A Negotiation-Based Approach to Finding Satisficing Solutions of Declaratively Specified Multi-Objective Optimization Problems

Yönet A. Eracar

November 6, 2003

Abstract

Many problems in engineering can be classified as *multi-objective optimization problems.* These problems are formulated in terms of performance criteria and constraints. To find a solution one needs to find an instantiation of the problem variables that satisfies the constraints and at the same time results in high values of the performance criteria. Two examples of such problems are design of a system and order negotiation in the manufacturing context. In design, one has to select components and a structure such that the components “fit together” while the resulting system delivers the desired performance, e.g., the system is 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 such as to satisfy goals of both the manufacturers and the customers. The manufacturer wants to maximize the price and lead times, while the customer is interested in the opposite. The constraints include manufacturing plant capabilities, storage capabilities and other.

Problems of this kind are known to be NP-hard and thus one cannot expect full algorithmic solutions. Moreover, since multiple performance criteria are involved, any solution needs to make tradeoffs among them. Consequently, one has to settle for less than optimality. Conceptualization of this kind of problem formulation is known as *satisficing* solutions or *good-enough-soon-enough* solutions. Solutions of this kind of problems 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. Consequently, a special program needs to be developed for any specific problem/domain. What if the problem formulation changes? Then either a new program needs to be developed, or the original program needs to have built-in mechanisms for adapting to changes. The latter approach is appropriate when the changes in the constraints are parametric, while the former applies to non-parametric changes in constraints and objective functions.

This thesis addresses this specific problem - how can a satisficing program be automatically synthesized from specifications such that it can: (1) monitor and control its complexity, (2) adapt to parametric changes in the constraints. In order to develop such a program we need to answer a number of questions. What is the language in which problem specifications can be expressed? What algorithm or system can be used to search for solutions to multi-objective 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 and program adaptation?

This research will investigate the use of: UML (Unified Modeling Language) to capture the structural aspects of the user’s view of problem formulations; OCL (Object Constraint Language) to capture quantitative and global constraints; a general Constraint Problem Solver to implement search; Self-Controlling Program Architecture to implement self-control and adaptation; mapping of multi-objective optimization problems to an agent based architecture in which each criterion (goal) is “represented”

by an agent; negotiation as a mechanism to determine the aspiration levels for each objective function; *phase transitions* for the purpose of monitoring for computation intensive regions and controlling the program to avoid these regions.

To evaluate the approach, an experimental system will be implemented and tested on two scenarios - a design scenario and an order negotiation scenario. Problem formulations will be specified using the UML representation. The formulations will be automatically translated to the selected Constraint Satisfaction Problem (CSP) language and then used by the system to find satisficing solutions. The goal will be to provide a proof-of-concept of developing such a self-controlling satisficing program that (1) is applicable to various multi-objective optimization problems, (2) has the ability to control its own complexity, and (3) can adapt to parametric changes in the constraints. The first requirement will be satisfied by demonstrating that the same system can be used in two different domains. In both scenarios, the system will be tested on both hard and easy regions in order to show its ability to control its own complexity. The phase transition phenomenon will be investigated, i.e., a model for phase transition (complexity as a function of the parameter characterizing the amount of change) will be developed and then used by the proposed system. The adaptability of the system will be tested by changing parameters in the constraints during the system operation. The definition of a parameter to measure the adaptability will be part of this research. Another aspect of the evaluation of the system is the quality of solutions to multi-objective optimization problems it finds. Towards this aim the solutions will be compared to known benchmark approaches.

Contents

List of Figures

1 Problem Statement

Multi-objective optimization (also referred to as “multi-criteria optimization”) is a core area in engineering, business practice and research. Application areas of multi-objective optimization include resource allocation, transportation, logistics, distribution, investment decisions, business planning with uncertain information, and others. Multi-objective optimization problems are formulated in terms of performance criteria (objective functions) and constraints. Two examples of such problems are design of a system and order negotiation in the manufacturing context. In design, one has to select components and a structure such that the components “fit together” while the resulting system delivers the desired performance, e.g., the system is 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 such as to satisfy goals of both the manufacturers and the customers. The manufacturer wants to maximize the price and lead times, while the customer is interested in the opposite. The constraints include manufacturing plant capabilities, storage capabilities and other.

To find a solution to a multi-objective optimization problem one needs to find an instantiation of the problem variables that satisfies the constraints and at the same time results in high values of the objective functions. Problems of this kind are known to be NP-hard and thus one cannot expect full algorithmic solutions. Moreover, since multiple performance criteria are involved, any solution needs to make tradeoffs among them. Consequently, one has to settle for less than optimality. In other words, a better approach can be first converting the original problem into a Constraint Satisfaction Problem (CSP) and then finding a solution to the new CSP problem. Conceptualization of this kind of problem formulation is known as *satisficing* solutions or *good-enough-soon-enough* solutions. Solutions of this kind of problems 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. Consequently, a special program needs to be developed for any specific problem/domain. What if the problem formulation changes? Then either a new program needs to be developed, or the original program needs to have built-in mechanisms for adapting to changes. The latter approach is appropriate when the changes in the constraints are parametric, while the former applies to non-parametric changes in constraints and objective functions.

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 [92]. The Unified Modeling Language (UML) is a combination of such abstract and graphical languages. The use of an abstract language like UML/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 this specific problem - how can a satisficing program be automatically synthesized from high-level specifications such that it can: (1) monitor and control its computational complexity, and (2) adapt to parametric changes in the constraints. In order to develop such a program we need to answer a number of questions. What is the language in which problem specifications can be expressed? What algorithm or system can be used to search for solutions to multi-criteria 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 and program adaptation?

2 Candidate Components of a Solution

The search for solutions starts with a short description of multi-objective optimization problems. The challenges in finding one solution that optimizes all objectives are discussed first. From optimization, we switch to *constraint satisfaction problems* (CSP) in general. An overview of existing CSP approaches are given. Since CSP algorithms are outside the scope of this thesis, we focus more on the CSP solvers, the evaluation of the performance of CSP solvers and the *phase transition* phenomena. Then, an introduction to the concept of satisficing, as an alternative to optimization, is given. Satisficing was originated from Herbert Simon's [106] 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 the satisficing approaches, a list of CSP solution methods and tools are given. Then, *self adaptive software* and negotiation are discussed since these research topics are related to the architecture we propose in this thesis. Finally, we compare our approach to the solution from the reviewed literature.

2.1 Multi-Objective Optimization Problems

Decision making for single-objective optimization has been well-studied. The problem becomes more difficult when one needs to consider several conflicting objectives. In many cases, it is unlikely that the 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 multi-objective or multi-criteria optimization problem (MOOP). Examples of multi-objective optimization were seen as early as in the nineteenth-century economics [24, 83].

A multi-objective 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, the multi-objective optimization problem is stated in its general form:

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

A solution x is a vector of N decision variables: $x = (x_1, x_2, \ldots, x_N)^T$. 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. 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. Multi-objective 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 multi-objective optimization problems are,

- *weighting objectives*,
- ϵ-constraint method,
- *goal programming*,
- *hierarchical optimization*,
- *global criterion*,
- *distance functions*,
- *min-max optimum*,
- and, *trade-off* methods [28, 25].

All of the classical methods listed above suggest a way to convert a multi-objective 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 multi-objective optimization methods use a concept called *domination*. In these methods, two solutions are compared on the basis of whether one dominates the other solution or not. In [20], domination is defined as the following:

Definition 2.1 (Better). *A solution x^2 is better than solution x^1 for a given objective function f_m, if*

- $f_m(x^1) \leq f_m(x^2)$, *when the objective is to maximize f_m,*
- $f_m(x^1) \geq f_m(x^2)$, *when the objective is to minimize f_m*

Definition 2.2 (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 2.3 (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 2.4 (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 $m = 1, 2, \dots, M$.*
2. *The solution x^1 is better than x^2 in at least one objective (or mathematically $\exists m' \in \{1, 2, \dots, M\}$ such that $f_{m'}(x^1) \triangleleft f_{m'}(x^2)$).*

Definition 2.5 (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 2.6 (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 [20].

However, as it is stated in [111, 41, 105], there are two major concerns or weaknesses in these methods. 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 emerged for exploring concepts of decision making that do not depend on the principle of optimality.

2.2 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 an assignment of a value to each variable from its domain such that all the constraints are satisfied. 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 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, such 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 format* (CNF), e.g. Equation 2, such that all clauses in boolean expressions are satisfied. A *clause* is a boolean sum of variables or their negations. A boolean expression in CNF is a product of many clauses.

$$(A \vee B \vee C) \wedge (\bar{C} \vee D \vee A) \wedge (\bar{D} \vee E) \tag{2}$$

The classical CSP framework has been introduced formally at the beginning of the 70's [74], and has been studied since ([76] gives a brief history of the studies on CSP). The general constraint satisfaction problem is of high-complexity (*NP*-complete or worse), which means that there is 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 ([37]); i.e., search in different parts of the space keeps failing for the same reasons. 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 [61] can be helpful to understand arc-consistency:

Definition 2.7 (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 2.8 (n-ary CSP). *An n-ary CSP is a CSP, in which each constraint is an n-ary constraint.*

Definition 2.9 (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 [93]. 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 2.10 (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 [30, 68, 74, 33, 21, 34, 125].

Later, a new research field emerged that focused on finding classes of constraint satisfaction problems where the preprocessing algorithms could find a solution by themselves. These classes of problems were referred to as "islands of tractability". Some identified classes are tree-structured problems, problems generated by graph grammars [75], and problems with a certain relationship between their graph structure and their level of consistency [22].

Sometimes a constraint satisfaction problem can be over-constrained. When there is no solution that will satisfy all constraints, some of the constraints could be relaxed to make the problem solvable. In such a case, the algorithm first needs to find that the problem is unsolvable, and then the algorithm needs 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) [10] 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 [76], Montanari and Rossi state that the constraints are dynamic in many real-life problems and there is a need for more interactive constraint satisfaction systems, which should both provide assistance in the modeling phase and support dynamic changes in the constraints.

2.2.1 Phase Transitions

Comparing the performance of different search algorithms is very important. Some of the 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 [60].

All these measures, however, are useful only if the algorithm terminates. For NP-complete problems 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* had been first introduced in statistical physics. From [49]: "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 [13], Cheeseman showed that for many NP-complete problems one or more "order parameters" can be defined, and 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 very 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.

Today, it is a normal approach to evaluate a proposed algorithm empirically on a set of randomly generated instances taken from the relatively "hard" *phase transition* region [103].

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 [87]. Phase transitions in constraint satisfaction problems were also studied by [85, 109, 38, 102, 50, 47, 48, 126]. At this point, the challenge is to identify an order parameter for a particular problem domain.

2.3 Satisficing

In [106, 107, 108], 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 to determine a threshold (aspiration level) that the payoff must exceed. Then, the payoff requirement would be represented as an additional inequality that needed to be satisfied. Aspiration level is 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 the determination of an aspiration level was criticized as arbitrary [112]. In [129], 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, while optimal meta-reasoning and bounded optimality were favored over other approaches.

Papers are beginning to emerge regarding the problems of satisficing multiple-agent decision making [104]. However, they are mainly interested in the algorithms, and little have been done to formalize a multi-agent satisficing concept.

2.3.1 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* [95] (or knowledge compilation as it is called in [101]), and *fuzzy logic.*

2.3.2 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 such a way as to maximize 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 [40], 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, hence they offer a tradeoff between resource consumption and output quality [18, 51, 19]. There are two types of anytime algorithms, interruptible and contract algorithms. 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 the contract algorithms the total time allocated for computation needs to be known in advance. Therefore, interruptible algorithms are more flexible than the contract algorithms. However, interruptible algorithms are more complicated to construct. First, the designer has to ensure the interruptibility of the composed system, or in other words, the designer has to ensure that the system as a whole can respond to immediate demands for output [97]. 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 [98], 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 been started to develop programming tools for composition and monitoring of anytime algorithms [128, 42, 78].

2.3.3 Bounded Optimality

A bounded optimal agent behaves as well as possible given its computational resources [96]. The following definitions should help better understanding 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 2.11 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 2.12 (Bounded optimality) *The bounded optimal program l_{opt} is defined as*

$$l_{opt} = argmax_{l \in L_M} V(Agent(l, M), \boldsymbol{E}, U) \tag{3}$$

In [99], 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 very hard to achieve. The approach described in [90] is one of many approaches that combine multilevel reasoning and bounded optimality.

2.3.4 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 [46]. (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 too 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 [112], Stirling and Goodrich 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 2.13 (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, and further elimination will be required before action can be taken.

2.4 CSP Solution Methods and Tools

2.4.1 Constraint Logic Programming

Application developers took advantage of the constraint solving methods and used constraint-related techniques successfully in applications like assignment problems, CAD, decision-making systems, graphics, network management, robotics, scheduling, typesetting, VLSI, and many others. This also led to the design and implementation of a number of constraint-based programming languages [16]. SKETCHPAD [115], CONSTRAINT [114], and ThingLab [8] 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 makes it easy to express constraints while its non-determinism removes the need for programming a search procedure. However, traditional logic programming languages (e.g., Prolog) could be very inefficient (i.e., trashing, repeated failure due to the same reason or having to do redundant work during backtracking) due to their passive use of constraints to test potential values instead of reducing the search space actively [124]. Also, defining rich data structures and operations on these structures were not possible. Earlier constraint logic programming (CLP) languages like CHIP [120], CLP($\Re$) [53], Prolog II, and Prolog III tried to preserve the advantages of logic programming without being affected by its limitations. Later, the CLP scheme [52] generalized the fundamental idea behind these constraint logic programming languages. The CLP scheme defined a family of programming languages based on their semantic properties. The CLP scheme could be instantiated to produce a specific constraint logic programming language by defining a constraint system. The CLP scheme was later generalized into the `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 [14], are Eclipse [26], OZ [100, 110, 71], CIAO [45], AKL [43], Prolog IV [15], HAL [23] and Salsa [62]. Their constraint vocabulary and solvers perform beyond traditional linear and non-linear constraints and support logical and global constraints. OZ, CIAO, and AKL used concurrent constraint (CC) framework and implemented distributed and concurrent systems. However, these languages are mostly targeting the computer scientists and have weaker abstractions for algebraic and set manipulation.

Helios language [121] and Numerica [122] have been designed to solve non-linear constraint systems using interval analysis techniques. CLP toolkits, like QOCA [70], EaCL [118] and Ultraviolet [9], implemented graphical user interfaces to monitor the progress of the constraint solver and provided the user with a mechanism to interact with the solver at run-time.

2.4.2 Modelling Languages for CSP

Mathematical modelling languages are another kind of tools used in optimization. Modelling languages like AMPL [31], GAMS [4], Claire [65, 66], CML [5], VISUAL SOLVER [123] provide high-level algebraic and set notations to express mathematical problems that can then be solved using the solvers presented above. In the case of CML, the models written by this language were later translated to CHIP. There are also a new set of optimization programming languages, like OPL [7], XPRESS-MP, Modeler++ [73] and Salsa, which aim to unify modelling and constraint programming languages.

Negotiation [35], machine learning [94] and constraint query languages [56, 12], where CLP and Database technologies were integrated, are other techniques that have been applied to constraint problems to help the user to model the system.

2.5 Active Software

In [88], 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, *self adaptive software* (see Section 2.5.1), *negotiated coordination* (see Section 2.5.2), *tolerant software*, and *physically grounded software*. 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.

All these technologies incorporate the knowledge of requirements, designs, structure, I/O sources in the running software and can be used together.

2.5.1 Self Adaptive Software

Self-adaptive software is defined as [63]: "Self-adaptive software evaluates its own behavior and changes behavior 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, (2) a collection of alternative implementations and algorithms.

Early research in self-adaptive software was concentrated on three paradigms: *dynamic planning systems*, *self-controlling systems*, and *self-aware systems*. Dynamic planning systems first plan their actions [39, 79]. After evaluating and confirming the effectiveness of their actions, they start the execution. Planning includes scheduling and configuration of resources like hardware, communication capacity, and software components.

In self-controlling systems [58, 59, 27, 36, 6] (see Section 4 for details), the runtime software behaves like a plant, with inputs and outputs monitored and controlled by separate monitoring and controlling units. Self-controlling systems support three levels of control: *feedback*, *adaptation*, and *reconfiguration*.

In a self-aware software [117, 57, 11, 91], the key factor is self-modeling approach. The application is built to contain knowledge of its operation, and it uses this knowledge to evaluate performance, to configure and to adapt to changing circumstances.

In [64], Laddaga points at the following problems and unsolved issues in the existing self-adaptive work:

- Evaluation of the functionality and the performance at runtime [72]
- Dynamism and software architecture representations for self-adaptive software: more introspective languages, better debug-ability, better process descriptions, better structural descriptions [84]
- Runtime performance while evaluating outcomes of computations and determining if expectations are met.

- The effort that takes to create software capable of evaluating and reconfiguring itself
- The lack of adequate metrics for degree of robustness and adaptation.
- Advances in computer hardware.

Two of the implementations of self-adaptive software are SAFER [1] and ASC [67].

2.5.2 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 [86].*

Definition 2.14 *Negotiated coordination is the coordination of independent software entities via mutual rational agreement on exchange conditions [88].*

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

Negotiated coordination is another approach covered under the active software paradigm. Government and research organizations have a number of programs to support the research for negotiated coordination in software. One of these programs is called the ANTs (Autonomous Negotiating Teams) program and sponsored by AFRL and DARPA. The objective of ANTs program [3] 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 started:

- CAMERA, stands for Coordination and Management of Environments for Responsive Agents, is 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 work collectively.
- ATTEND (Analytical Tools To Evaluate Negotiation Difficulty) project 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, complexity reduction via phase-transition aware partitioning of task space. Management of resource contention facilitated by SAT encoding of complex allocation problems.
- MARBLES approach [32] is a definition and comparison of cooperative negotiation schemes for distributed resource allocation. It assigns values (or prices) to the tasks and tasks with higher values are allocated with resources first.
- DEALMAKER utilized agents that select the best sources of supply to fill orders [116]. It used a flexible XML-based representation for the contracts and had an interactive user interface to enter contracts online and enter rules to govern the contracts.

- The MICANTS, stands for Model Integrated Computing and Autonomous Negotiating Teams for Autonomous Logistics, project is executed by Vanderbilt University. The project seeks to develop efficient negotiation protocols for distributed problem solving in the logistics domain.
- MAPLANT (Maintenance Planning Tool) project [119] is a part of MICANTS project. The main goal in the project is to create a schedule for airplane maintenance activity. First, scheduling problem was transformed to a finite domain constraint problem and then the CSP code was manually written in OZ. Input data is provided in XML format. Like DEALMAKER, MAPLANT has a graphical user interface that helps the user interact with the search process.
- ADEPT multi-agent system [54]: The motivation behind the Advanced Decision Environment for Process Tasks (ADEPT) project is that an agent based approach should be suitable for implementing systems to manage business processes. The ADEPT system and agent architectures are designed to ensure maximum flexibility to adapt as the business process changes. ADEPT provides a method for designing agent oriented business process management system, and agent implementation that is 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 [55].

Other current research on negotiation can be found [113, 35, 127, 17, 77].

2.6 Negotiation Models

2.6.1 Bilateral Negotiation Model

The model, discussed in this section, is the *two parties, multiple issues* value scoring model defined in [89]. That is a model for bilateral negotiations about a set of quantitative variables. A *negotiation thread* is a sequence of offers and counter-offers in a two-party negotiation. 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 (i.e., time, resources). To achieve flexibility in negotiation, agents may wish to change their ratings of the importance of the different criteria, therefore their tactics vary over time. *Strategy* is the term used to denote the way in which 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 $V_j^i : [min_j, max_j] \to [0, 1]$ 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:

$$V^i(x) = \sum_{1 \le j \le n} w^i_j V^i_j(x_j) \tag{4}$$

2.6.2 A Service-oriented Negotiation Model

In service-oriented negotiations, the agents can 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 $x_j \le y_j$, then $V^s_j(x_j) \le V^s_j(y_j) \Longleftrightarrow V^c_j(y_j) \le V^c_j(x_j)$.

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, $x^t_{a \to b}$ represents the value of the offer proposed by agent a to agent b, and $x^t_{a \to b}[j]$ represents the value of issue j proposed from a to b at time t.

Definition 2.15 (Negotiation thread). *A negotiation thread between agents $a, b \in$ Agents, at time $t \in$ Time, noted $x^t_{a \to b}$ or $x^t_{b \to a}$, is any finite sequence of the form $\{x^{t_1}_{d_1 \to e_1}, x^{t_2}_{d_2 \to e_2}, \cdots, x^{t_n}_{d_n \to e_n}\}$ where:*

1. *$e_i = d_{i+1}$, proposals alternate between both agents,*
2. *$t_k \le t_l$ if $k \le l$, ordered over time,*
3. *$d_i, e_i \in \{a, b\}$, the thread contains only proposals between agents a and b,*
4. *$d_i \ne e_i$, the proposals are between agents, and*
5. *$x^{t_i}_{d_i \to e_i} \in [min^{d_i}_j, max^{d_i}_j]$ or is one of $\{accept, reject\}$.*

Assume $x^{t'}_{b \to a}$ 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 2.16 (Offer). *The interpretation by agent a of an offer $x^t_{b \to a}$ sent at time $t < t'$, can be formalized as follows:*

$$I^a(t', x^t_{b \to a}) = \begin{cases} reject, & \text{If } t' \ge t^a_{max} \\ accept, & \text{If } V^a(x^t_{b \to a}) \ge V^a(x^{t'}_{a \to b}) \\ x^{t'}_{b \to a}, & otherwise \end{cases} \tag{5}$$

In order to prepare a counter offer the following families of tactics are defined [29]:

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 the 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 in which the agent is not under a great pressure to reach an agreement, it may choose to use imitative tactics that protect it from being exploited by other agents. In this case the counter offer depends on the behavior of the negotiation opponent.

3 Analysis of Candidate Components

The kind of problems that we try to solve in this thesis are multi-objective 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 was the conversion of a multi-objective optimization problem into a single-objective optimization problem by combining all the objective functions into one. This was described in Section 2.1. An example of the combination of the functions into one is assigning weights to each objective and then adding the weighted values of the objective functions. Therefore, this combination method is based on some additional information about the problem. Other methods, similarly as this one, also use some additional information about the problem. Our problem formulation described in Section 1 does not include this kind of information. We do not assume that this kind of information is available. Consequently, this approach cannot be used in the solution to our problem.

The next candidate is the satisficing approach. However, as we stated in Section 2.3, this approach is based on the assumption that an "aspiration level" for each objective is known. Again, we don't have this kind of an assumption in our problem formulation. Therefore, a straight forward application of the satisficing approach cannot be used. Instead, we propose to establish aspiration levels dynamically, using negotiated coordination 2.5.2. To achieve this, we propose to use an agent based approach in which each objective will be represented by an agent and then the agents will negotiate solutions. Each agent will use a separate CSP solver to find a solution that will satisfice its objective (cf. [54]). Like the MAPLANT project [119], we will transform each objective and the related constraints to a separate constraint satisfaction problem and use a CSP solver to solve the problem. The aspiration level will be an explicit parameter in our system. The value of the aspiration level achieved in one negotiation will be used as an initial value in the next negotiation.

The ATTEND project [2] also utilizes the satisficing approach. Their system monitors the performance of the negotiation to avoid phase transition regions. However, the problem domain for the ATTEND project was 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, we use specialized components to monitor the performance of the CSP solvers and to avoid phase transition regions. But, we do not limit our problem domain to SAT and thus we cannot use the same kind of invariants. Instead, one of our research goals is to find invariants specific to our problem domain and then formulate a more general pattern for invariants applicable to any multi-objective optimization problem.

The architecture that we use for the software agents is a metalevel architecture (see Section 2.3.2). This architecture is called *Self-Controlling Software Architecture* [58]. In its full implementation it includes three metalevels (also called *loops*): the feedback loop, the adaptation loop and the re-configuration loop. In this research we use only the first two metalevels. The two metalevel loops are implemented in each agent.

One of our goals stated in Section 1 was to choose a language for specifying multi-objective optimization problems. One possibility for us was to use CML [5]. For this language, there exist translation rules that could be used for translating a model expressed in CML to the constraint logic programming language CHIP [120]. The expressiveness of this language is, however, limited; it is not possible to express structural aspects of the user's view of the problem formulation. Since this feature is needed for our system, we have to use a more expressive language. We propose to use UML/OCL for this purpose. Our solution further extends 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 to change the target CSP solver with minimum effort.

Other modeling language candidates might include Claire [65, 66] and VISUAL SOLVER [123]. 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 us to utilize off-the-shelf products like Rational Rose, Rhapsody or any graphical UML tool.

4 Proposed Solution

To achieve the goals outlined in the previous sections, we will implement an experimental framework (see Figure 1) consisting of the solution elements described in Section 3. The framework will include a set of tools that will automate the synthesis of a satisficing program from specifications such that the program can: (1) monitor and control its complexity, (2) adapt to parametric changes in the constraints.

The following tools will be implemented:

- An ontology for multi-objective optimization. The ontology will be implemented in UML/OCL and will support the user in the specification of optimization problems (see Appendix A and B for details).
- 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 D for details).
- Code generator translation rules for the Oz target CSP language.
- Templates for creating software agents.
- An instantiation of the Self-Controlling Software Architecture (SCSA). This part will be semi-automatic. A template of SCSA will be developed and used to integrate the agents. Algorithms for each component of SCSA will be developed. Simple negotiation algorithms will be developed. For each instantiation the templates will have to be customized and then compiled. See Appendix C for details.
- Phase transition invariants will be defined. Experiments will be performed to assess the appropriateness of the invariants for defining phase transition regions. The invariants will be used in the instantiation of the SCSA described above (see Appendix G.2 for an experiment on phase transition invariants).

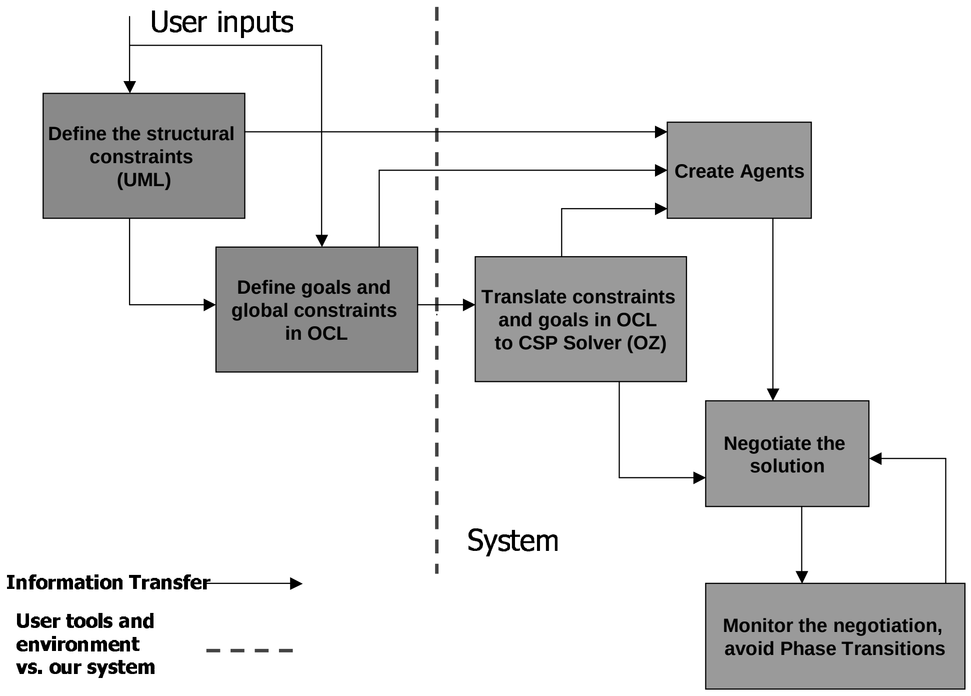
User inputs
Define the structural constraints (UML)
Create Agents
Define goals and global constraints in OCL
Translate constraints and goals in OCL to CSP Solver (OZ)
Negotiate the solution
System
Information Transfer
User tools and environment vs. our system
Monitor the negotiation, avoid Phase Transitions

Figure 1: Overview of the proposed solution

5 Contributions

Phase transition invariants as indications of complexity of multi-objective optimization

Phase transition invariants have been used as a measure of complexity of Constraint Satisfaction Problems. We will show that phase transition invariants can also be used as indicators of complexity in multi-objective optimization problems at search time. We will demonstrate that a synthesized CSP program can control its complexity by monitoring the search for a satisficing solution with respect to these phase transition invariants at execution time. We will investigate the phase transition phenomenon to specify a method to identify phase transition invariants and the related computation intensive critical regions for a given CSP domain. We will specify methods (relaxing constraints, changing parameters in constraints) for the synthesized CSP programs to avoid these critical regions.
bf Objective based agentification of multi-objective optimization problems

One of the proposed solution components of our problem will be using agents to establish aspiration levels of particular objective functions. We will show that this can be solved by creating one agent for each objective function and then using negotiation to come with the values of particular aspiration levels. We will develop a method to create such software agents. We will show that mapping multi-objective optimization problems to an this kind of agent-based architecture can help avoiding computation intensive critical regions, while still achieving good results for each of the optimization criteria.

Self-Controlling Software Architecture as a way of controlling complexity of CSP

One of the main objective of this thesis will be to show that the Self-Controlling Software Architecture can be used to both control the complexity of computation and to adapt to parametric changes in the specifications of multi-objective optimization problems. The SCSA will use its feedback loop mechanisms to monitor the phase transition invariants as the search for a satisficing solution progresses, as well as to control the agent's negotiation so that such computation intensive regions are avoided. The SCSA will use its adaptation loop to control the appropriateness of a specific controller to a given version (parameters) of the multi-objective optimization problem.

Ontology-based support of specification of multi-objective optimization problems

One of our goals is to replace the programming in a CSP language with a more user-friendly development of a specification in a graphical language. Towards this goal, we propose to use UML, a graphics oriented specification language. However, the graphical part of UML is not sufficient to express many constraints. Therefore, we propose to use OCL, the UML's constraint language that can be used to extend the capabilities of UML. However, since OCL is a constraint language, the development of specifications in this language is a tedious process. Therefore, to make the use of OCL more user-friendly we will propose an ontology that will support the specifier in the process of developing specifications of objective functions and constraints of multi-objective optimization problems. The ontology itself will be expressed in OCL. It will contain a selection of problems typically encoutered in the specification of muli-objective optimization problems. Additionally, the ontology will also include constructs to support the generation of software agents.

Proof-of-concept of automatic CSP code generation for multi-objective optimization problems

We will show a proof-of-concept of automatic CSP code generation for multi-objective optimization problems. We will specify a generic method that will support the generation of different target CSP programming languages and we will demonstrate the execution of the code generator by using OZ as the target CSP programming language.

6 Method Verification and Demonstration

To evaluate the approach, an experimental system will be implemented and tested on two scenarios - a fixture design scenario and an order negotiation scenario. Problem formulations will be specified using the UML/OCL representation. The formulations will be automatically translated to the selected Constraint Satisfaction Problem (CSP) language and then used by the system to find satisficing solutions. The goal will be to provide a proof-of-concept of developing such a self-controlling satisficing program that (1) is applicable to various multi-objective optimization problems, (2) has the ability to control its own complexity, and (3) can adapt to parametric changes in the constraints. The first requirement will be satisfied by demonstrating that the same system can be used in two different domains. In both scenarios, the system will be tested on both hard and easy regions in order to show its ability to control its own complexity. The phase transition phenomenon will be investigated, i.e., a model for phase transition (complexity as a function of the parameter characterizing the amount of change) will be developed and then used by the proposed system. The adaptability of the system will be tested by changing parameters in the constraints during the system operation. The definition of a parameter to measure the adaptability will be part of this research. Another aspect of the evaluation of the system is the quality of the solutions to the multi-objective optimization problems it finds. Towards this aim the solutions will be compared to known benchmark approaches.

6.1 Fixture Design Problem

The first experimental multi-objective optimization scenario 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) (see Section G for details). 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).

To formalize the objective functions and the constraint we need to introduce the following notation.

- P - the set of all UUT pins, $p \in P$
- 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.
- $Cont : C \rightarrow CC$ - the containment function. It maps a channel c to its container channel card, $cc \in CC$.
- $ContU : 2^C \rightarrow 2^{CC}$ - 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, $r \in R$.

- $Rec : P \rightarrow 2^R$ - 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.
- $Cap : C \rightarrow 2^{CAP}$ - the capability function that assigns a set of capabilities to each channel c.
- $PC : P \rightarrow C \cup \{c_{null}\}$ - channel assignment function. It assigns a channel to a UUT pin. This is an optimization variable.
- $f_{map} : R \rightarrow CAP$ - a function that maps the pin requirements to channel capabilities. The function is $m - to - 1$.
- $card : Set \rightarrow N$ - 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 number of UUT pins that are NOT assigned to any channel.

$$\begin{aligned} \text{Minimize} \quad & N_{PU} = card(PC^{-1}(c_{null})) && (6) \\ \text{s.t.} \quad & \forall p, p' \in P : \quad p \neq p' \Rightarrow PC(p) \neq PC(p') && (7) \\ & \forall p \in P, \forall r \in Rec(p) : \quad f_{map}(r) \in Cap(PC(p)) && (8) \end{aligned}$$

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

$$\begin{aligned} \text{Minimize} \quad & N_{CC} = card(ContU(C)) && (9) \\ \text{s.t.} \quad & N_{CC} \leq N_{CC_{MAX}} && (10) \\ & card(PC(P) - c_{null}) \leq N_P && (11) \end{aligned}$$

A complete listing of OCL representation of the functions, the constraints and the objective functions can be found in Appendix G.3.

6.1.1 Problem-Specific Metrics

For the Fixture Design experiment, brute force search will be used as the baseline algorithm. The metrics that will be used in the comparison are:

- Computation time vs. pin requirements (and constraints) change
- Response time for getting an answer that a solution exists (there is a fixture mapping between this UUT and the tester configuration that satisfies all the pin requirements)

- Percentage of UUT pins matched as pin requirements change

The following criteria will be used to determine success:

- Computation time and response time will be less than the baseline algorithm.
- The percentage of UUT pins matched will be higher if a 100% match is impossible.

6.2 Order Negotiation System

The second experimental system is an order negotiation system for a marketing or sales department of a manufacturing company. The agents of the manufacturer and the customers have different objectives. The objective of the manufacturer agents is to fill the orders coming from the customers according to the profitability and the constraints forced by the labor and raw material levels. The objective of the customer agents is to minimize the purchase price of a product. The constraints of the customer are about the price, the quantity, and the delivery time of the product. Appendix H specifies the objective functions for both the manufacturer and customer agents.

For the order negotiation experiment, linear programming will be used as the baseline algorithm. The metrics that will be used in the comparison are:

- Average negotiation time (whether an agreement is reached or not-time can be replaced by the number of offers)
- Percentage of negotiations ended with an agreement
- Percentage of change in the customers offer vs. percentage of change in the manufacturers offer:

$$\frac{\Delta X_C}{\Delta X_M} = \frac{|X_C^{t_f} - X_C^{t_0}|}{|X_M^{t_f} - X_M^{t_0}|} \tag{12}$$

The following criteria will be used to determine success:

- Average negotiation time will be shorter.
- Percentage of negotiations ended with agreement will be higher.

7 Plan

7.1 Completed Tasks

- Formalization of a software agent described (class definitions, interfaces) in UML
- Implementation of the OCL parser
- Design of the schema for OCLML
- Design of the software agents
- Design of the negotiation algorithm
- Formalization of the Fixture Design Experiment (goals, constraints)
- Formalization of the Order Negotiation Experiment (goals and constraints)

7.2 Schedule for remaining tasks

Task	**Effort(days)**	**Complete by**
Complete the proposal presentation	5	Sep, 12
Implement the generic code generator	5	Sep, 19
Specify the ontology for expressing goals and constraints	10	Oct, 3
Implement the translation rules file (XSL file) for OZ	20	Oct, 31
Implement the template code for agents in OZ	15	Nov, 21
Implement the QoS module for agents in OZ	10	Dec, 5
Express the goals and constraints of Fixture Design experiment in OCL	2	Dec, 8
Identify the phase transition invariants and regions for Fixture Design experiment	10	Dec, 23
Implement the benchmark program for Fixture Design experiment	10	Jan, 6
Execute the Fixture Design experiment	5	Jan, 13
Express the goals and constraints of Order Negotiation experiment in OCL	2	Jan, 15
Identify the phase transition invariants and regions for Order Negotiation experiment	10	Jan, 29
Implement the benchmark program for Order Negotiation experiment	10	Feb, 12
Execute the Order Negotiation experiment	5	Feb, 19
Analyze the results of the experiments	10	Mar, 4
Analyze and specify the methodology used in the determination of phase transition invariants	20	Apr, 1
Update the Literature Review section of the thesis	5	Apr, 8
Write the remaining sections of the thesis	15	Apr, 29
Send the thesis for review and incorporate the feedback	10	May, 13

A Defining Structural Constraints in UML

Domain specific knowledge about the problem area is expressed in this step. Class Diagrams, as defined in the Unified Modelling Language (UML) Specification [82], 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 [81]. 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) [80] objects.

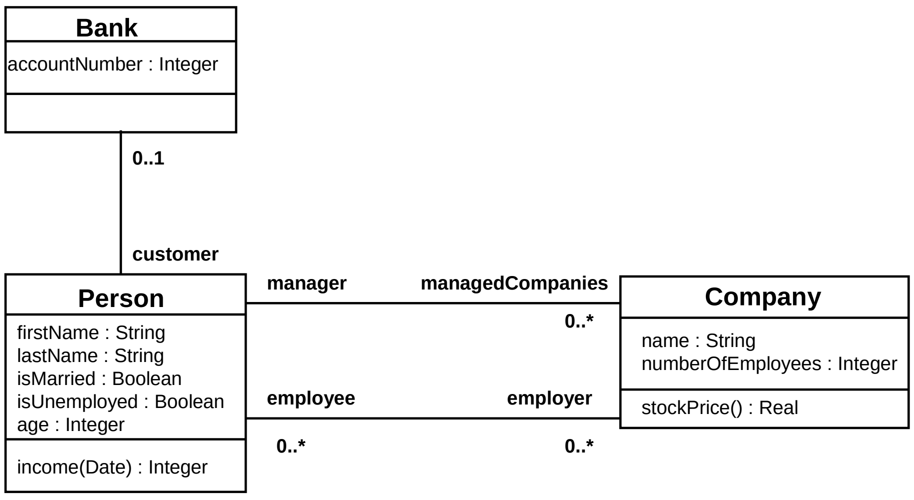

Figure 2: UML Class diagram for bank example

The following is a part of XMI file that is generated for the class diagram in Figure 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 Translator (see Appendix D). 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. Defining an action language is outside the scope of this thesis. Therefore, if we need to express state actions in any experiments, we will directly program in the target language.

B Defining Goals and Constraints in OCL

As it is described in the Object Constraint Language (OCL) Specification, "In object-oriented modelling, 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". We have used the March 2003 version of the OCL Specification which is a part of UML version 1.5. We considered to use OCL for a number of different purposes:

- 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

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

In this thesis, we will extend the pre-defined types and operations in OCL to cover the concepts like,

- Minimize, maximize functions
- Templates for offer and counter-offer concepts in negotiation
- Phase Transition invariants
- Phase Transition boundaries

C Architecture for software agents

In the proposed solution, a separate *software agent* represents each criteria in the multi-objective optimization problem. "Agent" is a theoretical concept from AI. The concept agent is used as Maes has defined in [69]: "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.

The architecture that is used in the software agents is a variation of the self-controlling software model. Self-controlling software model [58] 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 with rules for how its 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 systems' state. Self-controlling software identifies measurable inputs to the plant and classifies them as *control inputs*, which control the plant's behavior, or *disturbances*, which alter the plant's behavior

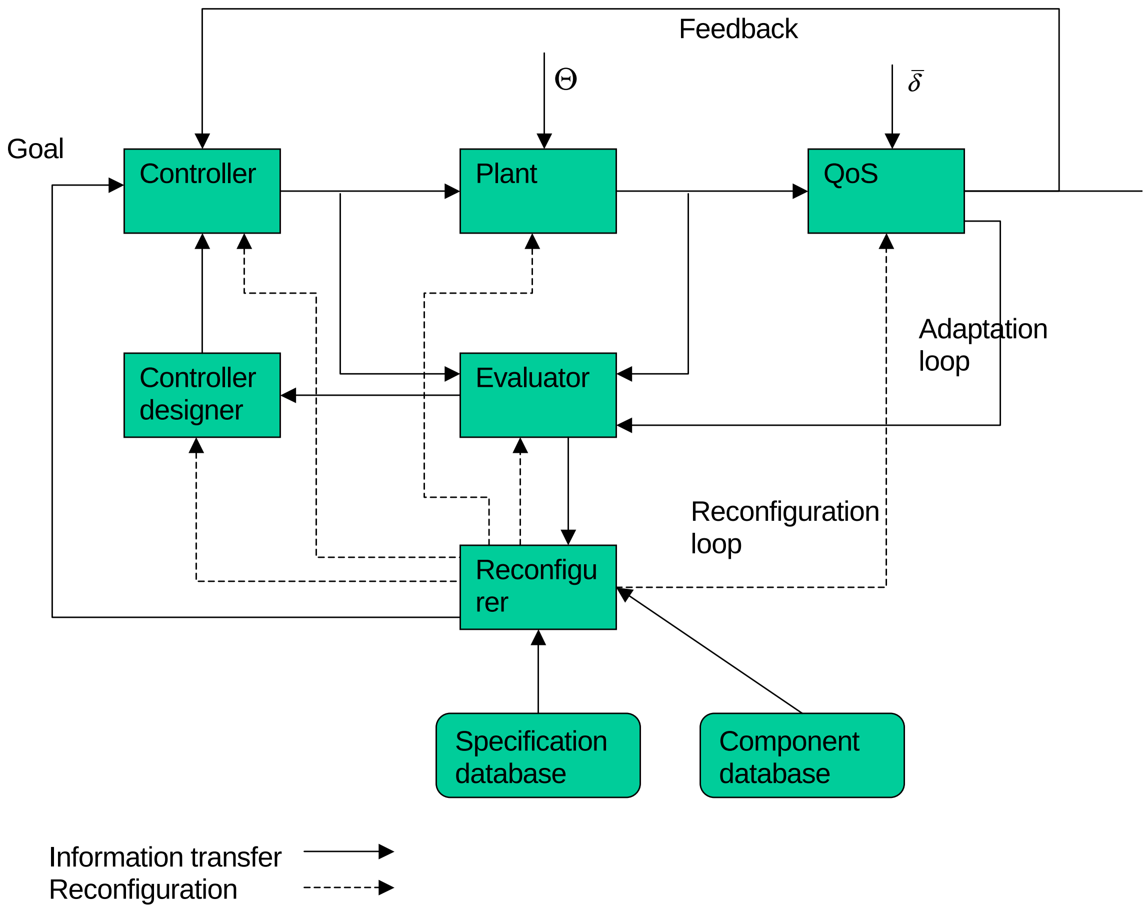

Figure 3: Self-controlling software model

unpredictably. Self-controlling software 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. Self-controlling software model includes the following three loops, each of which represents a different timescale for control activity (see Figure 3):

Feedback loop The controller sets parameters for the plant based on the goal and feedback received from the QoS subsystem.

Adaptation loop The evaluator evaluates the behavior and performance to determine whether the plants model is appropriate.

Reconfiguration loop It involves structural changes in the plant model, QoS subsystem, evaluator, controller, controller designer, goal, or even plant. *Reconfigurer* stays unchanged. Uses a specification database for decision making and a component database to assemble various system elements.

The premise behind mapping the concepts of control theory to software engineering is to use the concepts and tools developed in control theory – for example, *controllability*, *stability*, and *sensitivity analysis* (see Appendix C for a detailed description of the architecture used for the software agents in the proposed system).

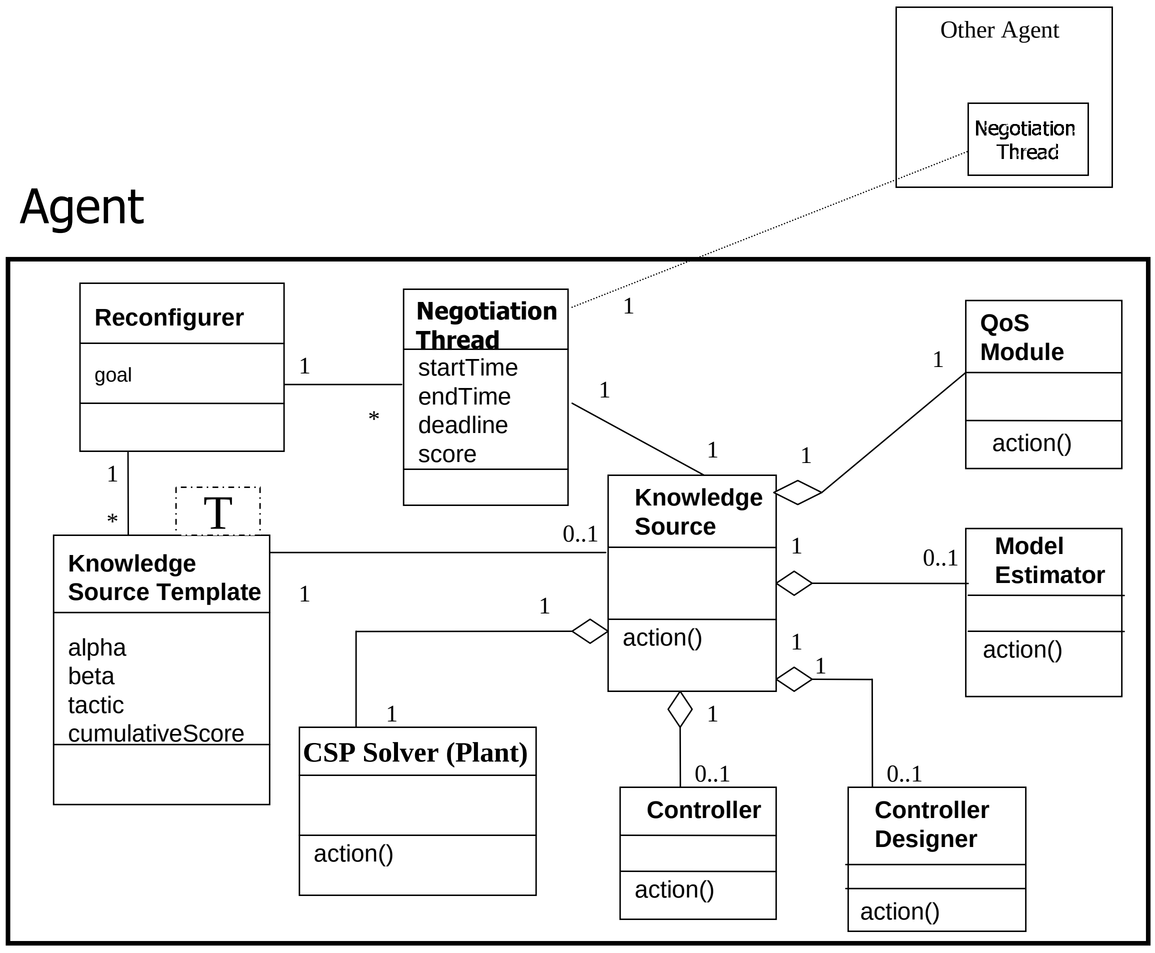

Figure 4: Agent Architecture

A software agent in our proposed system contains the following components (see Figure 4):

Reconfigurer It maintains the goal or optimization criteria. The reconfigurer instantiates and schedules a *Knowledge Source*(KS) when the agent needs to process an input or to communicate with another agent. The reconfigurer uses *Knowledge Source Templates* to instantiate new KSs. In addition to the creation of KSs, the reconfigurer instantiates *Negotiation Threads* between the current agent and other agents. After the negotiation is completed, the reconfigurer records the negotiation results and updates the algorithmic parameters of KS Templates according to their performance.

Knowledge Sources Knowledge Source (KS) is an AI concept used in Blackboard Architectures [44]. A KS perform the main functionality of the agent. An agent may utilize many algorithms to solve the same problem, and different algorithms are contained in different KSs. Adaptation loop occurs in the KSs. Each KS contains the following sub-components:

- **Plant:** The constraints that are translated to the language supported by the selected CSP Solver (in our experiments OZ) is executed in this component.
- **Quality of Service (QoS) Module:** QoS Module monitors phase transition invariants and detects the cases, where the values of these invariants fall into the computation intensive critical regions. Once such a region is entered the agent relaxes both performance criteria and constraints through negotiation. Phase transition order parameters (or invariants) will be determined (see Section 2.2.1 for brief explanation) and critical regions will be calculated empirically at design time.
- **Controller:** (Optional) Same as the Controller in self-controlling software model.
- **Controller Designer:** (Optional) Same as Controller Designer in self-controlling software model.
- **Model Estimator** (Optional) Same as Evaluator in self-controlling software model.

Negotiation Thread A negotiation thread supports and monitors the negotiation and the communication between the agent it belongs to and the negotiation thread of another agent. Each agent creates and commits its own negotiation thread object for the communication. The negotiation threads can be used as a communication link or interface between different platforms.

Knowledge Source Templates These templates either hold different algorithms or the same algorithms with different parameters. They are used to create new KSs. They are stored by the reconfigurer. These templates serve as the *specification database* in self-controlling software model.

Negotiation is used to determine the aspiration levels of the objective functions of each software agent. Specifically, the issues negotiated are the decision variables of the optimization problems (see Appendix E for the details of the mechanism). During negotiation, the decision variables cannot take a value outside their feasible solution set. The goal of the negotiation is to find a solution from the non-dominated set as it is defined in Definition 2.5.

D Translation of goals and constraints from UML / OCL to OZ

Translation of goals and constraints to OZ (or any other CSP programming language) is done in two steps (See Figure 6. 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 we will call OCL Markup Language (OCLML). The following is an example for a constraint written in OCL:

```
package Bank

context c:Company
    inv: c.numberOfEmployees > 50

endpackage
```

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>
```

C:\Yonet\OCLParser\OCLTranslator\OCLML.xsd 05/28/03 21:53:27

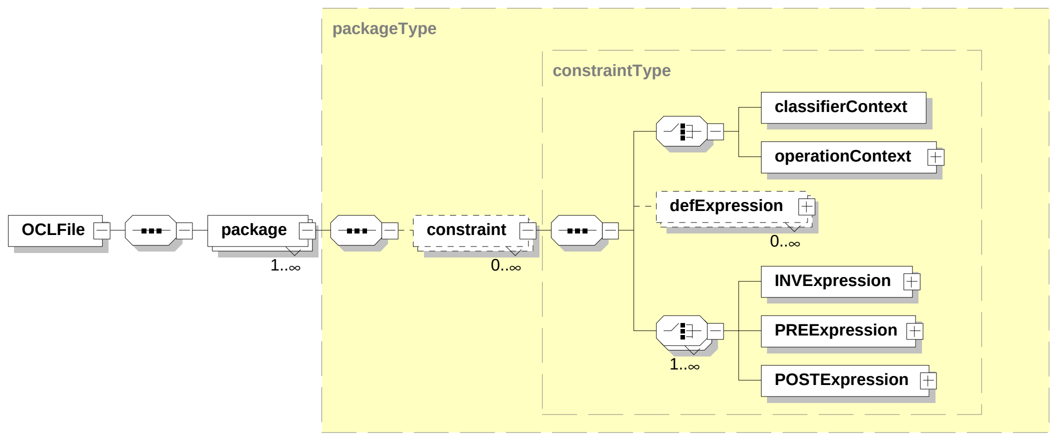

Figure 5: Schema for OCLML

A schema is provided to validate the intermediate files in OCLML (See Figure 5). The complete schema for OCLML can be found at Appendix F. After the constraints that are expressed in OCL are parsed, the generic CSP code generator translates the constraints in XML form to the target CSP programming language using a translation rules file for that language.

The separation of the parser from the code generator and the use of translation rules file makes the OCL Translator flexible enough to generate CSP code for different target CSP programming languages. In this thesis, for the demonstration, the translation rules file for OZ will be provided.

In this thesis, we selected OZ as the target CSP programming language and CSP solver, since it;

- Supports concurrency.
- Has an open architecture.
- Supports integration with other programs written in different programming languages, e.g. C, C++.
- Provides a run-time debugging environment.
- Provides a graphical user interface to monitor the progress of execution.

E Negotiation Algorithm

Negotiation mainly involves the Reconfigurer and the Negotiation Thread objects of an agent. The agent can be regarded as a service provider. And, every new input to

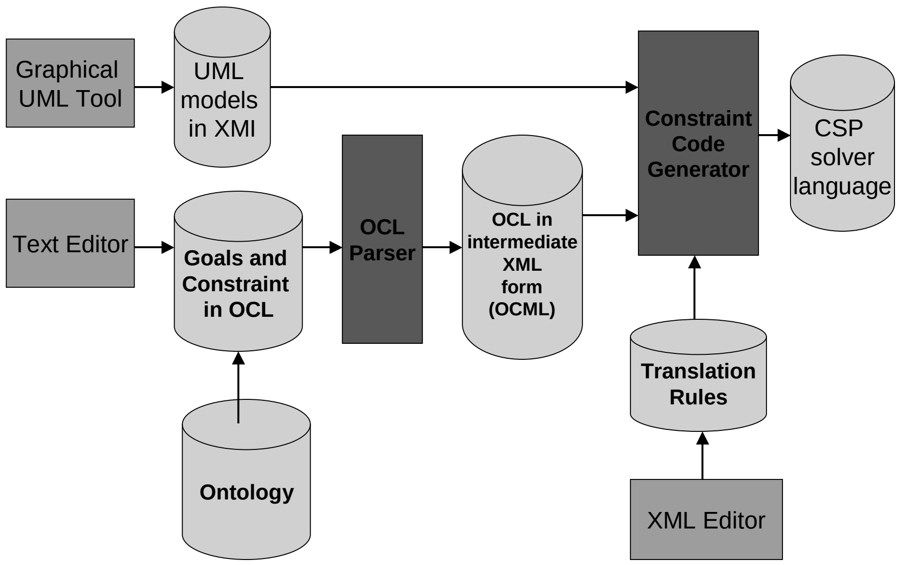

Figure 6: Translation of UML/OCL to the CSP programming language

the agent is passed as a service request, which the agent negotiates on a number of issues listed in the service request with the external client (can be the user or another agent) before fulfilling the request. In the case of multi-criteria optimization problem, the issues negotiated are the decision variables of the optimization problem.

In Figure 7, the state transition diagram for the Reconfigurer is given. Each time a new optimization problem is introduced to the agent as a request, the Reconfigurer creates a Negotiation Thread to do the negotiation and a Knowledge Source to solve the problem. From that point on, the Negotiation Thread provides the communication link between the Knowledge Source and the Negotiation Thread of the other agent (see Figure 8 for the State Transition Diagram for the Negotiation Thread).

F Schema for OCLML

```
<?xml version="1.0" encoding="UTF-8"?> <!-- edited with XML Spy
v4.1 U (http://www.xmlspy.com) by Yonet Arif Eracar (private) -->
<!--W3C Schema generated by XML Spy v4.1 U
(http://www.xmlspy.com)--> <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
    <xs:element name="OCLFile">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="package" type="packageType"
                    maxOccurs="unbounded"/>
```

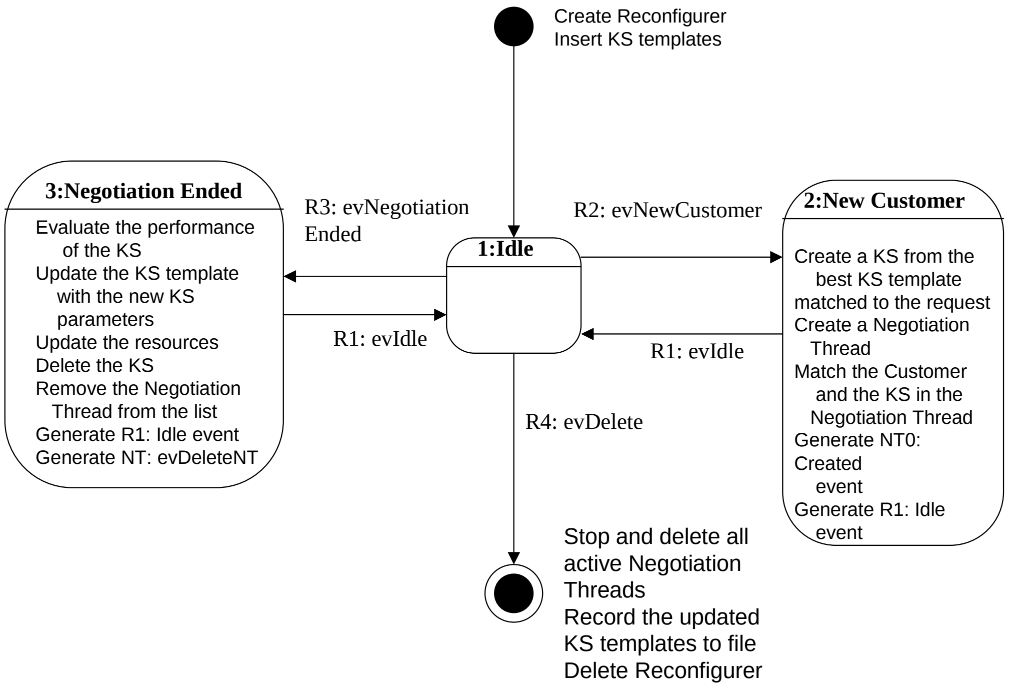
Create Reconfigurer
Insert KS templates
3:Negotiation Ended
Evaluate the performance of the KS
Update the KS template with the new KS parameters
Update the resources
Delete the KS
Remove the Negotiation Thread from the list
Generate R1: Idle event
Generate NT: evDeleteNT
R3: evNegotiation Ended
R1: evIdle
1:Idle
R2: evNewCustomer
R1: evIdle
R4: evDelete
2:New Customer
Create a KS from the best KS template matched to the request
Create a Negotiation Thread
Match the Customer and the KS in the Negotiation Thread
Generate NT0: Created event
Generate R1: Idle event
Stop and delete all active Negotiation Threads
Record the updated KS templates to file
Delete Reconfigurer

Figure 7: State transition diagram for Reconfigurer

```
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:complexType name="packageType">
    <xs:sequence>
        <xs:element name="constraint" type="constraintType"
            minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="Value" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="constraintType">
    <xs:sequence>
        <xs:element name="contextDeclaration">
            <xs:complexType>
                <xs:choice>
                    <xs:element name="classifierContext"
                        type="classifierContextType"/>
                    <xs:element name="operationContext"
                        type="operationContextType"/>
                </xs:choice>
            </xs:complexType>
```

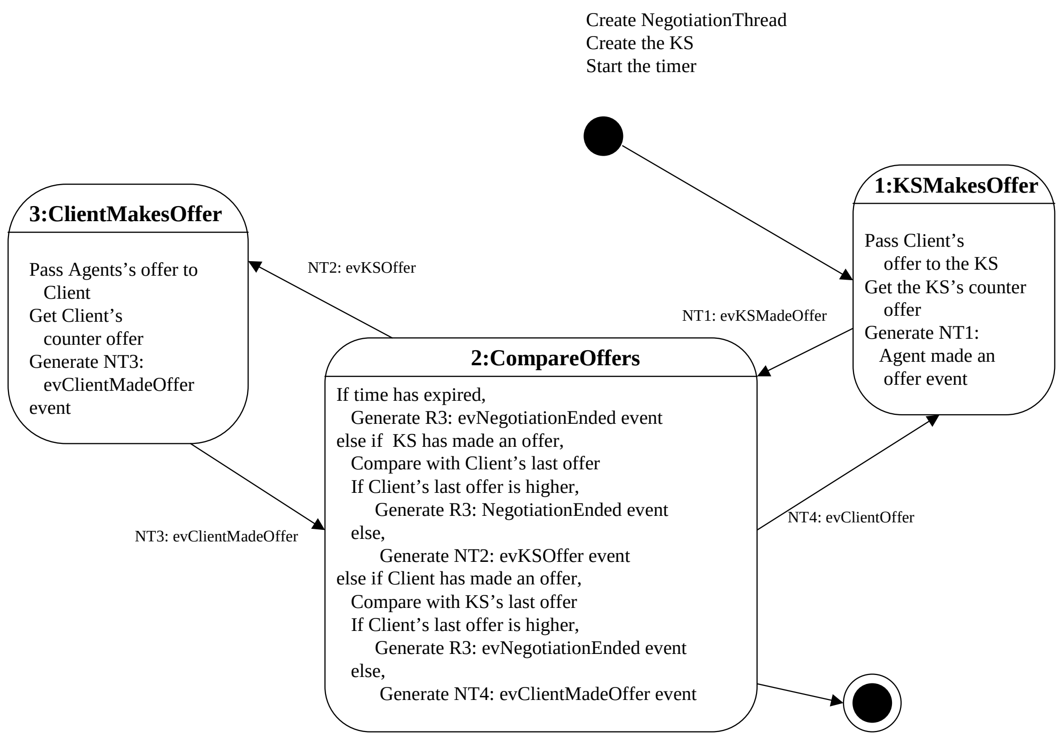
Create NegotiationThread
Create the KS
Start the timer
1:KSMakesOffer
Pass Client's offer to the KS
Get the KS's counter offer
Generate NT1: Agent made an offer event
NT1: evKSMadeOffer
3:ClientMakesOffer
Pass Agents's offer to Client
Get Client's counter offer
Generate NT3: evClientMadeOffer event
NT2: evKSOffer
2:CompareOffers
If time has expired,
Generate R3: evNegotiationEnded event
else if KS has made an offer,
Compare with Client's last offer
If Client's last offer is higher,
Generate R3: NegotiationEnded event
else,
Generate NT2: evKSOffer event
else if Client has made an offer,
Compare with KS's last offer
If Client's last offer is higher,
Generate R3: evNegotiationEnded event
else,
Generate NT4: evClientMadeOffer event
NT3: evClientMadeOffer
NT4: evClientOffer

Figure 8: State transition diagram for the Negotiation Thread object

```
        </xs:element>
        <xs:element name="defExpression" minOccurs="0"
            maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="letExpression"
                        maxOccurs="unbounded">
                        <xs:complexType>
                            <xs:complexContent>
                                <xs:extension base="letExpressionType">
                                    <xs:attribute name="Name"
                                    type="xs:string" use="required"/>
                                </xs:extension>
                            </xs:complexContent>
                        </xs:complexType>
                    </xs:element>
                </xs:sequence>
                <xs:attribute name="Value" type="xs:string"
                    use="optional"/>
            </xs:complexType>
        </xs:element>
        <xs:element name="stereotypedExpression" maxOccurs="unbounded">
            <xs:complexType>
                <xs:choice>
                    <xs:element name="INVExpression"
                        type="oclExpressionType"/>
                    <xs:element name="PREExpression"
                        type="oclExpressionType"/>
                    <xs:element name="POSTExpression"
                        type="oclExpressionType"/>
                </xs:choice>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="classifierContextType">
    <xs:attribute name="Value" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="operationContextType">
    <xs:sequence>
        <xs:element name="operationName" type="operationNameType"/>
        <xs:element name="formalParameterList"
            type="formalParameterListType" minOccurs="0"/>
        <xs:element name="returnType" type="typeSpecifierType"
            minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="Name" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="formalParameterListType">
    <xs:sequence>
        <xs:element name="formalParameter" type="formalParameterType"
            maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="collectionTypeType">
    <xs:sequence>
        <xs:element name="simpleTypeSpecifier" type="oclObjectType"/>
    </xs:sequence>
    <xs:attribute name="CollectionKind" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
                <xs:enumeration value="Set"/>
                <xs:enumeration value="Bag"/>
                <xs:enumeration value="Sequence"/>
                <xs:enumeration value="Collection"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
<xs:complexType name="oclExpressionType">
    <xs:sequence>
        <xs:element name="letExpression" type="letExpressionType"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="expression" type="expressionType"/>
    </xs:sequence>
    <xs:attribute name="Value" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="expressionType">
    <xs:sequence>
        <xs:element name="logicalExpression"
            type="logicalExpressionType"/>
    </xs:sequence>
    <xs:attribute name="InParenthesis" type="xs:boolean"
        use="required"/>
</xs:complexType>
<xs:complexType name="letExpressionType">
    <xs:sequence>
        <xs:element name="formalParameterList"
            type="formalParameterListType" minOccurs="0"/>
        <xs:element name="typeSpecifier" type="typeSpecifierType"
```

```
            minOccurs="0"/>
        <xs:element name="expression" type="expressionType"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="logicalExpressionType">
    <xs:sequence>
        <xs:element name="relationalExpression"
            type="relationalExpressionType"/>
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="logicalOperator"
                type="logicalOperatorType"/>
            <xs:element name="relationalExpression"
                type="relationalExpressionType"/>
        </xs:sequence>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="relationalExpressionType">
    <xs:sequence>
        <xs:element name="additiveExpression"
            type="additiveExpressionType"/>
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="relationalOperator"
                type="relationalOperatorType"/>
            <xs:element name="additiveExpression"
                type="additiveExpressionType"/>
        </xs:sequence>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="additiveExpressionType">
    <xs:sequence>
        <xs:element name="multiplicativeExpression"
            type="multiplicativeExpressionType"/>
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="addOperator" type="addOperatorType"/>
            <xs:element name="multiplicativeExpression"/>
        </xs:sequence>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="multiplicativeExpressionType">
    <xs:sequence>
        <xs:element name="unaryExpression" type="unaryExpressionType"/>
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="multiplyOperator"
                type="multiplyOperatorType"/>
            <xs:element name="unaryExpression"
```

```
                type="unaryExpressionType"/>
        </xs:sequence>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="unaryExpressionType">
    <xs:choice>
        <xs:sequence>
            <xs:element name="unaryOperator" type="unaryOperatorType"/>
            <xs:element name="postfixExpression"
                type="postfixExpressionType"/>
        </xs:sequence>
        <xs:element name="postfixExpression"
            type="postfixExpressionType"/>
    </xs:choice>
</xs:complexType>
<xs:complexType name="postfixExpressionType">
    <xs:sequence>
        <xs:element name="primaryExpression"
            type="primaryExpressionType"/>
        <xs:sequence minOccurs="0">
            <xs:element name="propertyCall"
                type="propertyCallType" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="primaryExpressionType">
    <xs:choice>
        <xs:element name="literalCollection"
            type="literalCollectionType"/>
        <xs:element name="propertyCall" type="propertyCallType"/>
        <xs:element name="expression" type="expressionType"/>
        <xs:element name="literal">
            <xs:complexType>
                <xs:choice>
                    <xs:element name="string" type="oclObjectType"/>
                    <xs:element name="number" type="oclObjectType"/>
                    <xs:element name="enumLiteral"/>
                </xs:choice>
            </xs:complexType>
        </xs:element>
        <xs:element name="ifExpression">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="conditionExpression"
                        type="expressionType"/>
                    <xs:element name="thenExpression"
                        type="expressionType"/>
                    <xs:element name="elseExpression"
                        type="expressionType" minOccurs="0"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:choice>
</xs:complexType>
<xs:complexType name="propertyCallParametersType">
    <xs:sequence>
        <xs:element name="declarator" type="declaratorType"
            minOccurs="0"/>
        <xs:sequence minOccurs="0">
            <xs:element name="expression" type="expressionType"
                maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="complexTypeSpecifierType">
    <xs:sequence>
        <xs:element name="typeSpecifier" type="typeSpecifierType"/>
        <xs:element name="expression" type="expressionType"/>
    </xs:sequence>
    <xs:attribute name="Value" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="declaratorType">
    <xs:sequence>
        <xs:element name="NameList" type="NameListType"/>
        <xs:element name="simpleTypeSpecifier" type="oclObjectType"
            minOccurs="0"/>
        <xs:element name="complexTypeSpecifier"
            type="complexTypeSpecifierType" minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="formalParameterType">
    <xs:sequence>
        <xs:element name="type" type="typeSpecifierType"/>
    </xs:sequence>
    <xs:attribute name="Name" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="literalCollectionType">
    <xs:sequence minOccurs="0">
        <xs:element name="collectionItem" type="collectionItemType"
            maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="CollectionKind" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:NMTOKENS">
                <xs:enumeration value="Set"/>
                <xs:enumeration value="Bag"/>
                <xs:enumeration value="Sequence"/>
                <xs:enumeration value="Collection"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
<xs:complexType name="oclObjectType">
    <xs:attribute name="Value" type="xs:string" use="required"/>
</xs:complexType>
<xs:complexType name="operationNameType">
    <xs:choice>
        <xs:element name="name" type="oclObjectType"/>
        <xs:element name="addOperator" type="addOperatorType"/>
        <xs:element name="multiplyOperator"
            type="multiplyOperatorType"/>
        <xs:element name="logicalOperator"
            type="logicalOperatorType"/>
        <xs:element name="relationalOperator"
            type="relationalOperatorType"/>
        <xs:element name="unaryOperator" type="unaryOperatorType"/>
    </xs:choice>
</xs:complexType>
<xs:complexType name="propertyCallListType">
    <xs:sequence>
        <xs:element name="propertyCall" type="propertyCallType"
            maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="propertyCallType">
    <xs:sequence>
        <xs:element name="pathName"/>
        <xs:element name="qualifiers" minOccurs="0">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="expression"
                        type="expressionType"
                        maxOccurs="unbounded"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="propertyCallParameters"
            type="propertyCallParametersType" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="Type" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
                <xs:enumeration value="ARROW"/>
                <xs:enumeration value="DOT"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="TimeExpression" type="xs:boolean"
        use="required"/>
</xs:complexType>
<xs:complexType name="typeSpecifierType">
    <xs:choice>
        <xs:element name="simpleTypeSpecifier"
            type="oclObjectType"/>
        <xs:element name="collectionType"
            type="collectionTypeType"/>
    </xs:choice>
</xs:complexType>
<xs:complexType name="unaryOperatorType">
    <xs:attribute name="Value">
        <xs:simpleType>
            <xs:restriction base="xs:NMTOKENS">
                <xs:enumeration value="MINUS"/>
                <xs:enumeration value="NOT"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
<xs:complexType name="collectionItemType">
    <xs:sequence>
        <xs:element name="expression" type="expressionType"/>
        <xs:element name="expression" type="expressionType"
            minOccurs="0"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="addOperatorType">
    <xs:attribute name="Value">
        <xs:simpleType>
            <xs:restriction base="xs:NMTOKENS">
                <xs:enumeration value="PLUS"/>
                <xs:enumeration value="MINUS"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
<xs:complexType name="multiplyOperatorType">
    <xs:attribute name="Value" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:NMTOKENS">
                <xs:enumeration value="MUL"/>
                <xs:enumeration value="DIV"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
<xs:complexType name="relationalOperatorType">
    <xs:attribute name="Value" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:NMTOKENS">
                <xs:enumeration value="EQ"/>
                <xs:enumeration value="GT"/>
                <xs:enumeration value="LT"/>
                <xs:enumeration value="GTE"/>
                <xs:enumeration value="LTE"/>
                <xs:enumeration value="NEQ"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
<xs:complexType name="logicalOperatorType">
    <xs:attribute name="Value" use="required">
        <xs:simpleType>
            <xs:restriction base="xs:NMTOKEN">
                <xs:enumeration value="implies"/>
                <xs:enumeration value="or"/>
                <xs:enumeration value="xor"/>
                <xs:enumeration value="and"/>
            </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
</xs:complexType>
<xs:complexType name="NameListType">
    <xs:sequence>
        <xs:element name="NameList" type="NameListType"
            minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="Value">
            <xs:simpleType>
                <xs:restriction base="xs:NMTOKEN">
                    <xs:enumeration value="Y"/>
                    <xs:enumeration value="p"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
</xs:schema>
```

G Fixture Design Experiment

G.1 Overview of the Fixture Design Problem

The inputs to the system are the edge connector pin (we will shortly call pin) requirements for one or more UUTs and the configuration of the tester (see Figure 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, analog instruments, and their position within the chassis that holds all the channel cards and the analog instruments. The outputs of the system are the mappings between the UUT pins and the channels of the channel cards (like UUT pin 1 will be connected to the channel 2 of M927 type channel card at slot 3 of the chassis).

In the experimental model, the system is represented by three sub-systems:

- Tester sub-system (see Figure 9 for the Class Diagram for the Tester sub-system).
- Fixture sub-system (see Figure 10 for the Class Diagram for the Fixture sub-system).
- UUT sub-system (see Figure 11 for the Class Diagram for the UUT sub-system)

G.2 Empirical Calculation of Phase Transition regions

At this point, we have a number of candidate phase transition variables (or Order Parameters as Cheeseman refers to) 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 Appendix G.2.1 for a details).

Candidate variables:

- *Channel card size* (Number of channels on a channel card)

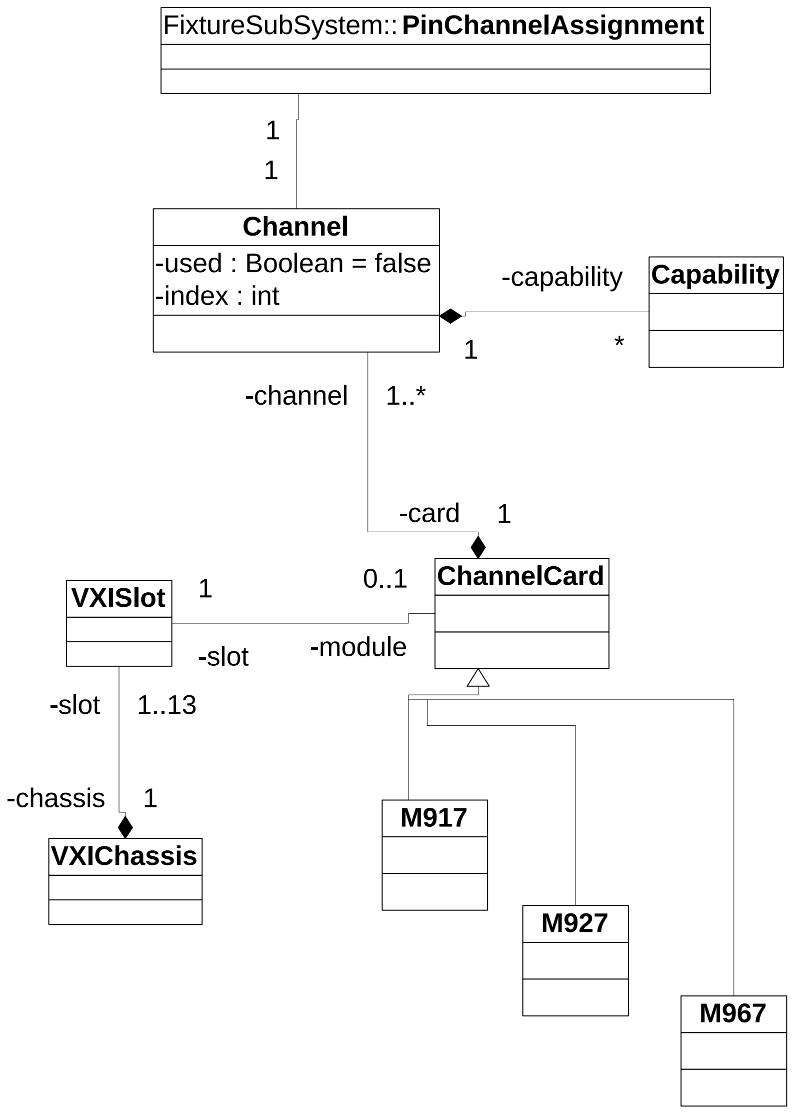

Figure 9: Functional Board Tester Sub-System

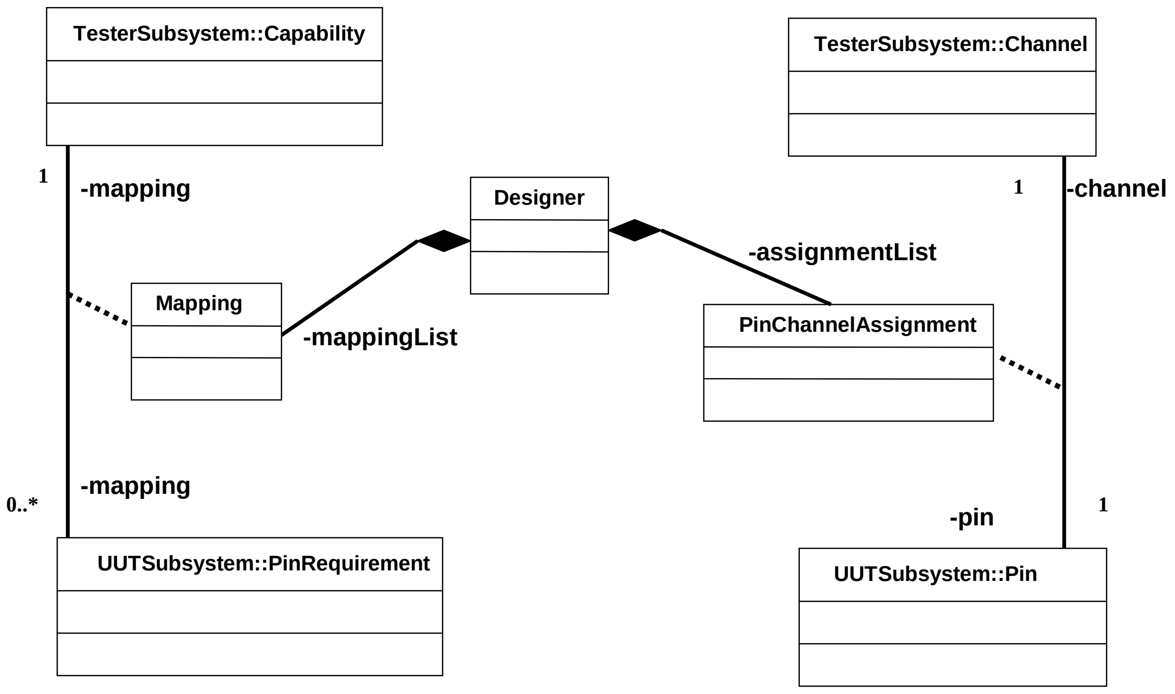

Figure 10: Fixture Sub-System

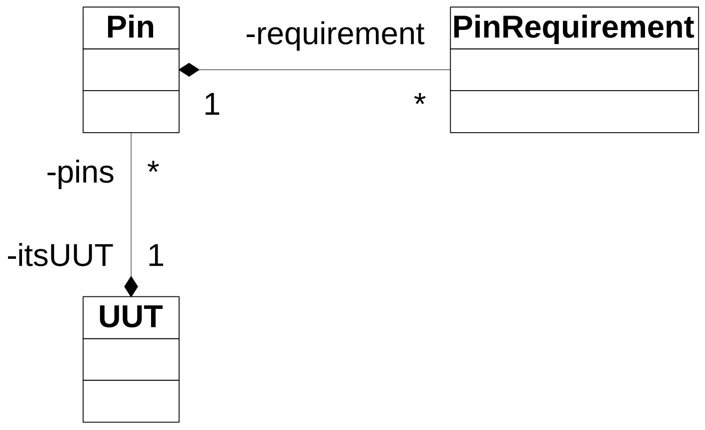

Figure 11: Unit Under Test (UUT) Sub-System

- *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

G.2.1 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 has 3 channels each. Two kinds of pin requirements are used; bus pins, where pins in the same bus needs 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
       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 2,3,4,5,6, and 7. The following table summarizes the results:

CC size	Nodes visited	Depth of search tree	Result
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

G.3 Constraints for Fixture Design Experiment in OCL

```
package FXT::Definitions


-- Rec() - maps a pin to its requirements..

context Pin::Rec() : Set(PinRequirement)
    post: result = self.requirements

-- UUT related definitions

context UUT def:

    -- P - the set of all UUT pins

    let P : Set(Pin) = self.pins
    let N_P : Integer = self.pins.size

    -- R - the set of all possible pin requirements.
    --     Start with an empty set and accumulate set R
    --     with the requirements collected from all pins.

    post: P->iterate(p : Pin; R : Set(PinRequirement) = Set{} |
        R->including( p.Rec() )  )


-- ChannelCard related definitions.

context VXIChassis def:

    -- CC - the set of all channel cards in the tester system.

    let CC : Set(ChannelCard) = self.slot.module
    let N_CC_MAX : Integer = 13 -- some constant

    -- C - the set of all channels in the tester system

    post: CC->iterate(cc : ChannelCard; C : Set(Channel) = Set{} |
        C->including( cc.channels ) )

    -- CAP - the set of all capabilities supported by the tester

    post: C->iterate(c : Channel; CAP : Set(Capability) = Set{} |
        CAP->including( c.capability ) )
```

```
-- Cont() - maps a channel to the channel card that contains the
--           channel

context VXIChassis::Cont(c : Channel) : ChannelCard
    post: result = c.card

-- Cont() - maps a set of channel to a set of channel cards that
--          contains these channels

context VXIChassis::ContU(setChannel : Set(Channel)) :
            Set(ChannelCard)

    post: result = setChannel->iterate(c:Channel;
                             ccSet:Set(ChannelCard) = Set{} |
        if Designer::PC_inv(c) <> Set{}
        then ccSet->including(c.Cont())
        else 1
        endif)

-- Cap() - the capability function that assigns a set of
--         capabilities to the given channel

context Channel::Cap() : Set(Capability)
    post: result = self.capability

-- f_map() - maps a pin requirements to a channel capability

context Designer::f_map(r : PinRequirement) : Capability
    post: result = r.mapping

-- PC() - the mapping function that returns the channel that is
--        associated with the given pin.

context Designer::PC(p : Pin) : Channel

    -- A constant with type Channel that represents
    -- invalid channel assignment.

    def: let c_NULL : Channel = NULL

    pre: P->includes(p)   -- p is an element of P

    -- search pin to channel assignment list to find p
    -- then return channel associated with p..
    -- else return c_NULL.
    post:
       let PCAssign : Set(PinChannelAssignment) = self.assignmentList
       let assignedChannels : Set(Channel)
                = PCAssign->select(a | a.pin = p)
       in
       result = if assignedChannels->size() = 1
            then p.assignment.channel
            else c_NULL
            endif

-- PC() - 2nd version of the the mapping function that returns
--        the channel that is associated with the given pin.

context Designer::PC(pSet : Set(Pin)) : Set(Channel)

    pre: P->includesAll(pSet)   -- pSet is a subset of P

    post:
       let PCAssign : Set(PinChannelAssignment) = self.assignmentList
       in
       result = PCAssign->iterate(a;
                        assignedChannels : Set(Channel) = Set{} |
        if pSet.includes(a.pin)
        then assignedChannels->including(a.channel)
        else 1
        endif )


-- PC^{-1} ()

context Designer::PC_inv(c : Channel) : Set(Pin)

    pre: C->includes(c)   -- c is an element of C

    post:
    result = self.AssignmentList->iterate(
            a:PinChannelAssignment;assignedPins:Set(Pin) = Set{}|
            if a.channel = c
            then assignedPins->including(a.pin)
            else 1
            endif)


endpackage
```

```
-- Constraint and Objective functions

package FXT::Constraints

-- First objective function that minimizes the the number of pins
-- that are NOT connected to any channels.

context FXT::Objective1()

    pre constraint1:  P->forAll(p1,p2 |
        p1 <> p2 implies Designer::PC(p1) <> Designer::PC(p2))

    pre constraint2:  P->forAll(p |
        p.Rec()->forAll(r | Designer::PC(p).Cap()->includes(f_map(r))))

    post: result = ONT::Minimize( Designer::PC_inv(c_NULL)->size() )

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

context FXT::Objective2()

    pre constraint1: CC->size() <= N_CC_MAX

    pre constraint2: ( Designer::PC(P)- Set{ c_NULL})->size() <= N_P

    post: result = ONT::Minimize( VXIChassis::Cont(C)->size() )

endpackage
```

H Order Negotiation Experiment

This system is a variation of *two parties, multiple issues* value scoring system defined in Section 2.6.1. The model includes more than two agents, the *manufacturer* (server) and the *customer(s)* (clients). Manufacturer produces one type of product and has n production machines, which can run parallel, in its plant. A potential customer starts a negotiation with a quantity, a fixed price (for the whole batch) and a delivery date. We should note that delivery date is different than the deadline t^c_{max} for the customer.

When multiple customers come with product requests, the *manufacturer* agent starts a separate negotiation thread for each *customer* agent. And, each negotiation thread progresses as it is defined in the bilateral negotiation model. Roman letters, m will stand for the manufacturer, $c_1, c_2, \cdots$ for customers. The manufacturer agent is capable of managing multiple negotiation threads parallel. Separate deadlines t^c_{max} are assigned for each thread.

There are mainly three negotiated issues, *price* (p), *quantity* (q) and *delivery date* (dd) of the service. The values of all issues are normalized, $x_j \in [0, 1]$, for the rest of the equations.

Our version of scoring function of the manufacturer can be defined as:

$$V^m(x^t) = w^m_p V^m_p(x^t_p) + w^m_q V^m_q(x^t_q) + w^m_{dd} V^m_{dd}(x^t_{dd}) \qquad (13)$$

The assumption for the scoring functions is that the manufacturer desires higher price, larger quantity, and later delivery date. Therefore, for all issues j ($j \in \{p, q, dd\}$),

$$V^m_j(x^t_j) = \begin{cases} \frac{x^t_j - min_j}{max_j - min_j}, & \text{for } x^t_j \in [min_j, max_j] \\ 0, & \text{for other values} \end{cases} \qquad (14)$$

In this model, weights for all issues j for manufacturer w^m_j is a function of time t, resources (machines used in the production) $r(t)$, and state S of *plant*, $w^m_j = f(t, r(t), S(t))$.
Overall scoring function for the customer $V^c(x^t)$ has the same form as the manufacturer's. However, the customers use different weights w^c_j and individual scoring functions V^c_j. For the customer, lower *price* is better, $V^c_p(x^t_p) = \frac{max_j - x^t_j}{max_j - min_j}$. A certain value is preferred, no more no less, for *quantity* and *delivery date*. Diverging from that value towards both directions will bring less satisfaction. For issues j ($j \in \{q, dd\}$), $mean_j = \frac{max_j + min_j}{2}$,

$$V^m_j(x^t_j) = \begin{cases} \frac{mean_j - |x^t_j - mean_j| - min_j}{mean_j - min_j}, & \text{for } x^t_j \in [min_j, max_j] \\ 0, & \text{for other values} \end{cases} \qquad (15)$$

Customer uses *Time-dependent* tactics. In these tactics, the dominant factor used to decide which value to offer is time, t. These tactics vary the value of the issue depending on the remaining negotiation time.
Using the definitions given in [29], we can use the following two formulas to calculate the value to be offered by a manufacturer to a customer for issue j at time t' as:

$$x^{t'}_{m \to c}[j] = min^m_j + \alpha^m_j(max^m_j - min^m_j) \tag{16}$$

$$x^{t'}_{m \to c}[j] = min^m_j + (1 - \alpha^m_j)(max^m_j - min^m_j) \tag{17}$$

where $0 \leq t' \leq t^{max}_c$, and α^m_j is a parameter, which depends on the tactic used by the manufacturer. If V^m_j decreases as the value of issue j increases, then the manufacturer uses Equation 16, otherwise the manufacturer uses Equation 17, for calculating the value to be offered.

A wide range of tactics can be defined by using different α^m_j. However, functions must ensure that $0 \leq \alpha^m_j \leq 1$ and $\alpha^m_j = 1$ at t^{max}_c. A simple form of α^m_j, which depends on the time passes during the negotiation, can be:

$$\alpha^m_j = \frac{min(t', t^{max}_c)}{t^{max}_c} \tag{18}$$

Manufacturer uses *resource-dependent* and *imitative* tactics. Resource-dependent tactics are similar to time-dependent ones. Therefore, Equation 16 and 17 for calculating the counter-offer, can be applied to resource-dependent tactics, as well. The only major change is in the calculation of $\alpha^m_j(r(t))$ functions. The assumption is that, at any time t', the least available resource type drives the calculation of the counter-offer. For the manufacturing agent m, for all $j \in \{p, q, dd\}$, and time t', $\alpha^m_j(r(t))$ can be defined as:

$$\alpha^m_j(t') = min(\kappa_{\vec{j/r}} \otimes \vec{r^{t'}}) \tag{19}$$

$$(a_1, a_2, \cdots, a_n) \otimes (b_1, b_2, \cdots, b_n) = (a_1b_1, a_2b_2, \cdots, a_nb_n) \tag{20}$$

where $\vec{r^{t'}}$ is a vector of quantities of each resource type $(r^{t'}_{labor}, r^{t'}_{servers}, r^{t'}_{money}, \cdots)$ at time t' (each resource type is represented by a different dimension of the vector), $\kappa_{\vec{j/r}}$ is a vector of reciprocals of the maximum amount of each resource type at the plant $(1/(r^{max}_{labor}, 1/r^{t'}_{servers}, 1/r^{t'}_{money}, \cdots))$.

References

[1] Self-Adaptive Software (SAFER). URL: www.irobot.com/rd/p16safer.asp, iRobot Corporation.

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

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

[4] 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.

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

[6] 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.

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

[8] 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.

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

[10] 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.

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

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

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

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

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

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

[17] 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.

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

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

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

[21] 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.

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

[23] 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.

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

[25] 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.

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

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

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

[29] 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.

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

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

[32] 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.

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

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

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

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

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

[45] 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.

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

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

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

[49] 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.

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

[51] 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.

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

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

[54] 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.

[55] 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.

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

[57] 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.

[58] 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.

[59] 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.

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

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

[62] 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.

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

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

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

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

[67] A. Ledeczi. Adaptive Model-Integrated Computing. URL: www.isis.vanderbilt.edu/projects/asc/asc.htm, ISIS Vanderbilt University.

[68] A. K. Mackworth. Networks of constraints: Fundemental properties and application to picture processing. *Artificial Intelligence*, 8(1), 1977.

[69] P. Maes. General tutorial on software agents. Presentation: http://pattie.www.media.mit.edu, MIT, 1997.

[70] 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.

[71] M. Mehl, M. Muller, T. Popov, and K. Scheidhauer. DFKI Oz User's Manual, May 1995.

[72] A.C. Meng. On Evaluating Self-Adaptive Software. In *IWSAS*, volume 1936 of *Lecture Notes in Computer Science*, pages 65–74. Springer, 2001.

[73] L. Michel and P. Van Hentenryck. Modeler++: A Modeling Layer for Constraint. Programming Libraries CS-00-07, Brown University, December 2000.

[74] U. Montanari. Networks of constraints: Fundemental properties and application to picture processing. *Information Science*, 7, 1974.

[75] U. Montanari and F. Rossi. Constraint relaxation may be perfect. *Artificial Intelligence Journal*, 48:143–170, 1991.

[76] U. Montanari and F. Rossi. Constraint solving and programming: What's next? *ACM Computing Surveys*, 28(4), 1996.

[77] 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.

[78] 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.

[79] 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.

[80] Object Management Group, Inc. *The Common Object Request Broker: Architecture and Specification*, 3.0.2 edition, December 2002.

[81] Object Management Group, Inc. *XML Metadata Interchange Specification*, 1.2 edition, January 2002.

[82] Object Management Group, Inc. *Unified Modeling Language Specification*, 1.5 edition, March 2003.

[83] V. Pareto. *Manual of Political Economy.* Societa editrice libraria, Milan, 1906.

[84] 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.

[85] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81, 1996.

[86] D.G. Pruitt. *Negotiation Behavior.* Academic Press, 1981.

[87] P.W. Purdom. Search Rearrangement Backtracking and Polynomial Average Time. *Artificial Intelligence*, 21(1–2):11–134, 1983.

[88] 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.

[89] H. Raiffa. *The Art and Science of Negotiation.* Harvard University Press, Cambridge, USA, 1982.

[90] Anita Raja and Victor Lesser. Efficient meta-level control in bounded rational agents. Technical report, UMass Computer Science Technical Report 2002-051, December 2002.

[91] 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.

[92] F. Rossi. Constraint Logic Programming. In *Proc. ERCIM/Compulog Net workshop on constraints.* Springer-Verlag, LNAI 1865, 2000.

[93] F. Rossi, C. Petrie, and V. Dhar. On the Equivalence of Constraint Satisfaction Problems. Technical Report ACT-AI-222-89, MCC, Austin,TX, 1989.

[94] F. Rossi and A. Sperduti. Learning solution preferences in constraint problems. *Journal of Experimental and Theoretical Computer Science*, 10, 1998.

[95] D. Roth. On the Hardness of Approximate Reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996.

[96] S. Russell. Rationality and Intelligence. In Renee Elio, editor, *Common sense, reasoning, and rationality*. Oxford University Press, 2002.

[97] 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.

[98] S. Russell and S. Zilberstein. Optimal Composition og Real-time Systems. *Artificial Intelligence*, 83, 1996.

[99] S. J. Russell and D. Subramanian. Provably bounded optimal agents. *Journal of Artificial Intelligence Research*, 3, May 1995.

[100] 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.

[101] B. Selman and H. Kautz. Knowledge compilation and theory approximation. *Journal of the ACM*, 43(2):193–224, 1996.

[102] B. Selman and S. Kirkpatrick. Critical behavior in the computational cost of satisfiability testing. *Artificial Testing*, 81:273–295, 1996.

[103] 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.

[104] S. Sen, editor. *Satisficing Models*. AAAI Press, 1998.

[105] H. Simon and J. Schaeffer. The Game of Chess. In Aumann and Hart, editors, *Handbook of Game Theory with Economic applications*. North Holland, 1992.

[106] H. A. Simon. A Behavioral Model of Rational Choice. *Quarterly Journal of Economics*, 59:99–118, 1955.

[107] H. A. Simon. Rational Choice and Structure of the Environment. *Psychological Review*, 63(2):129–138, 1956.

[108] H. A. Simon. Invariants of Human Behavior. *Annual Review Psychology*, 41:1–19, 1990.

[109] B. Smith and M.E. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81, 1996.

[110] G. Smolka. *An OZ Primer*. DFKI Oz documentation series, Saarbrucken, Germany, 1995.

[111] W. C. Stirling and M. A. Goodrich. Satisficing games. *Information Sciences*, 114:255–280, March 1999.

[112] 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.

[113] 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.

[114] G.J. Sussman and G.L. Steele. CONSTRAINTS-a language for expressing almost-hierarchical descriptions. *AI Journal*, 14(1), 1980.

[115] I.E. Sutherland. *SKETCHPAD: A Man-Machine Graphical Communication System.* MIT Lincoln Labs, Cambridge,MA, 1963.

[116] 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.

[117] J. Sztipanovits and G. Karsai. Model-Integrated Computing. *IEEE Computer*, pages 110–112, April 1997.

[118] E. Tsang, P. Mills, R. Williams, J. Ford, and J. Borrett. A computer aided constraint programming system. In *Proceedings PACPL'99*, 1999.

[119] 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.

[120] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

[121] P. Van Hentenryck. Helios: A modeling language for global optimization. In *Proceedings PACT96*, pages 317–335, 1996.

[122] P. Van Hentenryck. *Numerica: A modeling language for global optimization.* MIT Press, 1997.

[123] 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.

[124] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence*, 58:113–159, 1992.

[125] 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.

[126] K. Xu and W. Li. Exact phase transitions in random constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 12:93–103, 2000.

[127] 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.

[128] S. Zilberstein. *Operational Rationality through Compilation of Anytime Algorthms.* Ph.d. dissertation, Computer Science Division, University of California at Berkeley, 1993.

[129] 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.

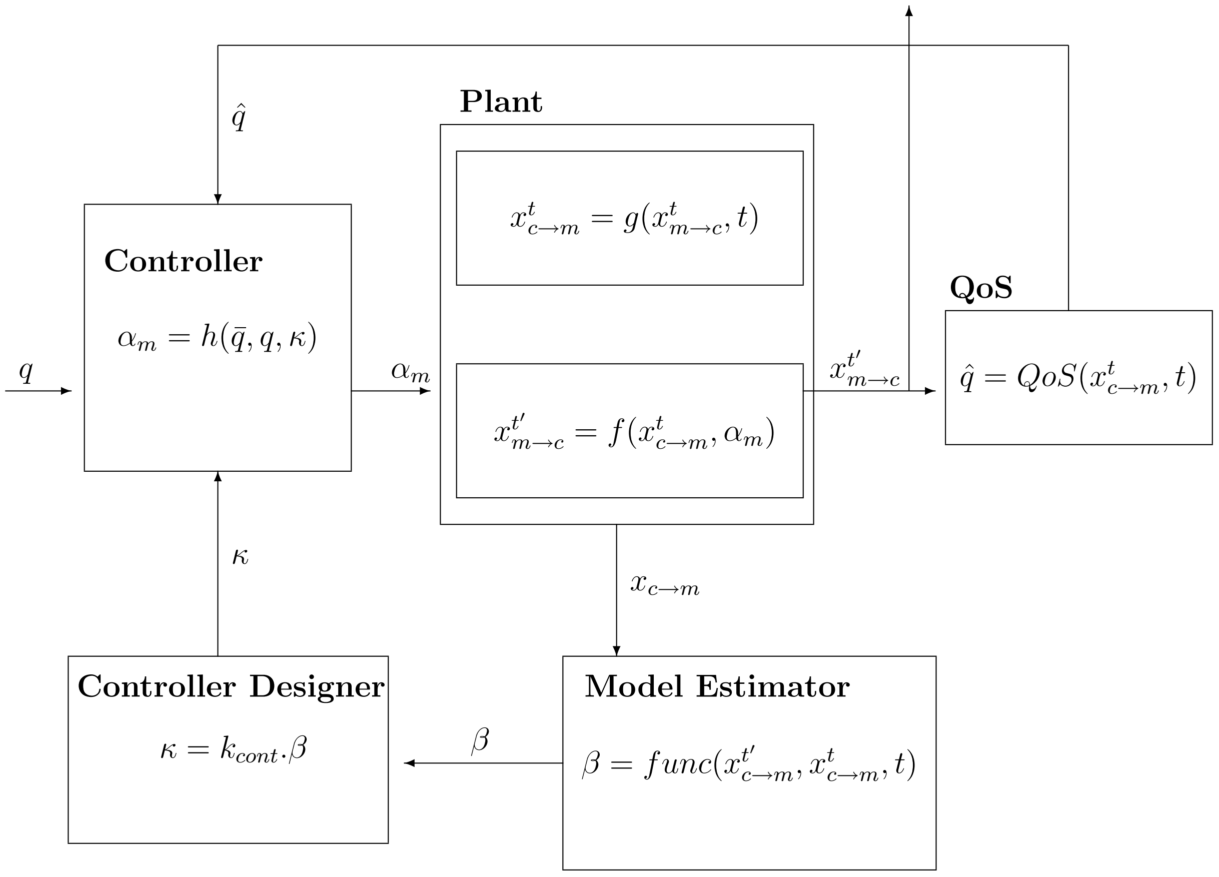
Plant
$x^t_{c\to m} = g(x^t_{m\to c}, t)$
$x^{t'}_{m\to c} = f(x^t_{c\to m}, \alpha_m)$
Controller
$\alpha_m = h(\bar{q}, q, \kappa)$
q
α_m
$\hat{q}$
QoS
$\hat{q} = QoS(x^t_{c\to m}, t)$
$x^{t'}_{m\to c}$
κ
$x_{c\to m}$
Controller Designer
$\kappa = k_{cont}.\beta$
β
Model Estimator
$\beta = func(x^{t'}_{c\to m}, x^t_{c\to m}, t)$

Figure 12: Adaptive control inside Order Negotiation agent