Tests (UUT) and the configuration of the tester (see Figure 4.9 for the components of the tester). Some of the pin requirements are the digital timing, the voltage levels that are used, and the analog capabilities that are required. The configuration of the tester contains the data on the number and the types of the channel cards and their positions within the chassis. The outputs of the system are the mappings between the UUT pins and the channels of the channel cards (for example, UUT pin 1 will be connected to the channel 2 of third channel card).

4.3.2 Problem Formulation

To formalize the objective functions and the constraints we need to introduce the following notation:

- P - the set of all UUT pins, $p \in P$

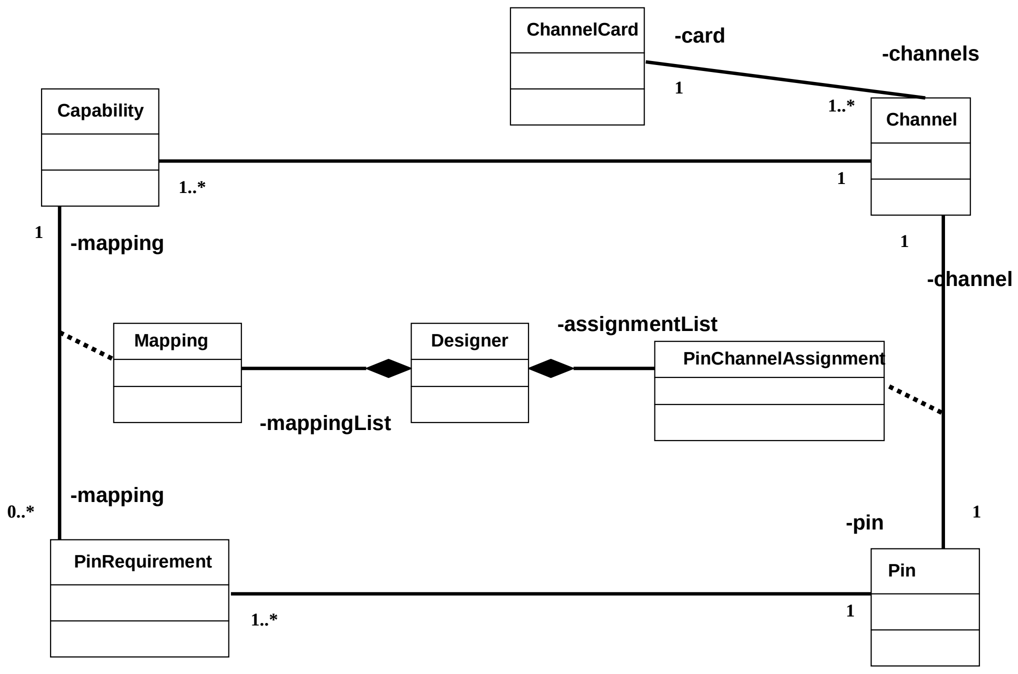


Figure 4.9: Class Diagram for the Fixture for a Functional Board Test System

- N_P - the cardinality of the set P. It is constant for this optimization problem.
- C - the set of all channels, $c \in C$, in the tester system. It is an optimization variable.
- CC - the set of all channel cards in the system.
- $Cont : 2^C \rightarrow 2^{CC}$ - It maps a set of channels to their container channel cards if the channels are assigned to any pins.
- R - the set of all possible pin requirements, $r \in R$.

- $Rec : P \to 2^R$ - the requirements function. It assigns the subset of requirements for each pin.
- CAP - the set of all possible channel capabilities. A capability can be digital measurement, digital sourcing, analog measurement, or analog sourcing.
- $Cap : C \to 2^{CAP}$ - the capability function that assigns a set of capabilities to each channel c.
- $PC : P \to C \cup \{c_{null}\}$ - channel assignment function. It assigns a channel to a UUT pin. This is an optimization variable.
- $f_{map} : R \to CAP$ - a function that maps the pin requirements to channel capabilities. The function is $m - to - 1$.
- $card : Set \to N$ - cardinality function that returns the size of a given set.
- $N_{CC_{MAX}}$ - the maximum number of channel cards that can be inserted in a FBT.
- c_{null} - null assignment for a UUT pin. This indicates that a channel has not been assigned to the UUT pin.

Objective 1: Minimize the use of high capability channels to decrease the cost.

$$\text{Minimize} \qquad UsedCap = \sum_{p \in P} card(Cap(PC(p))) \tag{4.5}$$

$$\text{s.t.} \quad \forall p, p' \in P : \quad p \neq p' \Rightarrow PC(p) \neq PC(p') \tag{4.6}$$

$$\forall p \in P, \forall r \in Rec(p) : \quad f_{map}(r) \in Cap(PC(p)) \tag{4.7}$$

Objective 2: Minimize the number of total channel cards used in the system.

$$\text{Minimize} \quad N_{CC} \quad = card(Cont(C)) \tag{4.8}$$

$$\text{s.t.} \quad N_{CC} \quad \leq N_{CC_{MAX}} \tag{4.9}$$

$$card(PC(P) - c_{null}) \quad \leq N_P \tag{4.10}$$

Goals and Constraints for Fixture Design Experiment in OCL

The following is a complete listing of OCL representation of the the constraints and the objective functions of the fixture design experiment.

```
package FXT

context Designer def:

    -- C - the set of all channels in the tester system
    let C : Sequence(Channel) = Data::channels
    let N_C : Integer = C->size

    -- CAP - the set of all capabilities supported by the tester
    let CAP : Sequence() = C->iterate(Chan : Channel;
        AccC : Sequence() = Sequence{} |
        AccC->append( Chan.capability ) )

    let P : Sequence(Pin) = Data::pins
    let REQ : Sequence(PinRequirement) = Data::pinRequirements
    let N_P : Integer = REQ->size
    let N_CC : Integer = Data::channelCards

-- Each UUTPin will be wired to a different channel.

context Designer::Constraint1(PC : Sequence(Integer))
    -- PC->forAll(p1,p2 |  p1 <> p2)
    post:
```

```
    ONT_Distinct(PC)

-- All the electrical requirements of a UUTPin need to ve
satisfied by -- the wired channel's capabilities.

context Designer::Constraint2(PC : Sequence(Integer))
    post:
    REQ->iterate(I R |
        CAP->iterate(J Ca |
            if not Ca->includesAll(R)
            then ONT_NotEqual(PC->at(I), J)
            endif))


-- The goal is to use the low-capability, more expensive channels
-- first. The average number of capabilities that each channel has
-- is an indicator of the type of channels that are used.

context Designer::objective1Value(PCAssign : Sequence(Integer)) :
Integer
    post:
    let Sum = PCAssign->iterate(PCItem; AccS : Integer = 0 |
        ONT_Inc(AccS , ONT_Size(CAP->at(PCItem))))
    in Sum

-- The goal is to use fewer number of channel cards. This function
-- goes over the channels, finds the channel cards that the used
-- channels belong to, and returns the count of channel cards.

context Designer::GetChanCardID(Ch : Integer) : Integer
    post:
    let ChItem = C->at(Ch)
    in ChItem.cc

context Designer::MarkUsedChanCards(PCAssign : Sequence(Integer))
              : Sequence(Integer)
    def:
    let UsedCC = ONT_NewArray(N_CC)
    pre: PCAssign->iterate(Ch |
        ONT_PutArray(UsedCC, GetChanCardID(Ch), 1))
    post: UsedCC

context Designer::objective2Value(PCAssign : Sequence(Integer)) :
                  Integer
    def:
    let UsedCC = MarkUsedChanCards(PCAssign)
    post:
    let Count = UsedCC->iterateArr(CCItem; AccC : Integer = 0 |
        ONT_Inc(AccC , CCItem))
    in Count

--  List of all constraints

context Designer::Constraints (PC : Sequence(Integer)) : ONT_PROC
```

```
    pre: PC = ONT_NewList(N_P, N_C)
    pre: Constraint1(PC)
    pre: Constraint2(PC)
    post: ONT_Dist(PC)

-- First objective function that minimizes the average number of
-- capabilities that the used channels possess. In other words,
--we like to use the less capable, less expensive channels.

context Designer::objective1()
    post: ONT_Minimize(objective1Value)

-- Second objective function that minimizes the number of channel
-- cards used to prevent the cost hike.

context Designer::objective2()
    post: ONT_Minimize(objective2Value)

endpackage
```

4.3.3 Empirical Calculation of Phase Transition Regions

We had a number of candidate phase transition variables for the fixture design experiment. A preliminary experiment with two of the variables, *UUT pin bus size* and *Channel card size* showed a behavior similar to phase transition (see Section 4.3.3 for details).

Candidate variables:

- *Channel card size* (Number of channels on a channel card)
- *UUT pin cluster size.* A UUT pin cluster is a group of UUT pins sharing similar requirements. Since they share similar requirements, the search algorithms will most likely try to match them to the channels on the same channel card.
- *UUT pin bus size* UUT pins which form an address bus or data bus are typically assigned to channels physically close to each other. This physical closeness is important for eliminating timing differences between different pins on a bus.

Candidate invariants:

- UUT pin cluster size / Channel card size
- UUT pin bus size / Channel card size

OZ program that demonstrates phase boundaries for Fixture Design Experiment

In the following OZ program, a set of pins (or UUT pins) are assigned to channel cards. In the input data, there are 16 pins and each channel card

Table 4.4: Empirical results for Phase Transition in the Fixture Design System problem

CC size	Nodes visited	Depth of search tree	Result
1	43679	55	Completed with no solution
2	$\sim 277K$	63	Completed with no solution
3	$\sim 900K$	34	Gave virtual memory error
4	$\sim 900K$	34	Gave virtual memory error
5	2190	17	Completed with a solution
6	62	14	Completed with a solution
7	30	14	Completed with a solution

has three channels each. Two kinds of pin requirements are used; bus pins, where pins in the same bus need to be assigned to the same channel card, and disjoint pins, where two disjoint pins cannot be in the same channel card.

```
local Data fun {FxtGen Data}
   NbPins    = Data.nbPins
   NbChannelCardSize = Data.nbChannelCardSize
   Constraints   = Data.constraints
   MinNbChannelCards    = NbPins div NbChannelCardSize
in
   proc {$ Assign}
      NbChannelCards  = {FD.int MinNbChannelCards#NbPins}
   in
      {FD.distribute naive [NbChannelCards]}
      %% Assign: Pin --> ChannelCard
      {FD.tuple assign NbPins 1#NbChannelCards Assign}
      %% at most NbChannelCardSize per ChannelCard
      {For 1 NbChannelCards 1
```

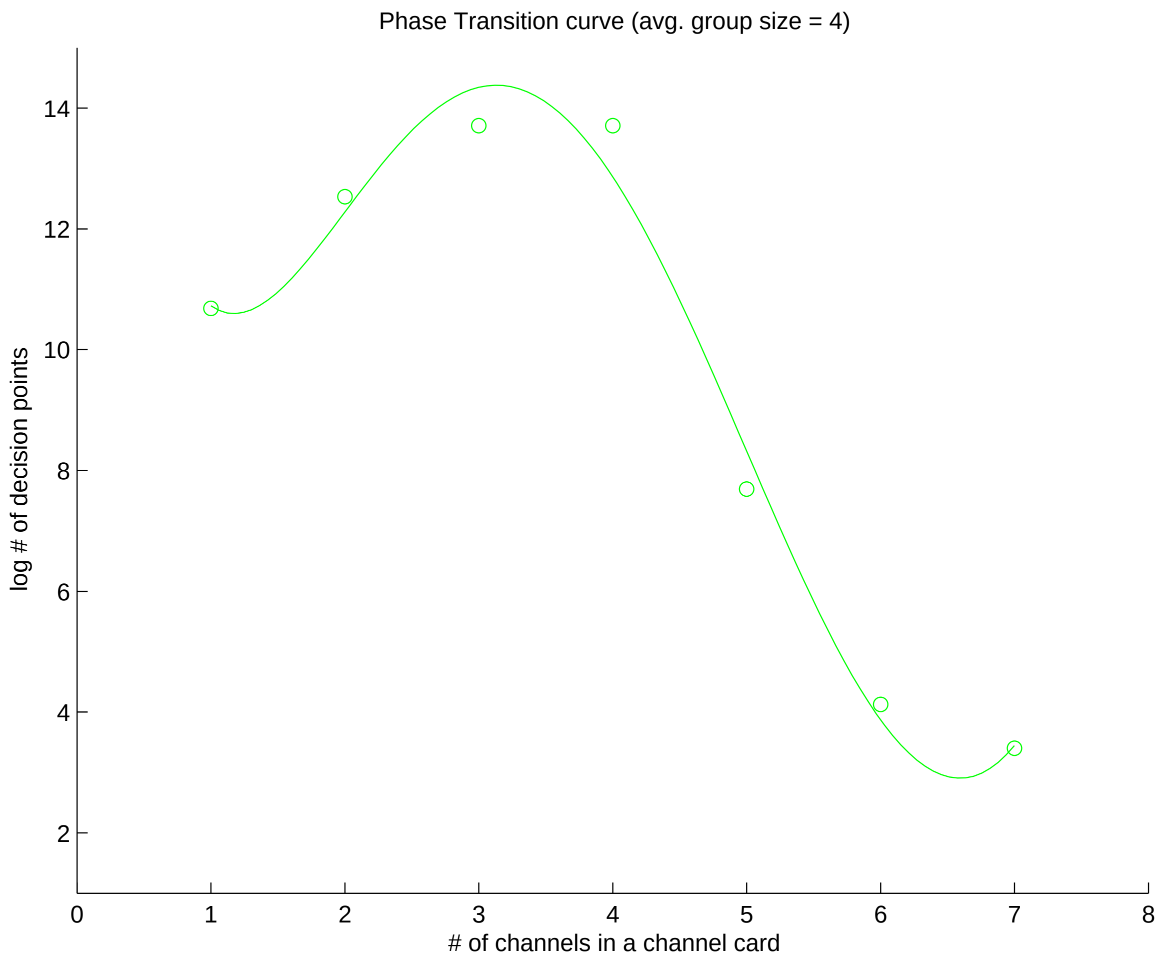


Figure 4.10: Effects of changing the channel card size on the complexity of the problem.

```
       proc {$ ChannelCard}
            {FD.atMost NbChannelCardSize Assign ChannelCard} end}
      %% impose constraints
      {ForAll Constraints
       proc {$ C}
          case C
          of bus(X Ys) then
             {ForAll Ys proc {$ Y} Assign.X =: Assign.Y end}
          [] disjoint(X Ys) then
             {ForAll Ys proc {$ Y} Assign.X \=: Assign.Y end}
          end
       end}
      {FD.distribute ff Assign}
   end
end in
    Data = data(nbPins:16  nbChannelCardSize:3
            constraints: [ bus(4 [8 11])  bus(12 [13 14 15 16])
                           disjoint(1 [2 3 5 7 8 10])
                           disjoint(2 [3 4 7 8 9 11])
               disjoint(3 [5 6 8])
               disjoint(4 [6 10])
               disjoint(6 [7 10])
               disjoint(7 [8 9])
               disjoint(8 [10]) ] )
    {ExploreOne {FxtGen Data}}
end}
```

Using Mozart's OZ environment, the same program was executed for channel card sizes of 1,2,3,4,5,6, and 7 (see Figure 4.10). Table 4.4 summarizes the results.

4.3.4 Experimental Setup

A test data generator was implemented based on the empirical results obtained in Section 4.3.3. The data generator required four test parameters; 1) number of UUT pins, 2) number of channel, 3) maximum number requirement types, and 4) the maximum number of channels per channel card. In the experiments, the number of pins were 13, the number of channels were 16, the number of requirements types were 4, and the maximum number of channels per channel card was 4. A total of 40 problem instances were generated.

4.3.5 Analysis of the Results

The primary goal of the fixture design experiments was to confirm the results and ideas obtained in the job scheduling experiments. This time the emphasis was on the effects of control on the search results.

Table 4.5 shows the results for the use of different control parameters. The value of goal was fixed at 0.5 and the initial value of the control variable κ_0 was 0.0. Similar to job scheduling experiments, the use of control resulted in slightly better PFA values than the runs without control. Due to the smaller size of the search space of the fixture design problem, the values of the termination points to reach steady PFA values were lower that the termination points observed in the job scheduling experiments.

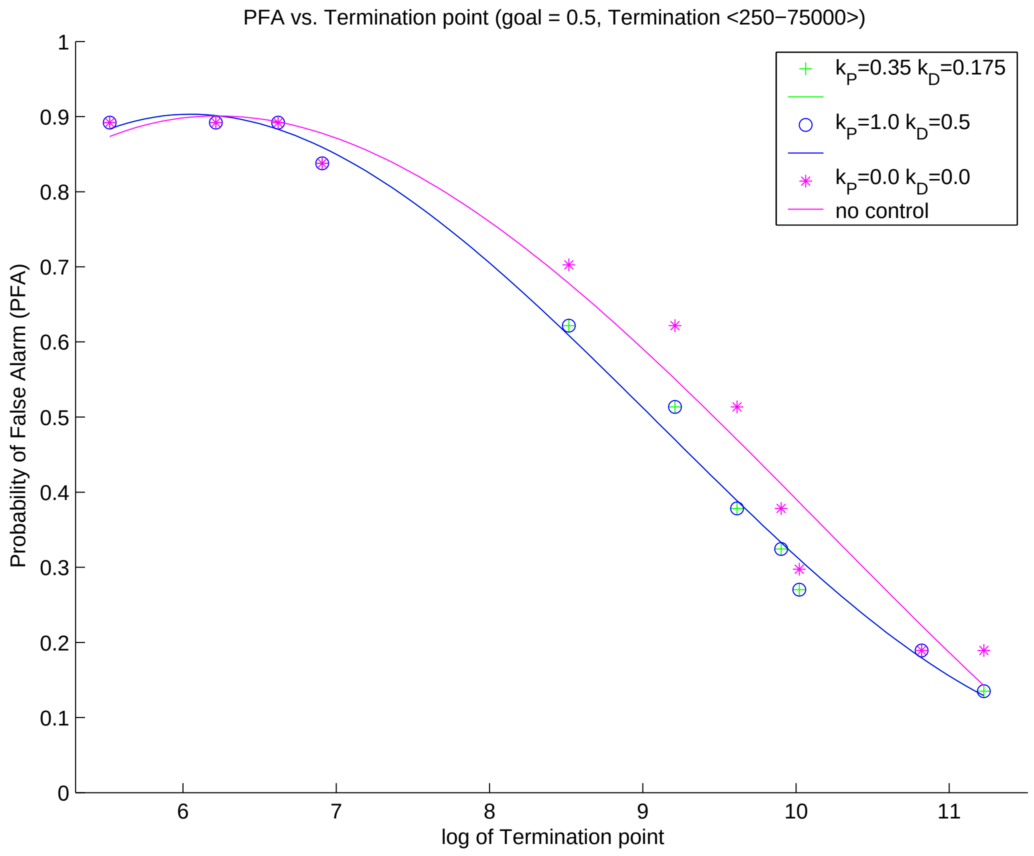


Figure 4.11: Fixture design experiment: PFA as a function of termination point and varying control parameters

Table 4.5: Probability of false alarm (PFA) as a function of termination point and varying control parameters: $goal = 0.5, \kappa_0 = 0.0, \tau = 500$

Termination Point	$k_D = 0.175$ $k_P = 0.35$	$k_D = 0.5$ $k_P = 1.0$	$k_D = 0.0$ $k_P = 0.0$
250	0.89	0.89	0.89
500	0.89	0.89	0.89
750	0.89	0.89	0.89
1000	0.84	0.84	0.84
5000	0.62	0.62	0.70
10000	0.51	0.51	0.62
15000	0.38	0.38	0.51
20000	0.32	0.32	0.38
25000	0.27	0.27	0.30
50000	0.19	0.19	0.19
75000	0.14	0.14	0.14

4.3.6 Complete Results

The following tables show the complete listing of the results obtained from the fixture design experiments. In the table each row corresponds to a different problem instance and each column corresponds to a different termination point. Problem instances are indexed 0 through 39. Each cell indicates the result of one execution of the problem. There can be 3 results; *S*, which means that there is at least one solution, *N*, which means that the entire search tree has been traversed and there are no solutions, and PT, which means that the search engine has reached the termination point before completing the traversal of the search tree and it couldn't find a solution up to the termination point.

Data Set	Expected Result	Term 75000	Term 50000	Term 25000	Term 20000	Term 15000	Term 10000	Term 5000	Term 1000	Term 750	Term 500	Term 250
0	S	S	S	S	S	S	S	S	S	S	S	S
1	S	S	S	S	S	S	S	S	S	S	S	S
2	S	S	S	S	S	S	PT	PT	PT	PT	PT	PT
3	S	S	S	S	S	S	PT	PT	PT	PT	PT	PT
4	S	S	S	S	S	S	PT	PT	PT	PT	PT	PT
5	N	N	N	N	N	N	N	N	N	N	N	N
6	S	S	S	S	S	S	S	PT	PT	PT	PT	PT
7	S	S	S	S	S	S	S	S	PT	PT	PT	PT
8	S	S	S	S	S	S	S	S	PT	PT	PT	PT
9	S	S	S	S	S	S	S	PT	PT	PT	PT	PT
10	S	S	S	S	S	S	S	S	PT	PT	PT	PT
11	N	N	N	N	N	N	N	N	N	PT	PT	PT
12	N	N	N	N	N	N	N	N	PT	PT	PT	PT
13	N	N	N	N	N	N	PT	PT	PT	PT	PT	PT
14	S	S	S	S	S	S	S	PT	PT	PT	PT	PT
15	N	N	N	N	PT	PT	PT	PT	PT	PT	PT	PT
16	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
17	N	N	N	N	N	N	N	N	N	N	N	N
18	N	N	N	PT	PT	PT	PT	PT	PT	PT	PT	PT
19	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
20	N	N	N	N	N	PT	PT	PT	PT	PT	PT	PT
21	N	N	N	N	N	N	N	N	PT	PT	PT	PT
22	N	N	N	N	PT	PT	PT	PT	PT	PT	PT	PT
23	N	N	N	N	N	N	N	PT	PT	PT	PT	PT
24	N	N	N	N	N	N	N	N	PT	PT	PT	PT
25	N	N	N	N	N	N	N	N	N	PT	PT	PT
26	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
27	N	N	N	N	N	N	N	N	PT	PT	PT	PT
28	N	N	N	N	N	PT	PT	PT	PT	PT	PT	PT
29	N	N	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
30	N	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
31	N	N	N	N	N	N	PT	PT	PT	PT	PT	PT
32	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
33	S	S	S	S	S	S	S	S	PT	PT	PT	PT
34	N	N	N	PT	PT	PT	PT	PT	PT	PT	PT	PT
35	N	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
36	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
37	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
38	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
39	N	N	N	PT	PT	PT	PT	PT	PT	PT	PT	PT

Figure 4.12: Test results for the Fixture Design experiment data set: $k_D = 0.5$, $k_P = 1.0$, $\kappa_0 = 0.0$, $goal = 0.5$

Data Set	Expected Result	Term 75000	Term 50000	Term 25000	Term 20000	Term 15000	Term 10000	Term 5000	Term 1000	Term 750	Term 500	Term 250
0	S	S	S	S	S	S	S	S	S	S	S	S
1	S	S	S	S	S	S	S	S	S	S	S	S
2	S	S	S	S	S	S	PT	PT	PT	PT	PT	PT
3	S	S	S	S	S	S	PT	PT	PT	PT	PT	PT
4	S	S	S	S	S	S	PT	PT	PT	PT	PT	PT
5	N	N	N	N	N	N	N	N	N	N	N	N
6	S	S	S	S	S	S	S	PT	PT	PT	PT	PT
7	S	S	S	S	S	S	S	S	PT	PT	PT	PT
8	S	S	S	S	S	S	S	S	PT	PT	PT	PT
9	S	S	S	S	S	S	S	PT	PT	PT	PT	PT
10	S	S	S	S	S	S	S	S	PT	PT	PT	PT
11	N	N	N	N	N	N	N	N	N	PT	PT	PT
12	N	N	N	N	N	N	N	N	PT	PT	PT	PT
13	N	N	N	N	N	N	PT	PT	PT	PT	PT	PT
14	S	S	S	S	S	S	S	PT	PT	PT	PT	PT
15	N	N	N	N	PT	PT	PT	PT	PT	PT	PT	PT
16	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
17	N	N	N	N	N	N	N	N	N	N	N	N
18	N	N	N	PT	PT	PT	PT	PT	PT	PT	PT	PT
19	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
20	N	N	N	N	N	PT	PT	PT	PT	PT	PT	PT
21	N	N	N	N	N	N	N	N	PT	PT	PT	PT
22	N	N	N	N	PT	PT	PT	PT	PT	PT	PT	PT
23	N	N	N	N	N	N	N	PT	PT	PT	PT	PT
24	N	N	N	N	N	N	N	N	PT	PT	PT	PT
25	N	N	N	N	N	N	N	N	N	PT	PT	PT
26	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
27	N	N	N	N	N	N	N	N	PT	PT	PT	PT
28	N	N	N	N	N	PT	PT	PT	PT	PT	PT	PT
29	N	N	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
30	N	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
31	N	N	N	N	N	N	PT	PT	PT	PT	PT	PT
32	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
33	S	S	S	S	S	S	S	S	PT	PT	PT	PT
34	N	N	N	PT	PT	PT	PT	PT	PT	PT	PT	PT
35	N	N	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
36	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
37	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
38	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
39	N	N	N	PT	PT	PT	PT	PT	PT	PT	PT	PT

Figure 4.13: Test results for the Fixture Design experiment data set: $k_D = 0.175$, $k_P = 0.35$, $\kappa_0 = 0.0$, $goal = 0.5$

Data Set	Expected Result	Term 75000	Term 50000	Term 25000	Term 20000	Term 15000	Term 10000	Term 5000	Term 1000	Term 750	Term 500	Term 250
0	S	S	S	S	S	S	S	S	S	S	S	S
1	S	S	S	S	S	S	S	S	S	S	S	S
2	S	S	S	S	S	S	PT	PT	PT	PT	PT	PT
3	S	S	S	S	S	S	PT	PT	PT	PT	PT	PT
4	S	S	S	S	S	S	PT	PT	PT	PT	PT	PT
5	N	N	N	N	N	N	N	N	N	N	N	N
6	S	S	S	S	S	S	S	PT	PT	PT	PT	PT
7	S	S	S	S	S	S	S	S	PT	PT	PT	PT
8	S	S	S	S	S	S	S	S	PT	PT	PT	PT
9	S	S	S	S	S	S	S	PT	PT	PT	PT	PT
10	S	S	S	S	S	S	S	S	PT	PT	PT	PT
11	N	N	N	N	N	N	N	N	N	PT	PT	PT
12	N	N	N	N	N	N	N	N	PT	PT	PT	PT
13	N	N	N	N	N	N	PT	PT	PT	PT	PT	PT
14	S	S	S	S	S	S	S	PT	PT	PT	PT	PT
15	N	N	N	N	PT	PT	PT	PT	PT	PT	PT	PT
16	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
17	N	N	N	N	N	N	N	N	N	N	N	N
18	N	N	N	PT	PT	PT	PT	PT	PT	PT	PT	PT
19	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
20	N	N	N	N	N	PT	PT	PT	PT	PT	PT	PT
21	N	N	N	N	N	N	N	N	PT	PT	PT	PT
22	N	N	N	N	PT	PT	PT	PT	PT	PT	PT	PT
23	N	N	N	N	N	N	N	PT	PT	PT	PT	PT
24	N	N	N	N	N	N	N	N	PT	PT	PT	PT
25	N	N	N	N	N	N	N	N	N	PT	PT	PT
26	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
27	N	N	N	N	N	N	N	N	PT	PT	PT	PT
28	N	N	N	N	N	PT	PT	PT	PT	PT	PT	PT
29	N	N	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
30	N	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
31	N	N	N	N	N	N	PT	PT	PT	PT	PT	PT
32	S	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
33	S	S	S	S	S	S	S	S	PT	PT	PT	PT
34	N	N	N	PT	PT	PT	PT	PT	PT	PT	PT	PT
35	N	N	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
36	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
37	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
38	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT	PT
39	N	N	N	PT	PT	PT	PT	PT	PT	PT	PT	PT

Figure 4.14: Test results for the Fixture Design experiment data set: no control $\kappa_0 = 0.0$, $goal = 0.5$

Chapter 5

Conclusions

In this chapter, first contributions of this thesis are summarized. This is followed by a summary of conclusions of research presented in this thesis. The chapter ends with a list of suggested future research topics related to the problem addressed here.

5.1 Contributions

Automatic synthesis of satisficing programs to solve multiobjective combinatorial optimization problems

A generic constraint satisfaction problem (CSP) code generator was designed to automatically convert specifications for multiobjective combinatorial optimization problems to a number of constraint satisfaction problems expressed in a target CSP programming language. In addition to the CSP code that specifies the new constraint satisfaction problems, the code generator creates code that will find a satisficing solution to these problems through negotiation and detect phase transition behavior if the particular problem is an hard instance. Upon detection of a phase transition behavior, unlike other CSP programs, the generated satisficing program gracefully terminates the search and returns the best solution obtained up to that point. The capabilities of the code generator was demonstrated by using OZ as the target CSP programming language.

The phase transition phenomenon was investigated to identify phase transition invariants and the related computation intensive critical regions for two experimental scenarios. Identified phase transition invariants were used to generate hard problem instances to test the robustness of the automatically-generated satisficing programs.

Specification for an agent-based Self-Controlling Software Architecture (SCSA) as a way of controlling complexity of a constraint satisfaction program

When automatic translation of problem specifications into CSP code replaces the human programmer, the automatically generated code may never terminate due to the complexity of the CSP search. Therefore, the replacement of manual CSP coding by a UML/OCL interface requires a solution to the handling of the complexity of the search for a satisficing solution. SCSA specification provides a control mechanism that is able to monitor the performance of the CSP search engine and change the direction of search towards the regions of search space densely populated with feasible solutions.

Objective-based agentification of multiobjective combinatorial optimization problems

For every objective in the original multiobjective combinatorial optimization problem, a different SCSA agent is utilized. This unbiased agentification of the objectives are especially important when the relative weights or the priorities of the objectives are unknown. Rather than the use of an artificial ordering of the objective functions, a mediator-based negotiation mechanism is used to find a solution that will satisfy all the objective functions. After finding the non-dominated solutions among a set of feasible solutions returned by the agents, the negotiation mechanism utilizes a negotiation approach to select the best solution among the non-dominated ones.

Use of UML, OCL and Ontology for specifying multiobjective optimization problems

Programming in a CSP language was replaced with a more user-friendly and graphical Unified Modeling Language (UML). Since the graphical parts of UML are not sufficient to express many constraints, Object Constraint Language (OCL) can be used to extend the capabilities of UML. In addition, an OCL-based ontology that supports specifications of objective functions and constraints of multiobjective optimization problems were defined. It contains a selection of constructs typically encountered in the specification of multiobjective optimization problems. In addition, the ontology includes constructs to support the generation of software agents.

5.2 Conclusion

This thesis has addressed the specific problem of the automatic synthesis of a satisficing program for a multiobjective combinatorial optimization problem with respect to specification such that it can monitor and control its complexity.

The primary focus has been on multiobjective optimization problems where the objectives conflicted and there was no prior information on the relative importance or weights of the objectives. For such problems, typically, either a globally optimum solution does not exist or an algorithm is not able to find it due to the high complexity of the problem.

In the absence of an algorithm that could find global optima, a satisficing technique was used. In this technique, first, the original multiobjective optimization problem was converted to many constraint satisfaction problems (CSP). Next, each CSP was solved individually by a CSP solver that used a branch and bound algorithm. Then, the feasible solutions for each CSP were ordered by the respective objective functions. Finally, a negotiation algorithm was used to select one solution among these ordered solutions that would satisfy all the CSPs.

To support this technique, an agent-based software architecture was specified. For each objective, a different software agent was created. Each agent independently executed the branch and bound algorithm and reported the feasible solutions found to an entity called "reconfigurer." The reconfigurer would mediate the negotiation among agents. Anytime a solution was demanded, the reconfigurer applied an imitative negotiation approach to choose the best solution among the ones that were reported up to that moment. The quality of the solutions improved as the system was allowed more time for the search.

Each agent utilized both a control strategy to control the direction of the search within the search space and a termination strategy to terminate the search gracefully if a phase transition behavior was detected in the CSP.

In order to develop such a satisficing program, UML/OCL was selected as the language in which problem specifications could be expressed. A conversion mechanism for translating multiobjective optimization problems to multiple constraint satisfaction problems was provided. The translation process made use of an ontology of combinatorial optimization problems.

A generic CSP code generator was implemented as a proof of concept for the conversion mechanism. Generic CSP code generator consisted of two parts: 1) an OCL parser to parse the constraints in an OCL file and to save in an intermediate XML format and 2) a configurable rule-based code generator to take the constraints in XML file as an input and to generate CSP code. An XML schema was defined for the intermediate XML format for constraints; the format was called OCL Meta Language (OCLML). Another XML schema was defined for capturing the translation rules from OCLML to any CSP programming language. The separation of the OCL parser from the code generator and the definition of the two schemas enabled the possibility of targeting to any CSP programming language.

For the proof of concept, OZ was selected as the target CSP programming language, and the translation rules from OCLML to OZ were provided. OZ libraries written for the branch and bound algorithm were modified to accept control input and return status information, such as the number of decision points, the number of solutions, the number of non-solutions, and the depth of the decision point within the search tree.

The proof of concept system was tested against two experimental scenarios, a job scheduling problem and a fixture design problem, with each problem having conflicting objectives. For both problems, phase transition invariants were identified empirically and later used to generate hard problems that showed phase transition behavior. The proof of concept system was executed against these hard problems, and the performance of system was measured for different control parameters and initial conditions. The critical control parameters, which the performance of the proof of concept was most sensitive to were identified.

The experiments with the proof of concept system have showed that satisficing programs can be automatically generated from the specifications, that the complexity of the CSP solving algorithms can be controlled, and that the hard problems can be detected with high probability and low probability of false alarm.

Analysis of the test results has showed that the control strategy implemented for the agents can be improved by selecting control constants and initial conditions for the control parameters by analyzing the specifications of the problems; that the automation of selecting a termination point used in detecting the phase transitions in hard problems is possible; and that the termination point can be determined by inspecting the size of the search space according to the problem specifications.

5.2.1 Future Research

Several research issues could be addressed in the future in order to both generalize and specialize the results of the research presented in this thesis and of the developed experimental system.

- A model of the relationship of the control constants and the initial conditions of the control variables to the problem specifications must be investigated. Later, these findings can be used to modify the CSP code generator to automatically calculate the control constants and the initial conditions of the control variables and to generate code to initialize the controller in an agent.

- In this thesis, SCSA used a feedback loop for control. The use of an adaptation loop in an agent in which a model of the plant would be monitored and updated automatically could be investigated. The adaptation loop might contribute to the robustness of the automatically synthesized optimization algorithms.

- In the experimental system developed in this research, the termination point used to detect the hard problems was determined prior to the execution of the CSP solver. The size of the search space for the problem could be used to calculate the termination point. Statistical significance of the termination point should be investigated to reduce the probability of false alarm further.

- Run time identification of models for the phase transition invariants for various families of combinatorial optimization problems can be investigated. These invariants can be utilized in the quality of service (QoS) subsystem of an agent to detect hard problems with phase transition behavior.

- In this research, the multiobjective optimization problems were limited to combinatorial optimization problems. The experimental system could be extended to address continuous optimization problems as well.

- The mapping from the SCSA architecture to the target CSP programming language was done manually. A way of possibly semi-automatic mapping could be investigated.

- UML and OCL were used for specifying multiobjective optimization problems. It would be desirable to investigate the limits of the expressibility of these languages with respect to the class of multiobjective optimization problems. The expressibility could be compared against other specification languages.

Appendix A

Source Code for the System

A.1 Agent.oz

```
%
% Agent.oz
%
% Executes RAACR constructs..
%
% Author: Yonet Eracar
%
% History:
% 20050128  Updated the control equation
% 20050120  Changed the message passing mechanism..
% 20040620  Creation

functor
import
   FD
```

```
export
   'class':Agent

define

   fun {NewPortObject2 Proc}
      Sin
   in
      thread
      try
        for Msg in Sin do {Proc Msg} end
        catch closeException then skip end
      end

      {NewPort Sin}
   end

   class Agent
     attr id
     solver   % Search object..

     % searchStatus values:   idle
     %                        running
     %                        terminated
     %                        prematurelyTerminated
     searchStatus:idle
```

```
curSoln:nil  % Current solution is
             % always the best solution.

maxDepth:0
lastFailCount:0
lastSuccessCount:0
lastDecPointCount:0
lastSolutionPoint:0

% Communication
recPort % Port to the reconfigurer.
searchDataPort % Port to the search engine..

% Control attributes
goal
controlVar
maxControlVar
minControlVar
k_P
k_D
lastFeedback
terminationPoint:0
logFile
dataVersion

meth init(Val RecP LogFile GoalValue InitialControlVarValue
   K_P K_D TerValue MaxD)
```

```
    id <- Val
    recPort <- RecP
    logFile <- LogFile

    % Initialize the control variables and limits.
    goal <- GoalValue
    controlVar <- InitialControlVarValue
    k_P <- K_P
    k_D <- K_D
    terminationPoint <- TerValue
    maxDepth <- MaxD

    lastFeedback <- GoalValue
    maxControlVar <- 1.0
    minControlVar <- 0

    searchDataPort <- {NewPortObject2
            proc{$ Msg}
               Agent,controlMsgProc(Msg)
            end}
    searchStatus <- running
  end

  meth get_id($)
     @id
  end
```

```
meth updateSearchStats
    CurStats = {@solver stats($)}
in
    lastSolutionPoint <- CurStats.decPoint
end

meth postStabilityProc
    CurStats = {@solver stats($)}
    NewDecCount = CurStats.decPoint - @lastDecPointCount
    DecCountSinceLastSolution =
         CurStats.decPoint - @lastSolutionPoint
in
   % Update MaxD
   if NewDecCount > @maxDepth then
      lastDecPointCount <- CurStats.decPoint
      Agent,control
   end

  if DecCountSinceLastSolution > @terminationPoint then
     Agent,terminate(prematurelyTerminated)
  end
end

meth calculateFeedback(DeltaSuccess DeltaFail $)
   Diff = DeltaSuccess - DeltaFail
   Sum = DeltaSuccess + DeltaFail
   Feedback
```

```
     in
        if Sum == 0 then
           Feedback = @goal
        else
           Feedback = {Float.'/' {Int.toFloat Diff}
                                 {Int.toFloat Sum}}
        end
        Feedback
     end

    meth control
       CurStats = {@solver stats($)}
       Failures = CurStats.failed
       Successes = CurStats.succeded

       DeltaFail = Failures - @lastFailCount
       DeltaSuccess = Successes - @lastSuccessCount

       Feedback = Agent,calculateFeedback( DeltaSuccess DeltaFail $)

       Error = @goal - Feedback
       DeltaFeedback = Feedback - @lastFeedback
     in
        % Save the feedback for future reference..
        lastFeedback <- Feedback

        % Calculate the new control variable..
        controlVar <- Agent,calculateControlVariable(Error
               @controlVar DeltaFeedback $)
       lastSuccessCount <- Successes
       lastFailCount <- Failures
     end

     meth calculateControlVariable(Error CurrentControlVar
                 DeltaFeedback $)
        NewControlVar = CurrentControlVar
                   + @k_P * Error
                   - @k_D * DeltaFeedback
     in
        if NewControlVar > @maxControlVar then
           @maxControlVar
        elseif NewControlVar < @minControlVar then
           @minControlVar
        else
           NewControlVar
        end
     end

     meth filterVariables(FL $)
        Length = {List.length FL}
     in
       Length > 1
     end
```

```
meth selectValue(L $)
  Length = {List.length L}
  SplitPoint = {Float.toInt
         {Int.toFloat Length} * @controlVar}
in
   if SplitPoint < 2 then
     {List.nth L 1}
   elseif SplitPoint > Length then
     {List.nth L Length}
   else
     {List.nth L SplitPoint}
   end
end

meth terminate(S)
   % Stop the search engine only if it is running..
   if @searchStatus == running then
      {@solver stop}
   end

   % Update the search status..
   searchStatus <- S

   if S == terminated then
      raise closeException end
   end
end
```

```
meth controlMsgProc(Msg)
  case Msg
  of filter(FL)#F then
     F = {self filterVariables(FL $)}

  [] value(L)#Val then
     Val = {self selectValue(L $)}

  [] stabilityReached then
     Agent,postStabilityProc

  [] terminatePort then
     Agent,terminate(terminated)
  end
end

% Using Generic distribution..
meth distribute(Xs)

  {FD.distribute
    generic(order:   nbSusps
       filter:  fun{$ D}
            X = {FD.reflect.domList D}
            Y
         in
            {Port.sendRecv @searchDataPort
```

```
                 filter(X) Y}
                Y
             end

           select:  id
           value:   fun {$ D}
                X = {FD.reflect.domList D}
                Y
             in
                {Port.sendRecv @searchDataPort
                 value(X) Y}
                Y
             end
           procedure: proc{$}
              {Port.send @searchDataPort
               stabilityReached}
               end)
    Xs}
  end

  meth nextSol($)
    Temp
  in
    Temp = {@solver next($)}

    % Only save the solution if it is other than nil or stopped..
    if Temp \= nil andthen Temp \= stopped then
```

```
      curSoln <- Temp
    end
    Temp
  end

  meth getObjValue(Sol $)
     SolValue
  in
     case @id
     of 1 then SolValue = {self obj1Value(Sol $)}
     [] 2 then SolValue = {self obj2Value(Sol $)}
     [] 3 then SolValue = {self obj3Value(Sol $)}
     end
     SolValue
  end

  meth objective
    case @id
    of 1 then {self objective1}
    [] 2 then {self objective2}
    [] 3 then {self objective3}
    end
  end

  meth messageQueue(Msg)

    % If we add more messages in the future, we can extend the
```

```
% list below...
case Msg
of setObjective then
   {self initFeat}
   {self objective}
[] soln then S in
   S = {self nextSol($)}

   % Each time send the id, so that the reconfigurer
   % will know the sender.
   if S == nil then
      {Send @recPort agentDone(@id @searchStatus)}

      % Need to kill all active ports...
      {Send @searchDataPort terminatePort}
      raise closeException end
   elseif S == stopped then
      {Send @recPort agentDone(@id prematurelyTerminated)}

      % Need to kill all active ports...
      {Send @searchDataPort terminatePort}
      raise closeException end
   else
      Agent,updateSearchStats
      {Send @recPort newSoln(@id S)}
   end
```

```
        [] objVal(Sol ?Val) then
           Val = {self getObjValue(Sol $)}
        end
      end

      meth reportStatistics(LogFile)
         CurStats = {@solver stats($)}
      in
         if @curSoln \= nil then
            {@logFile write(vs:"Result:  \n\n")}
         else
            {@logFile write(vs:"No Solutions!!\n\n")}
         end
      end
   end
end
```

A.2 Reconfigurer.oz

```
%
% Reconfigurer.oz
%
% Author: Yonet Eracar
%
% History:
% 20050123  Fixing starving thread problem.
% 20040629  Creation
%
functor
import
   Open
export
   'class':Reconfigurer

define
   Y
   class Reconfigurer
     attr
        nObjectives:0   % # of objectives
        blackboard:nil      % reference to the blackboard

        curState % States: start,idle,agent_done,end,error.
        agentPorts
        agents
```

```
      finalSoln
      searchComplete:false
      logFile:{NewCell unit}

   meth init(LogOpt)
      curState <- start

      if LogOpt == true then
         logFile <- {New Open.file
             init(name:'execution.log'
                 flags:[write text create]
                 mode: mode(owner: [read write]
                 group: [read write]))}
      end
   end

   meth setAgentPorts(APs As)
      nObjectives <- {Record.width As}
      agentPorts <- As
      agents <- As
      agents <- {Tuple.make compAg @nObjectives}
   end

   meth getSolution($)
      Temp
   in
      if {Value.isDet @finalSoln} then
```

```
         @finalSoln
      else
         if @curState == endState then
            Temp = Reconfigurer,selectBestSolution($)
            finalSoln <- Temp
         end
         @finalSoln
      end
   end

   meth closeSession
      Reconfigurer, closeLogFile
      raise closeException end
   end

   meth messageQueue(Msg)
      case Msg
      of agentPorts(As) then
         {self setAgentPorts(As)}
      [] ev_create then
         {self stateMachine(ev_create)}
      [] newSoln(I S) then
         {self stateMachine(soln(I S))}
      [] agentDone(I Stat) then
         {self stateMachine(ev_agent_done(I Stat))}
      end
   end
```

```
% This is the procedure that runs the state machine.
meth stateMachine(Msg)
  if Msg == delete_reconfigurer then skip
  else
     case Msg
     of soln(I S) then
        if @curState == idle then
           Reconfigurer,stateNewSolution(soln(I S))
        else
          Reconfigurer,stateError
        end

     [] ev_create then
        if @curState == start then
           Reconfigurer,stateCreate
        else
           Reconfigurer,stateError
        end

     [] ev_idle then
        if {Bool.'or' @curState == idle
           {Bool.'or' @curState == new_solution
           {Bool.'or' @curState == agent_done
            @curState == create}}} then
           Reconfigurer,stateIdle
        else
```

```
               Reconfigurer,stateError
            end

         [] ev_agent_done(I Stat) then
            if @curState == idle then
               Reconfigurer,stateAgentDone(I Stat)
            else
               Reconfigurer,stateError
            end

         [] ev_error then
            Reconfigurer,stateError
         else
            Reconfigurer,stateError
         end
      end
   end

   % The following procedures contain the actions
   % taken at each state. The state diagram can
   % be found in the thesis..

   meth stateError
      curState <- error
   end

   meth stateCreate
```

```
    curState <- create

    % Create threads for all agents..
    for I in 1..@nObjectives do
       {Send (@agentPorts).I setObjective}
    end

    % Tell all agents to start processing..
    for I in 1..@nObjectives do
       {Send (@agentPorts).I soln}
    end

    % Goto the next state
    Reconfigurer,stateMachine(ev_idle)
  end

  meth stateIdle
    curState <- idle
  end

  meth stateNewSolution(S)
    curState <- new_solution

    % Post the solution obtained from the agent..
    % Mark the sender of the last solution
    % Solution is in the form of soln(I,S)
    case S
    of soln(I Sol) then

       % Save the solution as well as the ID for
       % the agent that returned this solution.
       blackboard <- soln(I Sol)|@blackboard

       % Tell the solution sending agent to continue..
       {Send (@agentPorts).I soln}
    end

    % Go back to idle
    Reconfigurer,stateMachine(ev_idle)
 end

 meth stateAgentDone(I Stat)
    curState <- agent_done

    % Mark the agent as done...
    (@agents).I = complete

    % Check whether all agents are done or not..
    searchComplete <- true
    for J in 1..@nObjectives do
       if {Value.isDet (@agents).J} \= true then
          searchComplete <- false
       end
    end

    if Stat == prematurelyTerminated then
       Reconfigurer,
           writeLogFile("Agent "#I#" is prematurely done!")
    else
       Reconfigurer,
           writeLogFile("Agent "#I#" is done!")
    end

    if @searchComplete == true then
       Reconfigurer, stateEnd
    else
       Reconfigurer,stateMachine(ev_idle)
    end
  end

  meth stateEnd
    curState <- endState
    finalSoln <- Reconfigurer,selectBestSolution($)
    Y = @finalSoln
    Reconfigurer,closeSession
  end

  meth writeLogFile(S)
     if @logFile\=unit then {@logFile write(vs:S#"\n")} end
  end
```

```
meth closeLogFile
  if @logFile\=unit then {@logFile close} end
  logFile := unit
end



meth selectBestSolution($)
   case @blackboard
   of nil then
     nil
   [] B1|Br then
      case B1
      of soln(I Sol) then
         if Sol \= prematurelyTerminated then
            Reconfigurer,negotiateSolution(@blackboard $)
         else
            Reconfigurer,recurseBlackboard(Br $)
         end
       else
         nil
       end
     else
       nil
     end
end
```

```
% Recurse the solution list until finding a solution
% other that the prematurelyTerminated flag.

 meth recurseBlackboard(B $)
   case B
   of nil then
      prematurelyTerminated
   [] B1|Br then
      case B1
      of soln(I Sol) then
         if Sol \= prematurelyTerminated then
            Reconfigurer,negotiateSolution(B $)
         else
            Reconfigurer,recurseBlackboard(Br $)
         end
       else
         nil
       end
   else
      nil
   end
 end

% Top level negotiation method.

 meth negotiateSolution(SolStack $)
    AgentStacks = {Tuple.make agStacks @nObjectives}
    Sol
    NonDominatedSolutions
 in
    % Initialize the sub-stacks to be empty list.
    for I in 1..@nObjectives do
       AgentStacks.I = {NewCell nil}
    end

    Reconfigurer,splitStack(SolStack AgentStacks)
    NonDominatedSolutions =
      Reconfigurer,findNonDominatedSolutions(AgentStacks $)
    Sol =
      Reconfigurer,orderNonDominatedSolutions
         (AgentStacks NonDominatedSolutions $)
    Sol
 end

 % Split the solutions in the blackboard to separate lists,
 % where each list contains the solutions returned by a
 % different agent.
 meth splitStack(SolStack AgentStacks)
    case SolStack
    of B1|Br then
       case B1
       of soln(I Sol) then
          if Sol \= prematurelyTerminated then
             if {Access AgentStacks.I} == nil then
```

```
                {Assign AgentStacks.I Sol|nil}
             else
                {Assign AgentStacks.I
                   {List.append
                      {Access AgentStacks.I}
                     Sol|nil}}
             end
          end

          Reconfigurer,splitStack(Br AgentStacks)
       else
          skip
       end
    else
       skip
    end
 end

 % Populates the non-dominated solutions list by comparing the
 % objective values of the new solutions against the ones in
 % the existing non-dominated solutions list.

 meth findNonDominatedSolutions(AgentStacks $)
    NonDominatedSolutions = {NewCell nil}
 in
    % The following loop can be optimized,
    % if we pick one solution from another
```

```
      % agent each time.
      for I in 1..@nObjectives do
         for S in 1..{List.length {Access AgentStacks.I}} do
            NextSolution = {List.nth {Access AgentStacks.I} S}
         in
            Reconfigurer,
              testSolutionForNonDominance
                 (NextSolution NonDominatedSolutions)
         end
      end

      NonDominatedSolutions
   end

   % If a new solution is worse than all the solutions
   % in the non-dominated solutions list, it is
   % dropped from consideration.

   meth testSolutionForNonDominance(Sol NonDominatedSolutions)
      IsNDsolution = {NewCell true}
   in
      % Test the border case of NonDominatedSolutions list
      % being empty..
      if {Access NonDominatedSolutions} == nil then
        {Assign NonDominatedSolutions
           Sol | nil}
      else
```

```
% Need to make sure that we do NOT
% have a non-solution at our hands.
if Sol \= prematurelyTerminated then
   {Assign NonDominatedSolutions
      {List.filter {Access NonDominatedSolutions}
         fun{$ SolND}
            Comparison =
               Reconfigurer,compareSolutions(SolND Sol $)
         in
            if Comparison == better then
               {Assign IsNDsolution false}
               true
            elseif Comparison == worse then
               false
            else
               true
            end
         end}}
end

% If NextSolution is another candidate to be a
% non-dominated solution, it needs to be added
% to the NonDominatedSolutions list.

if {Access IsNDsolution} == true then
   {Assign NonDominatedSolutions
```

```
                {List.append
                  {Access NonDominatedSolutions}
                  Sol|nil}}
          end
        end
     end

     meth compareSolutions(SolND NewSol $)
        AllBetter = {NewCell true}
        AllWorse = {NewCell true}
     in
        for I in 1..@nObjectives do
           Obj = (@agents).I
           SolNDValue = {Obj getObjValue(SolND $)}
           NewSolValue = {Obj getObjValue(NewSol $)}
        in
           if SolNDValue >= NewSolValue then
              {Assign AllWorse false}
           else
              {Assign AllBetter false}
           end
        end

        if {Access AllBetter} then
           better
        elseif {Access AllWorse} then
           worse
```

```
      else
         noDomination
      end
   end

   meth orderNonDominatedSolutions(AgentStacks
            NonDominatedSolutions $)
      BestSolutionIndex = {NewCell 1}
      BestSumOfSquares = {NewCell 1000.0}
      NDSolutionsList = {Access NonDominatedSolutions}
   in
      for S in 1..{List.length NDSolutionsList} do
         SumOfSquares = {NewCell 0.0}
         NewSol
      in
         NewSol = {List.nth NDSolutionsList S}

         % Calculate the variance * (N-1) for each solution.
         for I in 1..@nObjectives do
            Obj = (@agents).I
            BestSol = {List.nth {Access AgentStacks.I} 1}
            BestValue = {Obj getObjValue(BestSol $)}
            NewSolValue = {Obj getObjValue(NewSol $)}
            Delta = {Int.toFloat NewSolValue - BestValue}
            PercentChange = Delta / {Int.toFloat BestValue}
         in
            {Assign SumOfSquares
```

```
                {Access SumOfSquares} +
                PercentChange * PercentChange}
            end

            if {Access SumOfSquares} < {Access BestSumOfSquares} then
               {Assign BestSolutionIndex S}
               {Assign BestSumOfSquares {Access SumOfSquares}}
            end
         end

         {List.nth NDSolutionsList  {Access BestSolutionIndex}}
      end

   end
end
```

A.3 Template OCL File for Optimization Problems

The following is a template OCL file that can be used as a starting point for specifying an optimization problem.

```
package <ProblemName>



-- Object level globals that will be used in the constraints..

context <MainObject> def:

    <Definition for globals: All let expressions>



-- Create the list for the decision variables..

context Scheduler::CreateDecisionVariableList() :
    Sequence(Integer)

    <Definition for the creation of decision variables list>



--  List of all constraints

context Scheduler::Constraints (DecisionVarList :
    Sequence(Integer))
    : ONT_PROC

    pre: DecisionVarList = CreateDecisionVariableList()

    pre: Constraint1(DecisionVarList)
    pre: Constraint2(DecisionVarList)
    pre: Constraint3(DecisionVarList)

    post: ONT_Dist(PC)


context Scheduler::Constraint1(DecisionVarList :
     Sequence(Integer))

    <Definition for the first constraint>


context Scheduler::Constraint2(DecisionVarList :
     Sequence(Integer))

    <Definition for the first constraint>


context Scheduler::Constraint3(DecisionVarList :
    Sequence(Integer))

    <Definition for the first constraint>
```

```
--  Calculation of Objective Values

context Scheduler::obj1Value(Ts : ONT_RECORD) : Integer

    <Definition for objective function 1>

context Scheduler::obj2Value(Ts : ONT_RECORD) : Integer

    <Definition for objective function 1>


-- Objective functions (2 or more)...

context Scheduler::objective1()
    post: ONT_Minimize(obj1Value)

context Scheduler::objective2()
    post: ONT_Minimize(obj2Value)

endpackage
```

A.4 Job Scheduling Benchmark Program

The following is a complete listing of the OZ benchmark program that is used in the job scheduling experiment. Data section at the end was the only changing part while running the experiments.

```
%
% JobSched_nbSusp.oz
%
% Solves the job scheduling problem with most suspensions
% first distribution strategy.
%
% Author: Yonet Eracar
%
local
   Data
   Products
   fun {GetDur TaskSpec}
      {List.toRecord dur {Map TaskSpec fun {$ T}
                       {Label T}#T.dur
                        end}}
   end
   fun {GetStart CusSpec}
      local
     Tasks
      in
     Tasks = {Map CusSpec Label}
     {FD.record start Tasks 0#FD.sup}
      end
   end
   fun {GetTasksOnResource TaskSpec}
      D={Dictionary.new}
   in
      {List.forAll TaskSpec
       proc {$ T}
      if {HasFeature T res} then
         {Dictionary.put D T.res {Label T}|
          {Dictionary.condGet D T.res nil}}
      end
       end}
      {Dictionary.toRecord tor D}
   end
   proc {CheckPreTasks PreTasks  L  DurRecord  Start}
      {List.forAll   PreTasks
       proc{$ P}
      Start.P + DurRecord.P =<: Start.L
       end}
   end
   proc {Constraint1  TaskSpec DurRecord  Start}
      {List.forAll   TaskSpec
       proc{$ T}
      if {HasFeature  T pre } then
         {CheckPreTasks T.pre {Label  T } DurRecord Start}
      end
       end}
```

```
end
proc {CheckDueDate DurRecord  Start  J}
   local
  P =  J.job
   in
  Start.P + DurRecord.P =<: J.due
   end
end
proc {Constraint2  Products DurRecord  Start  }
   {List.forAll   Products
    proc{$ J}
   {CheckDueDate DurRecord Start J}
    end}
end
fun {NegScheduling Data}
   TaskSpec = Data.tasks
   Products = Data.customers
   Dur = {GetDur TaskSpec}
    TasksOnResources = {GetTasksOnResource TaskSpec}
in
   proc {$ Start}
      Start = {GetStart TaskSpec}
      {Constraint1 TaskSpec Dur Start}
      {Constraint2 Products Dur Start}
      {Schedule.serializedDisj  TasksOnResources Start Dur }
      {FD.distribute
         generic(order: nbSusps
            value: min)
          Start}
      end
   end
in
   % The following Data section needs to be
   % modified for each problem instance
   Data = data(tasks: [c0_0(dur:4  res:r4)
        c0_1(dur:9 pre:[ c0_0] res:r2)
        c0_2(dur:8 pre:[ c0_0] res:r4)
        c0_4(dur:4 pre:[ c0_0 c0_2] res:r2)
        c0_5(dur:3 pre:[ c0_0 c0_1 c0_2] res:r3)
        c0_last(dur:0 pre:[ c0_0 c0_1 c0_2 c0_4 c0_5] )
        c1_0(dur:4  res:r4)
        c1_1(dur:9 pre:[ c1_0] res:r2)
        c1_2(dur:8 pre:[ c1_0] res:r4)
        c1_4(dur:4 pre:[ c1_0 c1_2] res:r2)
        c1_last(dur:0 pre:[ c1_0 c1_1 c1_2 c1_4] )
        c2_0(dur:4  res:r4)
        c2_1(dur:9 pre:[ c2_0] res:r2)
        c2_2(dur:8 pre:[ c2_0] res:r4)
        c2_3(dur:2 pre:[ c2_0 c2_1] res:r4)
        c2_4(dur:4 pre:[ c2_0 c2_2] res:r2)
        c2_last(dur:0 pre:[ c2_0 c2_1 c2_2 c2_3 c2_4] )
        c3_0(dur:4  res:r4)
        c3_2(dur:8 pre:[ c3_0] res:r4)
        c3_4(dur:4 pre:[ c3_0 c3_2] res:r2)
        c3_last(dur:0 pre:[ c3_0 c3_2 c3_4])]
           customers: [c0(due:55 job:c0_last)
            c1(due:55 job:c1_last)
            c2(due:55 job:c2_last)
            c3(due:55 job:c3_last)
              ]
          )
  {ExploreOne {NegScheduling Data}}
end
```

A.5 Fixture Design Benchmark Program

The following is a complete listing of the OZ benchmark program that is used in the fixture design experiment. Data section at the end was the only changing part while running the experiments.

```
%
% fxtDesign_benchmark.oz
%
% Solves the fixture design problem with the Explorer utility.
% As a distribution strategy it select the decision variable
% with the most suspensions.
% It does not solve the original optimization problem. It rather
% searches for any solution that will satisfy the constraints.
%
% Author: Yonet Eracar
%
local
   fun {FxtGen Data}
      Requirements = {List.toTuple req Data.pinRequirements}
      NbPins       = {Width Requirements}
      Capabilities = {List.toTuple cap Data.channels}
      NbChannels   = {Width Capabilities}
   in
      proc {$ PC}
        PC = {FD.tuple sol NbPins 1#NbChannels}
        {FD.distinct PC}
        for I in 1..NbPins do
```

```
          for J in 1..NbChannels do
             if {Not {List.sub Requirements.I
                Capabilities.J.capability}}
             then
               PC.I \=: J
             end
          end
       end
       {FD.distribute generic(order: nbSusps
               value:min)
          PC}
       end
    end
    Data = data(pinRequirements: [[1 2] [1] [1 3] [1 2] [3] %5
                [1] [3][1 2][1][3]  %10
                [1] [1 2] [1 2] [1 3] [3] %15
                [1 3] [1 2 3]] % 17 pins altogether
          channels: [ channel(capability:[1] cc:1)    % 1
             channel(capability:[1] cc:1)     % 2
             channel(capability:[1] cc:1)          % 3
             channel(capability:[1] cc:2)
             channel(capability:[1 2] cc:2)
             channel(capability:[1] cc:2)
             channel(capability:[1] cc:2)
             channel(capability:[1 2 3] cc:3)      % 8
             channel(capability:[2 3] cc:3)        % 9
             channel(capability:[1 2 3] cc:3)      % 10
```

```
              channel(capability:[2 3] cc:3)        % 11
              channel(capability:[1 2 3] cc:4)      % 12
              channel(capability:[1 2 3] cc:4)      % 13
              channel(capability:[1 2 3] cc:4)      % 14
              channel(capability:[1 2 3] cc:4)      % 15
              channel(capability:[1 2 3] cc:4)]
          )
in
   {Explorer.object one({FxtGen Data})}
end
```

A.6 Source Code for the OCL Parser

A.6.1 Lex (.l) File for the OCL Parser

The following is a complete listing of the Lex file that contains the lexical analyzer for the OCL parser.

```
%{
#include <ctype.h>
#include <stdio.h>
#include "string.h"
#include "OCLobject.h"
#include "OCLlist.h"
#include "yy2ozy.h"

int yykeyword(char *id);
int yyislegal(short xxchar);
int yylookup(char *id);
extern void OCLReportError(char * msg);
extern int isAmbiguousKeyword(int);

/*
 *  Macros to preprocess and return the tokens...
 */
#define copy_text(){ strcpy(yylval.LastString, yytext); }
#define token(x) { int y = x; copy_text(); return(y); }
#define ifok(x)
{ if (yyislegal(x)) token(x) else token(yylookup(yytext)); }
#ifdef YYLMAX
#undef YYLMAX
#endif
#define YYLMAX  1024

#ifdef YY_FATAL
#undef YY_FATAL
#endif
#define YY_FATAL(msg)
{
   OCLReportError(msg);
}

%}

%p 3000

[ \t]*"--"[^\n\000]*         ; /*token(TCOMMENT); */
\\([^\\\n])*\\               ;
\%           token(yytext[0]);
"<"[ \t]*">" token(TNOTEQUAL);
">"[ \t]*"="    | "="[ \t]*">"  token(TGREATEROREQUAL);
"<"[ \t]*"="    | "="[ \t]*"<" token(TLESSOREQUAL);
"::"               token(TDCOLUMN);
".."               token(TDDOT);
```

```
"->"                token(TARROW);
[0-9]+             ifok(TINT);
[0-9]+\.[0-9]*     ifok(TFLOAT);
([0-9]+\.[0-9]*)[eE][-+]?[0-9]+ ifok(TEXPONENTIAL);
[ \t\n\f\r\032]+         ;
(\%)*[A-Za-z0-9_]+        token(yylookup(yytext));


/*********************************************
 *  This . must be at the end, to catch all
 *  characters that do not match the expressions
 *  above.
 */

.                        token(yytext[0]);

%%

yywrap() {
  return(1);
}

/*
 *  NAME:       isAmbiguousKeyword
 *
 *  INPUTS:     int token,       the token
 *
 *  OUTPUT:     1,    if it is an ambiguos token
 *              0,    otherwise
 *
 *  AUTHOR:     Yonet Eracar (yae)
 *
 *  isAmbiguousKeyword,
 *
 *  NOTE:    This function is used in yylookup() function.
 */
int isAmbiguousKeyword(int token) {
   return 0;
}



/*
 *  NAME:       LtoMyy_resetLexer
 *
 *  INPUTS:     VOID
 *
 *  OUTPUT:     NOTHING
 *
 *  AUTHOR:     Yonet Eracar (yae)
 *
 *  LtoMyy_resetLexer,
 *  initializes start states used in the .SYM Lexer.
 *
 *  NOTE:    This function is called by yyerror() in OCLYacc.y.
 */
void yy_resetLexer() {
    BEGIN INITIAL;
    yy_reset();
}



/* Handle an identifier in the YY input.
   This will return the correct keyword
   token, if it is legal, otherwise it will
   try to find out what is legal and
   return that. */

int yylookup(char *id)
{
  char *c;
  int uscore,alphanum,t;

  /* check token for special characters */
  c = id;
  uscore = 0;
  alphanum = 1;
  while (*c) {
    /* note that _ and $ are considered "alphanumeric" ... */
    if (*c == '_') uscore++;
    else if (! ((*c=='$') || isalnum(*c))) alphanum = 0;
    c++;
  }
  t = yykeyword(id);

  if ((t>0) && (isAmbiguousKeyword(t)
       || yyislegal((short)t)) )
    return(t);
  else
  {
    if (alphanum && yyislegal(TNAME))
        return(TNAME);
    return(TNAME);
  }
}

/* Convert a string to upper case (in place). */
static void cvtupper(char *id)
{
  char *c;

  /* convert token to upper case */
  c = id;
  while (*c) {
    if (islower(*c)) *c = toupper(*c);
    c++;
  }
}
```

```
/* Keyword table.  This must be sorted alphabetically. */
typedef struct {
  char *name;
  int token;
} KITEM;

static KITEM yykeytab[] = {
   #include "yykey.txt"
};

int yykeywordTableSorted()
{
  int i, last;

  i = 0;
  last = (sizeof(yykeytab) / sizeof(KITEM)) - 2;

  while ( i < last )
  {
      if (strcmp(yykeytab[i+1].name, yykeytab[i++].name) < 0)
      {
         printf ("Keyword table is not properly sorted.\n");
         printf ("Please check section around %s.\n",
             yykeytab[i].name);
         return (0);
     }
  }
  return (1);
}

/* Look up the identifier in the keyword table.
   Note that we allow keywords to be abbreviated as
   long as they are not ambiguous.
   If a keyword is found, return the yacc token for the keyword.
   If no keyword is found, return 0.
   If an ambiguous one is found, return -1.

   The keyword is converted
   to upper case (in place!) before the search. */
int yykeyword(char * id)
{
    int low,mid,high,last, c;

//  if (!yykeywordTableSorted())
//  {
//      printf("Not sorted!");
//      return -1;
//  }
    last = (sizeof(yykeytab) / sizeof(KITEM)) - 1;
    low = 0;
    high = last;
    while (low <= high) {
    mid = low + (high-low)/2;
    /* note that an exact match is always nonambiguous */
    if ((c=strcmp(yykeytab[mid].name,id)) == 0)
    {
         return(yykeytab[mid].token);
    }
    else
    {
        if (c<0) low = mid + 1;
        else high = mid - 1;
    }
}

  /* not found */
  return(0);
}
```

A.6.2 Keywords for the OCL Syntax

The following is a complete listing of the keywords file that contains the tokens for the lexical analyzer of the OCL parser.

```
"Bag", TBAG,
"Collection",TCOLLECTION,
"Sequence", TSEQUENCE,
"Set", TSET,
"and", TAND,
"context", TCONTEXT,
"def", TDEF,
"else", TELSE,
"endif", TENDIF,
"endpackage", TENDPACKAGE,
"if", TIF,
"implies", TIMPLIES,
"in",  TIN,
"inv", TINV,
"let", TLET,
"not", TNOT,
"or", TOR,
"package",TPACKAGE,
"post", TPOST,
"pre", TPRE,
"self", TSELF,
"then", THEN,
"xor", TXOR,
```

A.6.3 Yacc (.y) File for the OCL Parser

The following is a complete listing of the Yacc file that contains the production rules for the OCL parser.

```
%{
#include <stdio.h>
#include <string.h>
#include "OCLobject.h"
#include "OCLlist.h"
#include "OCLpackage.h"
#include "OCLconstraint.h"
#include "OCLexpression.h"
#include "ocl2oz_util.h"
%}

%token TAND
%token TARROW
%token TBAG
%token TCOLLECTION
%token TCOMMENT
%token TCONTEXT
%token TDCOLUMN
%token TDDOT
%token TDEF
%token TELSE
%token TENDIF
```

```
%token TENDPACKAGE
%token <LastString>TEXPONENTIAL
%token <LastString>TFLOAT
%token TGREATEROREQUAL
%token THEN
%token TIF
%token TIMPLIES
%token TIN
%token <LastString>TINT
%token TINV
%token TLESSOREQUAL
%token TLET
%token <LastString>TNAME
%token TNOT
%token TNOTEQUAL
%token TOR
%token TPACKAGE
%token TPOST
%token TPRE
%token TSELF
%token TSEQUENCE
%token TSET
%token <LastString>TSTRING
%token TXOR


%union {
    int ivalue;
    char LastString [1024];
    OCLobject * baseObject;
    OCLlist * list;
};

%type <baseObject> additiveExpression classifierContext
%type <baseObject> collectionItem  collectionType constraint
%type <baseObject> context contextDeclaration
%type <baseObject> declarator enumLiteral expression expressionItem
%type <baseObject> ifExpression isAccumulatorTypeSpecifier
%type <baseObject> isPropertyCallParameters isReturnType
%type <baseObject> isSimpleTypeSpecifier isTypeSpecifier
%type <baseObject> letExpression literal literalCollection
%type <baseObject> multiplicativeExpression number oclExpression
%type <baseObject> logicalExpression operationName package
%type <baseObject> parameterItem pathName
%type <baseObject> postfixExpression primaryExpression propertyCall
%type <baseObject> propertyCallItem propertyCallParameters
%type <baseObject> operationContext relationalExpression
%type <baseObject> returnType string simpleTypeSpecifier
%type <baseObject> typeSpecifier unaryExpression

%type <ivalue> addOperator collectionKind logicalOperator
%type <ivalue> multiplyOperator unaryOperator
%type <ivalue> relationalOperator stereotype isTimeExpression
%type <LastString> isName name
```

```
%type <list> actualParameterList collectionList constraintList
%type <list> expressionList formalParameterList isFormalParameterList
%type <list> isQualifiers isActualParameterList letExpressionList
%type <list> nameList oclExpressions
%type <list> packageList parameterList propertyCallList  qualifiers

%%

/************************************************************
 * This is the Production Rules section. More details can be
 * found in Grammars Section of Functional Specification
 */

oclFile:    packageList         {   OCLreportPackages($1); }
    ;

packageList:   package          { $$ = OCLmakeList(NULL, $1);   }
    |   packageList package     { $$ = OCLmakeList($1, $2); }
    ;

package:    TPACKAGE pathName oclExpressions TENDPACKAGE
        {   $$ = OCLprocessPackage($2, $3); }
    ;

oclExpressions:             {   $$ = NULL;  }
    |       constraintList  {   $$ = $1;    }
    ;

constraintList:
        constraint  {   $$ = OCLmakeList(NULL,$1); }
    |   constraintList constraint
        {   $$ = OCLmakeList($1, $2);   }
    ;

constraint:     contextDeclaration expressionList
        {   $$ = OCLprocessConstraint($1, $2); }
    ;

expressionList:
       expressionItem
       {   $$ = OCLmakeList(NULL, $1); }
    |  expressionList expressionItem
       {   $$ = OCLmakeList($1, $2);   }
    ;

expressionItem:  TDEF isName ':' letExpressionList
        {   $$ = OCLprocessDefExpression($2, $4); }

    |   stereotype isName ':' oclExpression
        {   $$ = OCLprocessStereotypeExpression($1, $2, $4); }
    ;

contextDeclaration: TCONTEXT context {  $$ = $2; }
```

```
    ;

context:    operationContext {  $$ = $1; }
    |       classifierContext { $$ = $1; }
    ;

operationContext: name TDCOLUMN operationName
           '(' formalParameterList ')'
           isReturnType
    {   $$ = OCLprocessOperationContext($1, $3, $5, $7); }
    ;

classifierContext:
         name ':' name
         {  $$ = OCLprocessClassifierContext($1, $3); }
    |    name
         {   $$ = OCLprocessClassifierContext(NULL, $1); }
    ;

stereotype:
        TPRE {  $$ = OCL_PRE; }
    |   TPOST  {    $$ = OCL_POST; }
    |   TINV   {    $$ = OCL_INV; }
    ;

operationName:
        name        {   $$ =  OCLprocessUserOperationName($1); }
    |   addOperator         {   $$ = OCLprocessOperationName($1); }
    |   multiplyOperator    {   $$ = OCLprocessOperationName($1); }
    |   logicalOperator     {   $$ = OCLprocessOperationName($1); }
    |   relationalOperator  {   $$ = OCLprocessOperationName($1); }
    ;

formalParameterList:        {   $$ = OCLmakeList(NULL, NULL); }
    |   parameterList       {   $$ = $1; }
    ;

parameterList:  parameterItem
        {   $$ = OCLmakeList(NULL,$1); }
    |   parameterList ',' parameterItem
        {   $$ = OCLmakeList($1, $3);   }
    ;

parameterItem:
     name ':' typeSpecifier   {$$ = OCLprocessParameterItem($1, $3); }
    ;

typeSpecifier:  simpleTypeSpecifier  {  $$ = $1; }
    |           collectionType {    $$ = $1; }
    ;

collectionType: collectionKind '(' simpleTypeSpecifier ')'
        {   $$ = OCLprocessCollectionType($1, $3); }
    |   collectionKind '('  ')'
        {   $$ = OCLprocessCollectionType($1, NULL); }
    ;

oclExpression:  letExpressionList TIN expression
        {   $$ = OCLprocessOclExpression($1, $3); }

    |   expression      {   $$ = $1;    }
    ;

isReturnType:        {  $$ = NULL; }
    |   ':' returnType   {  $$ = $2; }
    ;

returnType: typeSpecifier    {  $$ = $1; }
    ;

expression:     logicalExpression   {   $$ = $1; }
    ;

letExpressionList: letExpression
       {   $$ = OCLmakeList(NULL, $1); }
    |   letExpressionList letExpression
       {   $$ = OCLmakeList($1, $2);   }
    ;

letExpression:  TLET name isFormalParameterList isTypeSpecifier
```

```
'=' expression
            {   $$ = OCLprocessLetExpression($2, $3, $4, $6);   }
    ;


isFormalParameterList:              {   $$ = NULL; }
    |   '(' formalParameterList ')' {   $$ = $2; }
    ;


isTypeSpecifier:            {   $$ = NULL; }
    |   ':' typeSpecifier   {   $$ = $2; }
    ;


ifExpression: TIF expression THEN expression TENDIF
        {   $$ = OCLprocessIfExpression($2, $4, NULL); }

    |   TIF expression THEN expression TELSE expression TENDIF
        {   $$ = OCLprocessIfExpression($2, $4, $6); }
    ;


logicalExpression: relationalExpression
        {   $$ = OCLmakeBinaryExpression(NULL,
                 OCL_ANY, $1, "logicalExpression"); }

    |   logicalExpression logicalOperator relationalExpression
        {   $$ = OCLmakeBinaryExpression($1, $2, $3,
                         "logicalExpression"); }
    ;
```

```
relationalExpression: additiveExpression
        {   $$ = OCLmakeBinaryExpression(NULL, OCL_ANY,
                   $1, "relationalExpression"); }

    |   relationalExpression relationalOperator additiveExpression
        {   $$ = OCLmakeBinaryExpression($1, $2, $3,
                  "relationalExpression"); }
    ;

additiveExpression: multiplicativeExpression
        {   $$ = OCLmakeBinaryExpression(NULL, OCL_ANY, $1,
                 "additiveExpression"); }

    |   additiveExpression addOperator multiplicativeExpression
        {   $$ = OCLmakeBinaryExpression($1, $2, $3,
                 "additiveExpression"); }
    ;

multiplicativeExpression: unaryExpression
        {   $$ = OCLmakeBinaryExpression(NULL, OCL_ANY, $1,
                 "multiplicativeExpression"); }

    |   multiplicativeExpression multiplyOperator unaryExpression
        {   $$ = OCLmakeBinaryExpression($1, $2, $3,
                 "multiplicativeExpression"); }
    ;
```

```
unaryExpression: unaryOperator postfixExpression
        {   $$ = OCLprocessUnaryExpression($1, $2); }

    |   postfixExpression
        {   $$ = OCLprocessUnaryExpression(OCL_ANY, $1); }
    ;

postfixExpression: primaryExpression propertyCallList
        {   $$ = OCLprocessPostfixExpression($1, $2); }

    |   primaryExpression
        {   $$ = OCLprocessPostfixExpression($1, NULL); }
    ;

propertyCallList: propertyCallItem
        {   $$ = OCLmakeList(NULL, $1); }

    |   propertyCallList propertyCallItem
        {   $$ = OCLmakeList($1, $2); }
    ;

propertyCallItem: '.' propertyCall
        {   $$ = OCLpropertyCallItem($2, OCL_DOT); }

    |   TARROW propertyCall
        {   $$ = OCLpropertyCallItem($2, OCL_ARROW); }
    ;
```

```
primaryExpression:
        literalCollection   {   $$ = $1; }
    |   literal             {   $$ = $1; }
    |   propertyCall        {   $$ = $1; }
    |   '(' expression ')'  {   $$ = OCLprocessInParenthesis($2); }
    |   ifExpression        {   $$ = $1; }
    ;

propertyCall: pathName isTimeExpression isQualifiers
isPropertyCallParameters
        {   $$ = OCLpropertyCallAction($1, $2, $3, $4); }
    ;

isQualifiers:       {   $$ = NULL;  }
    |   qualifiers  {   $$ = $1;    }
    ;

qualifiers: '[' actualParameterList ']' {   $$ = $2;    }
    ;

isPropertyCallParameters:       {   $$ = NULL;  }
    |   propertyCallParameters  {   $$ = $1;    }
    ;

propertyCallParameters: '(' isActualParameterList ')'
        {   $$ = OCLprocessPropertyCallParameters(NULL, $2); }
    |   '(' declarator isActualParameterList ')'
        {   $$ = OCLprocessPropertyCallParameters($2, $3); }
    ;



declarator: nameList [^ ')'] isSimpleTypeSpecifier
              isAccumulatorTypeSpecifier '|'
        {   $$ = OCLprocessDeclarator($1, $3, $4); }
    ;


nameList:   name
        {   $$ = OCLnameList(NULL, $1); }
    |   nameList  name [^ ')']
        {   $$ = OCLnameList($1, $2); }
    ;


isSimpleTypeSpecifier: {    $$ = NULL; }
    |   ':' simpleTypeSpecifier {   $$ = $2; }
    ;


isAccumulatorTypeSpecifier:  {  $$ = NULL; }
    |   ';' name ':' typeSpecifier '=' expression
        {   $$ = OCLisAccumulatorTypeSpecifier($2, $4, $6); }
    ;


literal:    string      {   $$ = OCLliteral($1); }
```

```
    |   number          {   $$ = OCLliteral($1); }
    |   enumLiteral     {   $$ = OCLliteral($1); }
    |   TSELF           {   $$ = OCLliteral(NULL); }
    ;

enumLiteral:    name TDCOLUMN name
        {   $$ = OCLprocessEnumLiteral(NULL, $1, $3); }
    |   enumLiteral TDCOLUMN name
        {   $$ = OCLprocessEnumLiteral($1, NULL, $3); }
    ;

simpleTypeSpecifier: pathName
        {   $1->SetType(OCLtype_SIMPLETYPESPECIFIER);   $$ = $1; }
    ;

literalCollection: collectionKind '{' collectionList '}'
        {   $$ = OCLprocessLiteralCollection($1, $3); }
    |   collectionKind '{' '}'
        {   $$ = OCLprocessLiteralCollection($1, NULL); }
    ;

collectionList: collectionItem
        {   $$ = OCLmakeList(NULL, $1); }
    |   collectionList ',' collectionItem
        {   $$ = OCLmakeList($1, $3); }
    ;
```

```
collectionItem:
        expression     { $$ = OCLprocessCollectionItem(NULL, $1); }
    |   expression TDDOT expression
        { $$ = OCLprocessCollectionItem($1, $3); }
    ;

pathName:   name {  $$ = OCLpathName($1); }
    |   enumLiteral
        {   $1->SetType(OCLtype_PATHNAME);  $$ = $1; }
    ;

isTimeExpression:           {   $$ = OCL_NOTPRE;    }
    |   timeExpression      {   $$ = OCL_TPRE;  }
    ;

timeExpression: '@' TPRE
    ;

isActualParameterList:          {   $$ = NULL;  }
    |   actualParameterList     {   $$ = $1;    }
    ;

actualParameterList:
        expression
          { $$ = OCLmakeList(NULL, $1);   }
    |   actualParameterList ',' expression
          { $$ = OCLmakeList($1, $3); }
    ;

logicalOperator:
        TAND    {   $$ = OCL_AND; }
    |   TOR         {   $$ = OCL_OR; }
    |   TXOR        {   $$ = OCL_XOR; }
    |   TIMPLIES    {   $$ = OCL_IMPLIES; }
    ;

collectionKind: TSET    {   $$ = OCL_SET; }
    |   TBAG            {   $$ = OCL_BAG; }
    |   TSEQUENCE       {   $$ = OCL_SEQUENCE; }
    |   TCOLLECTION     {   $$ = OCL_COLLECTION; }
    ;

relationalOperator:
        TNOTEQUAL           {   $$ = OCL_NOTEQUAL; }
    |   TGREATEROREQUAL     {   $$ = OCL_GREATEROREQUAL; }
    |   TLESSOREQUAL        {   $$ = OCL_LESSOREQUAL; }
    |   '>'         {   $$ = OCL_GREATER; }
    |   '<'         {   $$ = OCL_LESS; }
    |   '='         {   $$ = OCL_EQUAL; }
    ;

addOperator:    '+'     {   $$ = OCL_Plus; }
    |   '-'     {   $$ = OCL_Minus; }
    ;
```

```
multiplyOperator:   '*'     {   $$ = OCL_Mul; }
    |   '/'     {   $$ = OCL_Div; }
    ;

unaryOperator: '-' {    $$ = OCL_Minus; }
    |   TNOT        {   $$ = OCL_NOT; }
    ;

isName:         { strcpy($$, ""); }
    |   name    { strcpy($$, $1); }
    ;

name:   TNAME   { strcpy($$, $1); }
    ;

number:     TINT
            { $$ = new OCLobject($1, OCLtype_INT); }
    |       TFLOAT
            { $$ = new OCLobject($1, OCLtype_FLOAT); }
    |       TEXPONENTIAL
            { $$ = new OCLobject($1, OCLtype_EXPONENTIAL); }
    ;

string:  TSTRING   { $$ = new OCLobject($1,OCLtype_STRING); }
    ;
```

Bibliography

[1] Analytical Tools to Evaluate Negotiation Difficulty (ATTEND). URL: www.isi.edu/attend/, USC: Information Sciences Institute, 2003.

[2] Autonomous Negotiating Teams Program (ANT). URL: www.if. afrl.af.mil /div/ift/iftb/ants/ants.html, IFTB: Information Awareness and Understanding, 2003.

[3] D. Achlioptas, L.M. Kirousis, E. Kranakis, M.S. Krizanc, M.S.O. Molloy, and Y.C. Stamatiou. Random constraint satisfaction: A more accurate picture. In *Proc. 3rd Int. Conf. on Principles and Practice of Constraint Programming (CP98)*. Springer-Verlag, LNCS 1330, 1998.

[4] R.K. Ahuja, T.L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[5] K. Andersson and T. Hjerpe. Modeling constraint problems in CML. In *Proc. PACT98*, 1998.

[6] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. On the solution of traveling salesman problems. *Documenta Mathematica*, Extra Volume Proceedings ICM III (1998):645–656, 1998.

[7] N. Badr, D. Reilly, and A. Taleb-Bendiab. A Conflict Resolution Control Architecture for Self-Adaptive Software. In *ICSE 2002 Workshop on Architecting Dependable Systems*, Orlando,Florida, May 2002.

[8] S. Bistarelli, U. Montanari, and F. Rossi. Constraint solving over semirings. In *Proc. IJCAI95*. Morgan Kauffman, 1995.

[9] A. Borning. The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *ACM Transaction on Programming Languages and Systems*, 3(4):353–387, 1981.

[10] A. Borning and B. Freeman-Benson. Ultraviolet: A constraint satisfaction algorithm for interactive graphics. *CONSTRAINTS: An International Journal*, 3(1), 1998.

[11] A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint hierarchies and logic programming. In M. Martelli and G. Levi, editors, *Proc. 6th Int. Conf. on Logic Programming*, pages 149–164. MIT Press, 1989.

[12] M. Brandozzi and D. E. Perry. Architectural Prescriptions for Dependable Systems. In *ICSE 2002 Workshop on Architecting Dependable Systems*, Orlando,Florida, May 2002.

[13] A. Brodsky. Constraint databases: Promising technology or just intellectual exercise? *CONSTRAINTS: An International Journal*, 2(1):33–44, 1997.

[14] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the Really Problems Are. In *Proc. IJCAI91*, pages 331–337, 1991.

[15] A. Colmerauer. An Introduction to Prolog III. *Communications of ACM*, 28(4):412–418, August 1990.

[16] A. Colmerauer. Specification de Prolog IV. Technical report, Laboratoire d'informatique de Merseille, 1996.

[17] J. Csontó and J. Paralič. A Look at CLP: Theory and Application. *Applied Artificial Intelligence*, 11(1):59–69, 1997.

[18] M.R. Cutkovsky, R.S. Engelmore, R.E. Fikes, M.R. Genesereth, T.R. Gruber, W.S. Mark, J.M. Tenenbaum, and J.C. Weber. PACT: An experiment in integrating concurrent engineering systems. *IEEE Computer*, 26(1):28–37, January 1993.

[19] B. Van de Walle and P. Faratin. Fuzzy Preferences for Multi-Criteria Negotiation. In Costas Tsatsoulis, editor, *Negotiation Methods for Autonomous Cooperative Systems*, pages 116–118. AAAI Press, November 2001.

[20] T. Dean and M. Boddy. An Analysis of Time-Dependent Planning. In *Proceedings AAAI-88*, pages 49–54, St. Paul, Minnesota, 1988.

[21] T. Dean and M. Boddy. Decision-Theoretic Deliberation Scheduling for Problem Solving in Time-Constrained Environments. *Artificial Intelligence*, 67(2):245–286, 1994.

[22] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley and Sons, Co., 2001.

[23] R. Dechter. On the expressiveness of networks with hidden variables. In *Proc. of the Eight National Conference on Artificial Intelligence (AAAI-90)*, pages 556–562, Boston, MA, July 1990.

[24] R. Dechter. From local to global consistency. *Artificial Intelligence*, 55:87–107, 1992.

[25] B. Demoen, M. G. de la Banda, W. Harvey, K. Marriott, and P. Stuckey. An overview of HAL. In *Principles and Practice of Constraint Programming*, pages 174–188, October 1999.

[26] F.Y. Edgeworth. *Mathematical Physics: An Essay on the Application of Mathematics to the Moral Sciences.* Kegan Paul and Co., London, 1881.

[27] M. Ehrgott and X. Gandibleux. An Annotated Bibliography of Multi-objective Combinatorial Optimization. Technical report, Fachbereich Mathematic – Universitat Kaiserslautern, April 2000.

[28] M. Ehrgott and X. Gandibleux. *Multiple Criteria Optimization: State of the Art Annotated Bibliographic Surveys*, volume 52 of *Operations Research and Management Science*. Kluwer Academic Publishers, Boston, June 2002.

[29] H. El Sakkout and M. Wallace. Probe Backtrack Search for Minimal Perturbation in Dynamic Scheduling. *Constraints*, 5(4), 2000.

[30] Y. A. Eracar. RAACR: A Reconfigurable Architecture for Adapting to Changes in the Requirements. Master's thesis, Northeastern University, Boston, MA, September 1996.

[31] H. Eschenauer, J. Koski, and A. Osyczka. *Multicriteria Design Optimization.* Springer-Verlag, Berlin, 1990.

[32] P. Faratin, C. Sierra, and N. R. Jennings. Negotiation decision functions for autonomous agents. *Int. Journal of Robotics and Autonomous Systems*, 24(3-4):159–182, 1998.

[33] R.E. Fikes. REF-ARF: a system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

[34] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In *Third International Conference on Information and Knowledge Management.* ACM Press, November 1994.

[35] M.L. Fisher. The lagrangian method for solving integer programming problems. *Management Science*, (27):1–18, 1981.

[36] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* The Scientific Press, San Francisco, CA, 1998.

[37] M. Frank, A. Bugacov, J. Chen, G. Dakin, P. Szekely, and B. Neches. The Marbles Manifesto: A Definition and Comparison of Cooperative Negotiation Schemes for Distributed Resource Allocation. In *AAAI 2001 Fall Symposium on Negotiation Methods for Autonomous*

Cooperative Systems, pages 36–45, North Falmouth, Massachusetts, November 2001.

[38] E. C. Freuder. Synthesizing constraint expressions. *Communications of ACM*, 21(11), 1978.

[39] E. C. Freuder. *In Kanal and Kumar, editors, Search in Artificial Intelligence*, chapter Backtrack-free and backtrack-bounded search. Springer-Verlag, 1988.

[40] E. C. Freuder and R.J. Wallace. Suggestion strategies for constraint-based matchmaker agents. In *Proc. 3rd Int. Conf. on Principals and Practice of Constraint Programming (CP97)*. Springer-Verlag, LNCS 1330, 1998.

[41] D. Garlan, B. Schmerl, and J. Chang. Using Gauges for Architecture-Based Monitoring and Adaptation. In *The Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Australia, 2001.

[42] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms.* PhD thesis, Dept. of Computer Science, Carnegie Mellone University, 1979.

[43] I. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh. The constrainedness of arc consistency. In *Proc. 3rd Int. Conf. on Principals and Practice of Constraint Programming (CP97).* Springer-Verlag, LNCS 1330, 1997.

[44] R. P. Goldman, D. J. Musliner, K. D. Krebsbach, and M. S. Boddy. Dynamic Abstraction Planning. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 680–686, Providence, Rhode Island, 1997. AAAI Press / MIT Press.

[45] R.E. Gomory. Outline of an algorithm for integer solution to linear programs. *Bulletin American Mathematical Society*, 64:275–278, 1958.

[46] I.J. Good. Twenty-seven principles of rationality. In V.P. Godambe and D.A. Sprott, editors, *Foundations of statistical inference*, pages 108–141. Holt Rinehart Wilson, Toronto, 1971.

[47] M. A. Goodrich, W. C. Stirling, and E. R. Boer. Satisficing revisited. *Minds and Machines*, 10:79–109, February 2000.

[48] J. Grass and S. Zilberstein. Programming with Anytime Algorithms. In *Proceedings of the IJCAI-95 Workshop on Anytime Alorithms and Deliberation Scheduling*, pages 91–94, Montreal, Canada, 1995.

[49] M. Grotschel and L. Laszlo. Combinatorial optimization: A survey. Technical Report 93-29, DIMACS, May 1993.

[50] M. Grotschel and M.W. Padberg. On the symmetric travelling salesman problem ii: lifting theorems and facets. *Mathematical Programming*, (16):281–302, 1979.

[51] M. Guignard and S. Kim. Lagrangian decomposition: a model yielding stronger lagrangian bounds. *Mathematical Programming*, 39:215–228, 1987.

[52] J.A. Hanley and B.J. McNeil. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *RADIOLOGY*, 143(1):29–36, April 1982.

[53] S. Haridi and S. Janson. Kernel Andorra Prolog and its computational model. In *Proc. ICLP90*. MIT Press, 1990.

[54] B. Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26(3):251–321, July 1985.

[55] M. Hermenegildo and CLIP group. Some methodologocal issues in the design of CIAO - a generic, parallel concurrent constraint system. In *Principals and Practice of Constraint Programming (CP97)*. Springer-Verlag, LNCS 874, 1994.

[56] K. Hoffman. Combinatorial and integer optimization. URL:http://iris.gmu.edu/ khoffman/papers/newcomb1.html, George Mason University.

[57] R.M. Hogarth and M.W. Reder. *Rational Choice*. Univ. Chicago Press, Chicago, 1986.

[58] T. Hogg. Quantum Computing and Phase Transitions in Combinatorial Search. *Journal of Artificial Research*, 4:91–128, 1996.

[59] T. Hogg. Which search problems are random? In *Proc. of AAAI98*, pages 438–443, Menlo Park, CA, 1998. AAAI Press.

[60] T. Hogg, B. A. Huberman, and C. Williams. Frontiers in Problem Solving: Phase Transitions and Complexity, Introduction. *Special issue of Artificial Intelligence*, 81(1–2):1–15, March 1996.

[61] T. Hogg and C. P. Williams. The hardest constraint problems: A double phase transition. *Artificial Intelligence*, 69:359–377, 1994.

[62] E. J. Horvitz. Reasoning About Beliefs and Actions Under Computational Resource Constraints. In *Proceedings of the Third Workshop on Uncertainty in Artificial Intelligence*, Seattle, Washington, 1987.

[63] J. Jaffar and J.L. Lassex. Constraint Logic Programming. In *POPl-87*, Munich,FRG, January 1987.

[64] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In *Fourth International Conference in Logic Programming*, Melbourne, Australia, May 1987.

[65] N. Jennings, P. Faratin, M. Johnson, T. Norman, P. O'brien, and M. Wiegand. Agent-based business process management. *International Journal on Intelligent Cooperative Information Systems*, 5(2–3):105–130, 1996.

[66] N. R. Jennings, T. J. Norman, and P. Faratin. ADEPT: An Agent-based Approach to Business Process Management. *ACM SIGMOD*, 27(4):32–39, 1998.

[67] E.L. Johnson and S. Powell. *Integer programming codes*, chapter Design and Implementation of Optimization Software, pages 225–248. NATO Advanced Study Institute. Sijthoff and Noordhoff, 1978.

[68] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51(1):25–52, 1995.

[69] G. Karsai and J. Sztipanovitz. A Model-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems and Their Applications*, 14(3):46–53, May/June 1999.

[70] M.M. Kokar, K. Baclawski, and Y.A. Eracar. Control Theory-Based Foundations of Self-Controlling software. *IEEE Intelligent Systems and Their Applications*, 14(3):37–45, May/June 1999.

[71] M.M. Kokar, K.M. Passino, K. Baclawski, and J.E. Smith. Mapping an Application to a Control Architecture: Specification of the Problem. In *IWSAS*, volume 1936 of *Lecture Notes in Computer Science*, pages 75–89. Springer, 2001.

[72] G. Kondrak and P. van Beek. A theoretical valuation of selected algorithms. *Artificial Intelligence*, 89:365–387, 1997.

[73] V. Kumar. Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine*, 13(11):32–44, 1992.

[74] F. Laburthe and Y. Caseau. SALSA: A Language for Search Algorithms. In *Fourth International Conference on Principals and Practice of Constraint Programming (CP'98)*, Pisa, Italy, October 1998.

[75] R. Laddaga. Self-Adaptive Software. Proposer Information Pamphlet BAA No 98-12, DARPA: Information Processing Technology Office, 1998.

[76] R. Laddaga. Creating Robust Software through Self-Adaptation. *IEEE Intelligent Systems and Their Applications*, 14(3):26–29, May/June 1999.

[77] C. Le Pape and P. Baptiste. A constraint programming library for preemptive and non-preemptive scheduling. In *Proc. PACT97*, 1997.

[78] C. Le Pape and P. Baptiste. Resource constraints for preemptive job-shop scheduling. *CONSTRAINTS: An International Journal*, 3(4), 1998.

[79] A. Ledeczi. Adaptive Model-Integrated Computing. URL: http://www. isis.vanderbilt.edu /projects/asc/asc.htm, ISIS Vanderbilt University.

[80] A. K. Mackworth. Networks of constraints: Fundemental properties and application to picture processing. *Artificial Intelligence*, 8(1), 1977.

[81] P. Maes. General tutorial on software agents. pattie.www.media.mit.edu, MIT, 1997.

[82] K. Marriott, S.S. Chok, and A. Finlay. A tableau based constraint solving toolkit for interactive graphical applications. In *Proc. 4th Int. Conf. on Principles and Practice of Constraint Programming (CP98)*. Springer-Verlag, 1998.

[83] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley and Sons., New York, 1990.

[84] M. Mehl, M. Muller, T. Popov, and K. Scheidhauer. DFKI Oz User's Manual, May 1995.

[85] A.C. Meng. On Evaluating Self-Adaptive Software. In *IWSAS*, volume 1936 of *Lecture Notes in Computer Science*, pages 65–74. Springer, 2001.

[86] L. Michel and P. Van Hentenryck. Modeler++: A Modeling Layer for Constraint. Programming Libraries CS-00-07, Brown University, December 2000.

[87] U. Montanari. Networks of constraints: Fundemental properties and application to picture processing. *Information Science*, 7, 1974.

[88] U. Montanari and F. Rossi. Constraint relaxation may be perfect. *Artificial Intelligence Journal*, 48:143–170, 1991.

[89] U. Montanari and F. Rossi. Constraint solving and programming: What's next? *ACM Computing Surveys*, 28(4), 1996.

[90] T. K. Moon and W. C. Stirling. Satisficing Negotiation for Resource Allocation with Disputed Resources. In Costas Tsatsoulis, editor, *Negotiation Methods for Autonomous Cooperative Systems*, pages 106–115. AAAI Press, November 2001.

[91] A.I. Mouaddib and S. Zilberstein. Knowledge-Based Anytime Computation. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 775–781, Montreal, Canada, 1995.

[92] D. J. Musliner. Imposing Realtime Constraints on Self-Adaptive Controller Synthesis. In P. Robertson, H. Shrobe, and R. Laddaga, editors, *Self-Adaptive Software, First International Workshop, IWSAS 2000*, volume 1936 of *Lecture Notes in Computer Science (LNCS)*, pages 143–160, Oxford, UK, April 2000. Springer-Verlag Berlin Heidelberg.

[93] Object Management Group, Inc. *The Common Object Request Broker: Architecture and Specification*, 3.0.2 edition, December 2002.

[94] Object Management Group, Inc. *XML Metadata Interchange Specification*, 1.2 edition, January 2002.

[95] Object Management Group, Inc. *Unified Modeling Language Specification*, 1.5 edition, March 2003.

[96] M. Padberg and G. Rinaldi.

[97] L Padulo and M.A. Arbib. *System Theory: A unified state-space approach to continuous and discrete systems.* W.B Saunders Company, Philedelphia,PA, 1974.

[98] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Dover Publications, Inc., 1998.

[99] V. Pareto. *Manual of Political Economy.* Societa editrice libraria, Milan, 1906.

[100] D. Pavlovic. Towards self-Adaptive Software. In P. Robertson, H. Shrobe, and R. Laddaga, editors, *Self-Adaptive Software, First International Workshop, IWSAS 2000*, volume 1936 of *Lecture Notes in Computer Science (LNCS)*, pages 65–74, Oxford, UK, April 2000. Springer-Verlag Berlin Heidelberg.

[101] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81, 1996.

[102] D.G. Pruitt. *Negotiation Behavior.* Academic Press, 1981.

[103] P.W. Purdom. Search Rearrangement Backtracking and Polynomial Average Time. *Artificial Intelligence*, 21(1–2):11–134, 1983.

[104] P. R., H. E. Shrobe, and R. Laddaga, editors. *Self-Adaptive Software, First International Workshop, IWSAS 2000, Oxford, UK, April 17-19, 2000, Revised Papers*, volume 1936 of *Lecture Notes in Computer Science*. Springer, 2001.

[105] H. Raiffa. *The Art and Science of Negotiation.* Harvard University Press, Cambridge, USA, 1982.

[106] Anita Raja and Victor Lesser. Efficient meta-level control in bounded rational agents. Technical report, UMass Computer Science Technical Report 2002-051, December 2002.

[107] S. Reece. Self-Adaptive Multi-Sensor Systems. In P. Robertson, H. Shrobe, and R. Laddaga, editors, *Self-Adaptive Software, First International Workshop, IWSAS 2000*, volume 1936 of *Lecture Notes in Computer Science (LNCS)*, pages 224–241, Oxford, UK, April 2000. Springer-Verlag Berlin Heidelberg.

[108] F. Rossi. Constraint Logic Programming. In *Proc. ERCIM/Compulog Net workshop on constraints.* Springer-Verlag, LNAI 1865, 2000.

[109] F. Rossi, C. Petrie, and V. Dhar. On the Equivalence of Constraint Satisfaction Problems. Technical Report ACT-AI-222-89, MCC, Austin,TX, 1989.

[110] F. Rossi and A. Sperduti. Learning solution preferences in constraint problems. *Journal of Experimental and Theoretical Computer Science*, 10, 1998.

[111] D. Roth. On the Hardness of Approximate Reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996.

[112] S. Russell. Rationality and Intelligence. In Renee Elio, editor, *Common sense, reasoning, and rationality*. Oxford University Press, 2002.

[113] S. Russell and S. Zilberstein. Composing Real-time Systems. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, Sydney, August 1991. Morgan Kaufmann.

[114] S. Russell and S. Zilberstein. Optimal Composition og Real-time Systems. *Artificial Intelligence*, 83, 1996.

[115] S. J. Russell and D. Subramanian. Provably bounded optimal agents. *Journal of Artificial Intelligence Research*, 3, May 1995.

[116] C. Schulte. Programming Constraint Inference Engines. In *Proceedings of the Third International Conference on Principals and Practice of Constraint Programming*, volume 1330, pages 519–533, Schloss Hagenberg, Linz, Austria, October 1997. Springer-Verlag.

[117] C. Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, Berlin, Germany, 2002.

[118] C. Schulte and G. Smolka. *Finite Domain Constraint Programming in OZ.* Mozart Documentation, 1.3.1 edition, June 2004.

[119] I. Seilonen, G. Teunis, and P. Leitao. Mediator-based communication, negotiation, and scheduling for decentralized production management. Grenoble, France, July 2000. MCPL'2000 Management and Control of Production and Logistics Conference.

[120] B. Selman and H. Kautz. Knowledge compilation and theory approximation. *Journal of the ACM*, 43(2):193–224, 1996.

[121] B. Selman and S. Kirkpatrick. Critical behavior in the computational cost of satisfiability testing. *Artificial Testing*, 81:273–295, 1996.

[122] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, 1992.

[123] S. Sen, editor. *Satisficing Models.* AAAI Press, 1998.

[124] H. Simon and J. Schaeffer. The Game of Chess. In Aumann and Hart, editors, *Handbook of Game Theory with Economic applications.* North Holland, 1992.

[125] H. A. Simon. A Behavioral Model of Rational Choice. *Quarterly Journal of Economics*, 59:99–118, 1955.

[126] H. A. Simon. Rational Choice and Structure of the Environment. *Psychological Review*, 63(2):129–138, 1956.

[127] H. A. Simon. Invariants of Human Behavior. *Annual Review Psychology*, 41:1–19, 1990.

[128] B. Smith and M.E. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81, 1996.

[129] G. Smolka. *An OZ Primer.* DFKI Oz documentation series, Saarbrucken, Germany, 1995.

[130] W. C. Stirling and M. A. Goodrich. Satisficing games. *Information Sciences*, 114:255–280, March 1999.

[131] W. C. Stirling, M. A. Goodrich, and D. J. Packard. Satisficing equilibria: A non-classical approach to games and decisions. *Autonomous Agents and Multi-Agent Systems Journal*, 5:305–328, 2002.

[132] W. C. Stirling and T. K. Moon. A Praxeology for Rational Negotiation. In Costas Tsatsoulis, editor, *Negotiation Methods for Autonomous Cooperative Systems*, pages 79–88. AAAI Press, November 2001.

[133] G.J. Sussman and G.L. Steele. CONSTRAINTS-a language for expressing almost-hierarchical descriptions. *AI Journal*, 14(1), 1980.

[134] I.E. Sutherland. *SKETCHPAD: A Man-Machine Graphical Communication System.* MIT Lincoln Labs, Cambridge,MA, 1963.

[135] P. Szekely, B. Neches, D. Benjamin, J. Chen, and C. M. Rogers. DEALMAKER: An Agent for Selecting Sources of Supply to Fill Orders. In *The Agents'99 Workshop on Agent-based Decision-Support for Managing the Internet-Enabled Supply Chain.*, Seattle, Washington, May 1999.

[136] J. Sztipanovits and G. Karsai. Model-Integrated Computing. *IEEE Computer*, pages 110–112, April 1997.

[137] E. Tsang, P. Mills, R. Williams, J. Ford, and J. Borrett. A computer aided constraint programming system. In *Proceedings PACPL'99*, 1999.

[138] C. Van Buskirk, B. Dawant, G. Karsai, Sprinkle J., Szokoli G., and K. Suwanmongkol. Computer-Aided Aircraft Maintenance Scheduling. Technical Report ISIS–02–303, ISIS Vanderbilt University, July 2002.

[139] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

[140] P. Van Hentenryck. Helios: A modeling language for global optimization. In *Proceedings PACT96*, pages 317–335, 1996.

[141] P. Van Hentenryck. *Numerica: A modeling language for global optimization.* MIT Press, 1997.

[142] P. Van Hentenryck. Visual Solver: A modeling language for constraint programming. In *Proc. 3rd Int. Conf. on Principals and Practice of Constraint Programming (CP97)*, volume 2. Springer-Verlag, LNCS 1330, 1997.

[143] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58:113–159, 1992.

[144] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, 2004.

[145] D. Waltz. Understanding Line Drawings of Scenes with Shadows. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw Hill, Cambridge, MA, 1975.

[146] Z. Xanthopulos, E. Melachrinoudis, and M.M. Solomon. Interactive multiobjective group decision making with interval parameters. *Management Science*, 46(12):1585–1601, December 2000.

[147] K. Xu and W. Li. Exact phase transitions in random constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 12:93–103, 2000.

[148] X. Zhang, V. Lesser, and T. Wagner. A Proposed Approach to Sophisticated Negotiation. In Costas Tsatsoulis, editor, *Negotiation Methods for Autonomous Cooperative Systems*, pages 96–105. AAAI Press, November 2001.

[149] S. Zilberstein. *Operational Rationality through Compilation of Anytime Algorithms.* Ph.D. dissertation, Computer Science Division, University of California at Berkeley, 1993.

[150] S. Zilberstein. Satisficing and Bounded Optimality. In *Proceedings of the 1998 AAAI Symposium. Technical Report SS-98-05*, pages 91–94, Stanford, California, March 1998.