Using UML and OCL for Representing Multiobjective Combinatorial Optimization Problems

Yönet A. Eracar
Department of Mechanical and Industrial Engineering,
Northeastern University, 360 Huntington Avenue, Boston, MA 02115, USA,
yeracar@gmail.com

Mieczyslaw M. Kokar
Department of Electrical and Computer Engineering,
Northeastern University, 360 Huntington Avenue, Boston, MA 02115, USA,
kokar@coe.neu.edu

Abstract

This paper describes the results of a preliminary feasibility study of an approach to representing Multiobjective Combinatorial Optimization Problems in UML (structural constraints) and OCL (procedural constraints) and then automatically translating the representations to a Constraint Satisfaction solving language (Oz) for execution. The paper presents two examples of the application of the approach - a job scheduling problem and a (fixture) design problem. The main goal of this paper is to investigate directions towards a standard, graphical language for representing combinatorial optimization problems. The paper shows that for the two selected problems it is easy to represent structural constraints in UML and that procedural constraints are representable in OCL. The results also show that a developed translator automatically converts the UML/OCL representations to Oz and that the resulting Oz program performs very reasonably, in some cases outperforming the hand-written benchmark programs.

1 Introduction

Multiobjective optimization is a core area in engineering, business practice and research. Application areas of multiobjective optimization include resource allocation, transportation, logistics, distribution, investment decisions, business planning with uncertain information, and others [1, 2, 3, 4, 5]. Multiobjective optimization problems are formulated in terms of performance criteria (objective functions) and constraints. Optimization problems divide into two categories – those with continuous variables and those with discrete variables; the latter are called *combinatorial optimization* problems [6, 7, 8, 9, 10, 11, 12, 13]. The word "combinatorial" refers to the fact that only a finite number of alternative feasible solutions exist. Multiobjective combinatorial optimization problems are referred to as "MOCOPs."

The general formulation of MOCOP is as follows:

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

The intent here is to state that the M objective functions, f_m, are to be optimized (either minimized or maximized), subject to three types of constraints (J inequality constraints, K equality constraints and N *variable bounds*). The terms $h_k(x)$ and $g_j(x)$ are called *constraint functions*. The variables $x_1, \ldots, x_N$ are called *decision variables*. A *feasible solution* $x = (x_1, x_2, \ldots, x_N)$ is an assignment that satisfies all of the $(J+K)$ constraints and all of the $2N$ variable bounds. The values of decision variables bounded by these constraints constitute the *decision space* D.

Finding a solution to a MOCOP requires finding an instantiation of the problem variables that not only satisfies the constraints (i.e., solving a Constraint Satisfaction Problem), but also provides high values for the performance criteria defined by the M objective functions. A way to search for an optimal solution is first converting the original problem into a number of constraint satisfaction problems (CSP) [14, 15] and then selecting a solution that is best from the point of view of optimality.

However, when a CSP problem is NP-complete, full algorithmic optimal solutions cannot be expected [16]. Moreover, since multiple performance criteria are involved, any solution requires trade-offs among them. One has to settle for less than optimality. Conceptualizations of this kind of problem formulation are known as *satisficing* solutions [17, 18, 19, 20, 21, 22, 23]. Solutions of this kind of problem typically use search as one of the components. However, since a generic search algorithm does not guarantee finding solutions within a finite time, the search incorporates features specific to a particular problem. In other words, the search is domain and problem specific.

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

To solve the CSP problems described above, off-the-shelf CSP solvers could be used. However, all of them require low-level coding in a proprietary CSP language. A standard-based, user-oriented, high-level language that would hide the implementation details related to the CSP solver is desirable [24]. However, we have not found a standard language for this kind of task. This prompted us to investigate the feasibility and the reasonableness of an approach in which an optimization problem would be specified in a high-level language and then solved by automatically translating the specification into a CSP programming language. The first decision along this path of reasoning was to select a language for representing problem specifications and a target CSP language for executing search for a satisficing solution.

While there are no formal requirements for a language to capture an optimization problem specification, it seems that having an abstract graphical representation of a problem would be desirable. Moreover, in order to be able to use such specifications in various systems, the specification language should be open and standardized. For these reasons, we have focused on the Unified Modeling Language (UML) [25, 26]. UML has a reasonably well defined semantics, it is open, standardized and has a relatively large community of users. Perhaps the most powerful feature of UML is its ability to represent structural constraints. However, not all aspects of an optimization problem can be represented in UML. In particular, optimization objectives are not representable and also some constraints would be either difficult or even impossible to capture. For this reason we decided to extend the capabilities of UML with the features of the Object Constraint Language (OCL) [27], which is the constraint language associated with UML.

For the purpose of this investigation, Oz [28, 29] has been selected as the target language for the execution of an optimization problem. Oz provides good support for constraint programming (basic constraints, language for definition of other constraints, "computation spaces" for user-controlled search) and for agent-based problem solving. It has an open architecture and the source code is publicly available; supports integration with other programs written in different programming languages, e.g. C, C++; comes with a run-time debugging environment (called Mozart); provides a graphical user interface that helps to monitor the progress of execution.

The approach we are investigating includes the following steps (activities):

1. Specification of structural constraints of a MOCOP in UML using a UML tool.

2. Specification of optimization objectives and procedural constraints in OCL using an OCL editor.

3. Mapping of the UML/OCL representation to an agent-based architecture in which each objective (goal) is "represented" by an agent. This mapping is defined using the UML tool.

4. Automatic translation of the UML/OCL representation to a CSP language.

5. Running a CSP solver for each of the objective functions.

6. Running an algorithm to negotiate among multiple optimization objectives.

7. A mechanism for recognizing computation-intensive regions in the search space (not described in this paper).

To evaluate the feasibility of the approach, an experimental system has been implemented and tested using two scenarios – a testing fixture design and a job-scheduling system.

Since the problems are NP-complete, test cases were designed to cover both hard and easy regions. The quality of solutions to the two MOCOPs was compared against known benchmark approaches. In both cases, the obtained results were comparable. A small advantage of our system was that it had a lower probability of false alarm, i.e., it was quicker to recognize that the search problem was very hard and thus could not be solved within given time bounds.

This paper focuses on the first two of the steps described above; they cover the issue of representing MOCOPs in UML/OCL. The main question here is whether it is possible and easy to use UML/OCL for representing MOCOPs, and whether it is worthwhile to work on a system that uses UML/OCL as a front end. This paper does not provide any quantitative assessments of this direction. However, it shows all the details of representations of the two selected MOCOPs in UML/OCL so that the reader who is interested in pursuing this direction can use these details in his/her considerations. The reader of this paper is expected to have some basic knowledge of both UML and OCL.

The paper is structured as follows. In the next section we provide an overview of the approach. Then in the sections that follow we provide a step by step explanation how the particular elements of the approach have been realized in our implementation. Sections 3 and 7 describe the two optimization problems used as our examples. Sections 3.2 and 7.2 show how the structural constraints of the two optimization problems are represented in UML. Sections 3.3 and 7.3 discuss the representation of the optimization goals in OCL. Sections 3.4 and 7.4 show how constraints of the optimization problems can be represented in OCL. Sections 4 and 5 briefly discuss how we developed the OCL parser and the Oz code generator, respectively. They also overview the inputs these modules take and the results they produce. Since one part of our approach involves the use of agents, Section 6 discusses how this technology was used in our program. In Section 8 we provide a brief statement on the evaluation of the system. A more full presentation of these results, including the adaptation mechanisms of the developed code, will be subject of another paper. Section 9 gives a brief overview of the literature on various languages for representing and solving constraint satisfaction problems. Finally, Section 10 presents conclusions and suggestions for future work.

2 Approach Overview

To achieve the goals outlined in Section 1, a prototype system was implemented. In Figure 1 we show the process used in the development and execution of the system. The process consists of seven *activities* subdivided into three phases - *Specification, Code Generation* and *Execution*. Arrows that connect activities show the flow of results among the activities.

The Specification phase is manual - the user of the system develops the specification (in UML/OCL) of a given MOCOP, including the optimization objectives and the constraints. The Code Generation phase uses automatic translation tools. The Execution phase is the running of the code generated in the previous phase. In the subsections that follow we describe each of the activities.

3 Specification: A Job Scheduling Example

3.1 Problem Formulation

This example deals with a job scheduling problem. There are a number of jobs, each consisting of one or more tasks. Jobs have due dates by which all the tasks for the job need to be completed. Tasks have prerequisites, i.e., sets of tasks that need to be completed before they can start. Each task requires a single resource and takes a certain time to be completed. There is only one of each resource type available and a resource can not be shared by more than one task at the same time. The objectives are to complete all the jobs as soon as

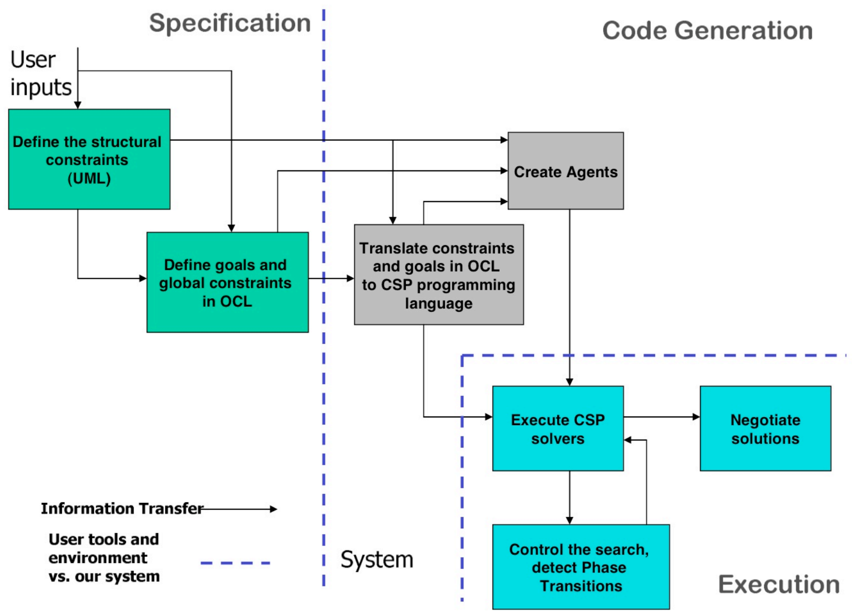

Specification
Code Generation
User inputs
Define the structural constraints (UML)
Create Agents
Define goals and global constraints in OCL
Translate constraints and goals in OCL to CSP programming language
Execute CSP solvers
Negotiate solutions
Control the search, detect Phase Transitions
Information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

Figure 1: Overview of the proposed approach

possible (minimize the completion time for the set of jobs) and to minimize the average time for the completion of all the jobs.

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

- Job - the set of all jobs
- $Task$ - the set of all tasks (the union of tasks for all the jobs)
- $Resource$ - the set of all resources
- $Times \subset Int$ - time interval
- $dur : Task \to Times$ - a function that returns the duration of a given task
- $preTask : Task \to 2^{Task}$ - a function that returns the set of pre-requisite tasks for a given task
- $due : Job \to Times$ - a function that returns the due date or deadline for the completion of a given job
- $|Job|$ - the size of the job set, i.e., the number of jobs to be scheduled
- $res : Task \to Resource$ - a function that maps a task to a resource (to simplify the problem, we assume that a task depends on a single resource)
- $start : Task \to Times$ - a function that returns the starting date for a given task (in our problem, starting dates for the tasks are the decision variables)

We use the *dot* notation to represent values of functions. For instance, we say $T.start$ to denote $start(T)$ for a given task T. This notation is easier to relate to the UML representation of the problem.

The objectives and the constraints for this problem are formalized below. Note that the notation $Times^{|Task|}$ represents the Cartesian product of the set $Times$ with itself n times, i.e., $Times \times Times \times \ldots \times Times$, where n is equal to the number of tasks. Consequently, $\overrightarrow{start}$ represents n-tuples (elements of the Cartesian product).

Objectives:

Objective 1: Minimize the end date, *endDate*, for the completion of all jobs:

$$endDate = \min_{\overrightarrow{start} \in Times^{|Task|}} \max_{T \in Task} (T.start + T.dur) \tag{2}$$

Objective 2: Minimize the average time, *avgEndDate*, to complete a job, where

$$avgEndDate = \min_{\overrightarrow{start} \in Times^{|Task|}} \frac{\sum_{J \in Job} \max_{T \in J.task}(T.start + T.dur)}{|Job|} \tag{3}$$

Constraints:

Constraint 1: All tasks that are prerequisites for task T should end before T can start:

$$\forall T \in Task\ \forall P \in T.preTask \bullet P.start + P.dur \leq T.start \tag{4}$$

Constraint 2: All jobs must finish by their deadlines:

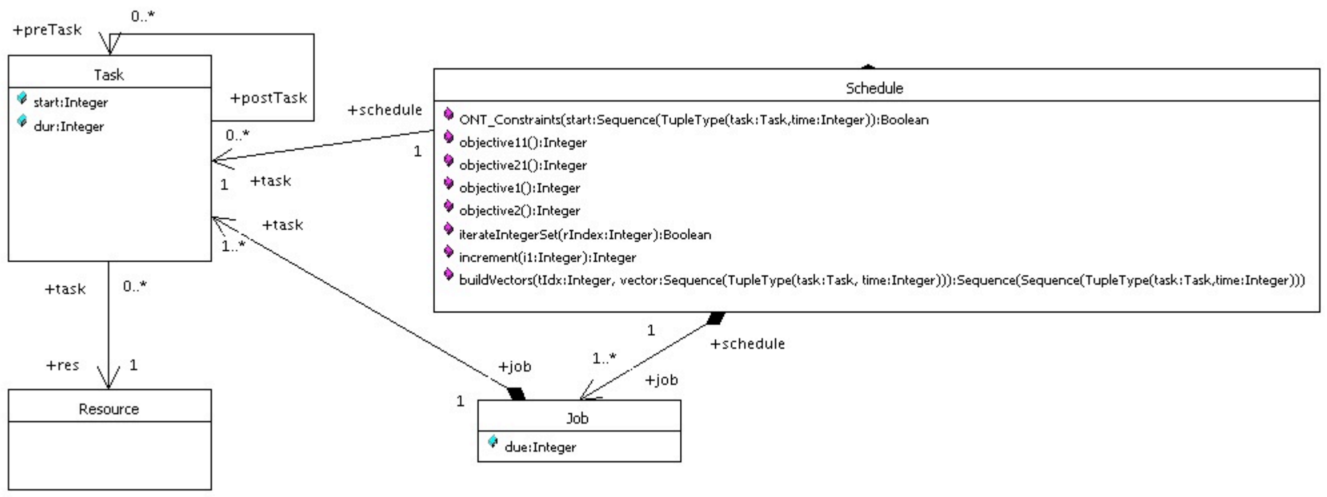
$$\forall J \in Job \bullet \max_{T \in J.task} (T.start + T.dur) \leq J.due \tag{5}$$

Constraint 3: A resource cannot be used by more than one task concurrently:

$$\begin{gathered}\forall T_1, T_2 \in Task \bullet T_1.start \leq T_2.start \ \& \\ T_2.start \leq T_1.start + T_1.dur \Rightarrow T_1.res \neq T_2.res\end{gathered} \tag{6}$$

3.2 Structural Constraints in UML

Structural constraints for this problem are shown in the form of a UML class diagram in Figure 2. Three classes are introduced to represent the three sets: `Job, Task` and `Resource`. Moreover, the `Schedule` class represents schedules. The diagram shows that any task is an exclusive part of (is owned by) one job. This is represented by the filled diamond, which is called *composite aggregation* in UML. Relations are shown as lines connecting the classes with annotations that represent *roles.* Thus `+task` on the association between `Job` and `Task` indicates the existence of a relation between the two classes. Moreover, the annotation `1..*` means that for each job there must be at least one task, but can be more. The role `+job` shows that each task is related to a job. There is no need to put a multiplicity constraint on this role since it is a composite aggregation. The annotation `+preTask` with `0..*` means that for each task there may be multiple prerequisite tasks. Similarly, the multiplicity on `+postTask` indicates that a task can be followed by any number of tasks. The relationship between `Task` and `Resource` is such that the `res` association is a function (exactly one resource is required by a task) and yet the same resource can be required by multiple tasks. And finally, the compartments of the boxes representing classes show that `due` is an attribute of each job, while `dur` is an attribute of a task (the duration) and `start` is an attribute of a task (the start time).



Task
start:Integer
dur:Integer
+preTask
0..*
+postTask
+schedule
0..*
1
1
+task
+task
1..*
+task
0..*
+res
1
Resource
Schedule
ONT_Constraints(start:Sequence(TupleType(task:Task,time:Integer)):Boolean
objective11():Integer
objective21():Integer
objective1():Integer
objective2():Integer
iterateIntegerSet(rIndex:Integer):Boolean
increment(i1:Integer):Integer
buildVectors(tIdx:Integer, vector:Sequence(TupleType(task:Task, time:Integer))):Sequence(Sequence(TupleType(task:Task,time:Integer)))
1
+schedule
1..*
+job
+job
1
Job
due:Integer

Figure 2: Class Diagram for the Job Scheduling System

Finally, `Schedule` is a collection of jobs. It *owns* all the jobs, i.e., a job cannot be member of any other object. `Schedule` holds all the information about objective functions and constraints. Both the objective functions and the constraints are expressed in OCL, as described in the next section. In particular, `objective1()` represents the objective function defined in Eq. 2 and `objective2()` represents the objective function given by Eq. 3. In our definitions of the objective functions we chose to define each of them in two steps, i.e., first define a sub-function and then use the sub-function to define the overall objective. For this reason, `Schedule` shows two auxiliary functions `objective11()` and `objective21()`. The constraints are defined as a Boolean function `ONT_Constraints()`.

3.3 Goals in OCL

Now we discuss the details of the OCL specifications that capture the objective functions in Equations 2 and 3, as well as constraints in Equations 4 through 6. According to Eq. 2 the first objective is to find such assignments of start times to all the tasks so that the overall time of execution of all the jobs is minimized. In other words, the optimal schedule is such that the sum of execution times of all the jobs is minimal. Towards this aim the algorithm must be able to access all the tasks that need to be scheduled. Thus the first thing that needs to be specified is the set of all the tasks. In OCL this can be achieved by collecting all the objects from the class `Task` associated with a given schedule (`self`) by following the links to the tasks by using the `task` association (see Figure 2). However, our intent here is to arrange all the tasks in a sequence (vector), thus we use the OCL construct `asSequence`, as shown below:

```
context Schedule
let Tasks = self.task->asSequence()
```

Moreover, we need to define the set of values for the start times of particular tasks. We assume that the first start time will be 1. The last reasonable start time can be assessed as the max due time for all the jobs. We can specify these facts by the following OCL statements.

```
let MaxDue : Integer = self.job->iterate(J : Job; maxI : Integer = 0 |
if J.due > maxI then J.due else maxI endif)

let Times : Sequence(Integer) = Sequence{1..MaxDue}
```

The first `let` statement iterates over all jobs, `J`, and accumulates the values of `J.due`, keeping the largest found. That value becomes `MaxDue`. The second `let` statement defines a sequence of Integers (`Times`) from 1 to `MaxDue`. `Times` and `Tasks` are then used in the definitions of the objective functions and constraints, as will be shown below.

According to Eq. 2, we need to iterate over all possible vectors $\overrightarrow{start} \in Times^{|Task|}$. Each such vector represents assignments of starting times to particular tasks. We can represent such vectors as sequences of tuples, where each tuple includes a task and a (start) time associated with the task. The set of all such vectors, $Times^{|Task|}$, can be represented as a list; we call it `StartMatrix`. This matrix has the size of $|Task|$ columns and $|Times^{|Task|}|$ rows.

We define `StartMatrix` using the OCL's `buildVectors` function. `StartMatrix` is a sequence of sequences of tuples (task followed by starting time of this task), as specified in line 1. The definition of `buildVectors` is shown below - the main part is between lines 4 and 10. It is a recursive function. The result is a matrix (`StartMatrix`) shown in line 11. The outer quantifier (iterator) of this function uses the `Times` vector described earlier; the iterator (time) index is `t`. The accumulator associated with this iterator is `matrix`. It is initialized to a sequence of empty sequences (line 6) and then, depending on the value of the task index (`tIdx`) either continues appending the next tuple to the vector of tuples, or appends a whole vector as a new row (sequence). The former is stated in the `else` part (line 9) of the `if` statement (a recursive invocation of `buildVectors`), while the latter is stated in the `then` part (line 8). A tuple to be appended to a sequence is represented as `Tuple{task=Tasks->at(tIdx),time = t}`.

```
1: let StartMatrix:Sequence(Sequence(TupleType(task:Task,time:Integer))) =
       buildVectors(1,Sequence{})
2: context Schedule::
3: buildVectors(tIdx:Integer,vector:Sequence(TupleType(task:Task,time:Integer))) :
   Sequence(Sequence(TupleType(task:Task,time:Integer)))

4: def: let startMatrix : Sequence(Sequence(TupleType(task:Task,time:Integer))) =
5:  Times->iterate(t:Integer; matrix:Sequence(Sequence(TupleType(task:Task,time:Integer))) =
6:  Sequence{Sequence{}} |
7:    if tIdx = Tasks->size()
8:      then matrix->append( vector->append(Tuple{task=Tasks->at(tIdx),time = t}))
9:       else buildVectors(tIdx+1, vector->append(Tuple{task=Tasks->at(tIdx),time = t}))
10:    endif)
11: post: result = startMatrix
```

The first objective function (Eq. 2) is represented by two functions, `objective11()` and `objective1()`. First, we show how to map the second part of Eq. 2, i.e.,

$$\max_{T \in Task} (T.start + T.dur)$$

to OCL. In other words, we show the definition of `objective11()`.

```
1: context Schedule::
2: objective11(StartVector:Sequence(TupleType(task:Task,time:Integer))):Integer
3:      def: let Max : Integer = StartVector->iterate(T; AccM = 0 |
4:             if  T.time + T.task.dur >= AccM then T.time + T.task.dur else AccM endif)
5:      post: result = Max
```

The main point of this definition is captured by the `iterate` construct of OCL. It iterates over all the tasks in a `StartVector`, adds the start time to the duration of each task and keeps the maximum value (comparison made in line 4) in the accumulator (`AccM`). In the end it returns the maximum value of this variable as the value of `objective11()` (line 5).

The first objective function, `objective1()`, is defined in OCL as shown below. It iterates over all start vectors (line 3), initializes the accumulator `minVal` to 1000 (line 4), checks constraints (line 5) and invokes `objective11()` for each start vector (line 6). If `objective11()` returns a value that is smaller than the old value of the accumulator `minVal` then it replaces the old value with the new one (line 6), otherwise it keeps the old value of the accumulator (line 7). To refer to the previous value of `minVal` it uses the OCL construct `@pre`. It ends up with the best value of this objective function. Note that the value of 1,000 in line 4 is arbitrary; the only importance of this value is that it should be a large number, which depends on the specifics of the domain of application. The OCL representation of the constraints used in line 5 is discussed after the discussion of the objective functions.

```
1: context Schedule::objective1():Integer
2: def: let I : Integer = 0
3:     let minObjFunc1Value:Integer = StartMatrix->iterate(StartVector :
4:     Sequence(TupleType(task:Task,time:Integer)); minVal: Integer = 1000 |
5:         (if ONT_Constraints(StartVector) then
6:         (if objective11(StartVector) < minVal@pre then objective11(StartVector)
7:             else minVal@pre endif)
8:         else minVal@pre   endif)  )
9:     post: result = minObjFunc1Value
```

The second objective function, `objective2()`, is a translation of the mathematical representation shown in Eq. 3 in a similar way as the first objective function. We first define `objective21()` which is then used by `objective2()`.

```
1: context Schedule::
2: objective21(StartVector:Sequence(TupleType(task:Task,time:Integer))):Integer
3:  def:let total:Integer = self.job->iterate(J:Job; AccT:Integer = 0 |
4:    AccT +
5:     StartVector->select(J.task->includes(task))->sortedBy(time+task.dur)->last().time +
6:     StartVector->select(J.task->includes(task))->sortedBy(time+task.dur)->last().task.dur)
7: let average : Integer = total.div(self.job->size)
8: post: result = average
```

`objective21` takes a start vector (a sequence of tuples representing the start times for all tasks) as the input and returns an integer, which is the average time to complete a single job. The iteration over jobs (lines 3 through 7) adds the completion times for the final tasks of each job. In line 5, the expression `StartVector->select(J.task->includes(task))` creates a collection of (task, time) tuples in `StartVector` that are part of job `J`. This collection is then sorted according to their completion times. And finally, the completion time of the last task is selected and added to `AccT`. Line 6 repeats the same process for selecting the duration time of the last task for job `J`. Line 7 divides the total (definition of `total` starts in line 3) of completion times by the number of jobs and finds the average completion time for all the jobs. Line 8 returns the average completion time.

`objective2()` is very similar in nature to `objective1()` described above so we skip the OCL code here. It should suffice to say that it iterates over all start vectors in `StartMatrix`, checks whether the constraints are satisfied and if so, it calculates the value of `objective21()` and compares it with the previous value. In the end it selects the start vector for which the value of `objective2()` is the smallest.

3.4 Constraints in OCL

The next step is to express the constraints (Equations 4 through 6) in OCL. Below we show the three constraints for this problem. The constraints have a common context as shown below. These constraints are checked in `objective1()` and `objective2()`.

```
context Schedule::
ONT_Constraints(StartVector:Sequence(TupleType(task:Task,time:Integer))):Boolean
```

The first constraint (see Eq. 4) states that tasks can only start after their pre-tasks are completed.

```
let constraint1Value:Boolean=StartVector->forAll(T |
 if T.task.preTask->notEmpty() then T.task.preTask->forAll(P |
  StartVector->select(task=P).time+P.dur<=T.time) else true endif)
```

The second constraint for this problem (see Eq. 5) states that jobs need to be completed before their due dates.

```
let constraint2Value : Boolean = self.job->forAll(J |
 (StartVector->select(J.task->includes(task))->sortedBy(time+task.dur)->
  last().time+StartVector->select(J.task->includes(task))->
  sortedBy(time+task.dur)->last().task.dur  ) <= J.due)
```

The third constraint (see Eq. 6) states that a resource can be used only by at most one job at a time.

```
let constraint3Value : Boolean = StartVector->forAll(T1 |
 StartVector->forAll(T2 |
  if T1 <> T2 then if T1.time < T2.time and T1.time+T1.task.dur>T2.time
   then T1.task.res <> T2.task.res else true endif else true endif))
```

The value of `ONT_Constraints` is true only if all the above three constraints are satisfied.

```
post: result = constraint1Value and constraint2Value and constraint3Value
```

4 Parsing OCL

To produce constraints in OCL we used the tool OCLE: Object Constraint Language Environment [27]. The tool supported UML 1.5 [30]. The user manually inputs constraints using this tool. As shown in Figure 3, the OCL constraints are parsed by the OCL Parser. The output is represented in the format that we call the "OCL Markup Language" (OCLML). This format is defined by an XML Schema represented in the XML Schema Language [31]. The user of the system does not need to see this representation, so we don't show any examples of its use in this paper.

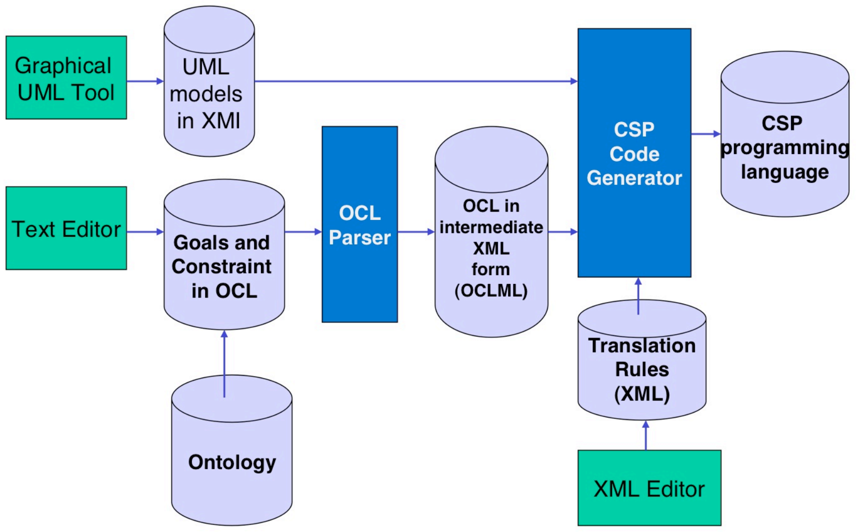


Figure 3: Translation of UML/OCL to the CSP programming language

In developing this system we used the Lex (version 3.3.3.136) and Yacc (version 3.3.3.138) utilities from Mortice Kern Systems, Inc. to generate the OCL parser. To achieve this goal, we had to develop a representation of the OCL grammar in BNF form, using the OCL specification. Moreover, we had to provide the *action* parts of the parser's production rules. This step was based upon the XML schema of the OCLML representation.

5 Code Generation

The results of parsing are passed to the CSP Code Generator. As shown in Figure 3, this module uses Translation Rules (expressed in XML). These rules are specific to a given CSP language. In our experiments we used the Oz CSP language, however the system is designed to work with other CSP languages, provided appropriate translation rules are developed. This feature is achieved due to the separation of the parser from the code generator and due to the use of language specific translation rules. Additionally, the Code Generator takes as input the UML model of the optimization problem (in XMI). The output of this module is code in the target CSP language, Oz.

The Code Generator is written in C#. It first loads the three files - the XMI file representing the UML model, the XML file capturing the OCL constraints and the XML file representing the translation rules for a given language. After verifying that the OCL constraints are valid with respect to the OCLML schema, it invokes the main functionality that walks the OCLML tree and generates code for each UML class in the optimization problem. Member variables of UML classes and associations among the UML classes are extracted by walking through the XMI tree. UML classes in the OCLML tree are then updated with the member variables and the associations, such as inheritance and containment relationships, obtained from the XMI tree.

The function that walks UML classes creates a source file and walks through the sub-elements of the UML classes. Actual code generation is performed for each sub-element. Translation rules are found for a given OCL element and the need for special handling is checked. If there is special code to be generated for a given OCL element then a rule for handling such cases is invoked.

6 Agents

Each objective function in a MOCOP is represented in the CSP code by a separate *software agent*. According to the definition provided by Maes [32], "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 reasons for using software agents are to decentralize the computation and to improve the performance of search by utilizing the adaptive and autonomous nature of agents. In our approach we make agents responsible for solving single objective problems of a MOCOP. Each agent is a CSP solver in the CSP language (Oz). Solutions found by particular agents are sent to a *blackboard* [33]. The blackboard and the agents execute a special synchronization protocol regarding solutions found and termination conditions. These two functionalities are shown in Figure 1 as *Negotiate solutions* and *Control the search, detect Phase Transitions*.

Once the blackboard and the agents reach a search termination point, the blackboard selects the "best" solution found according to the *selection rules*. Towards this end, only the solutions that are *non-dominated* [1] are considered. A non-dominated solution in the set of solutions $S = \{s_1, \ldots, s_n\}$ is such a solution for which there does not exist another solution in the set that would be at least as good for all of the objectives O_i and better on at least one objective. The best solution is defined as the one that is the closest to the so called *utopian solution* $s^* = (s_1^*, \ldots, s_n^*)$ in which each of the solutions s_i^* is the best solution for objective i [4]. The distance metric is defined by the following formula:

$$d(s, s^*) = \sum_S \left(\frac{O_i(s) - O_i(s_i^*)}{O_i(s_i^*)}\right)^2 \tag{7}$$

7 Fixture Design

In this section we describe the second experimental multiobjective optimization scenario used in this paper. It deals with the domain of testing of electronic circuitry boards in which units under test (UUT) are tested using a functional board tester (FBT).

7.1 Problem Formulation

A UUT has a number of *pins* (e.g., 32). For testing, each pin needs to be connected to a *channel*, which is part of a *card* in the FBT. The FBT has a menu of cards, each with a set of *capabilities* associated with particular channels. To test a UUT, each pin's *requirements* need to be matched by a channel's capability. Examples of pin requirements are the digital timing (e.g., 10 MHz, 25 MHz, 50 MHz), the voltage levels that are used (e.g., [+5V,-2V], [+10V,-10V], [+15V,-15V], [+32V,-32V]) and the analog capabilities that are required (e.g., digital multi meter (DMM) input high / input low, function generator (FG) output, timer counter (TC) start / stop / trigger, digital storage oscilloscope (DSO) channel). The description of the configuration of the tester contains data on the number and the types of the channel cards and their positions within the chassis, as well as capabilities of their channels.

A testing facility is interested in keeping the overall cost of testing as low as possible. On one hand, the goal is to minimize the number of channel cards in order to minimize the capital cost. On the other hand, the goal is to minimize the operational cost by using channels that are cheaper to use during testing (e.g., a 25 MHz channel is less expensive than a 50 MHz channel). Thus this problem is an example of a multiobjective optimization problem. The result of this optimization is a mapping between the UUT pins and the channels of the channel cards, e.g., UUT pin 1 will be connected to channel 2 of the third channel card, and so on.

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

- $ChannelCard$ - the set of channel cards in an FBT
- $Channel$ - the set of all channels
- Pin - the set of all pins
- $PinRequirement$ - the set of all pin requirements
- $Capability$ - the set of all channel capabilities
- $channel : Pin \rightarrow Channel$ - function that assigns channels to pins (wirings)

- $requirement : Pin \rightarrow 2^{PinRequirement}$ - function that assigns a set of requirements to a given pin
- $capability : Channel \rightarrow 2^{Capability}$ - function that assigns a set of capabilities to a given channel
- $card : Channel \rightarrow ChannelCard$ - function that assigns a channel card to a channel
- $mapping : PinRequirement \rightarrow Capability$ - function that assigns a channel capability to a pin requirement (the capability must satisfy the pin requirement)

Objectives:
Objective 1: Minimize *avgCap*, the average number of capabilities that the used channels possess. In other words, we would like to use the less capable, less expensive channels first. The decision variable here is *channel*, i.e., the wirings of pins to channels. Elements of *channel* are tuples (pairs) $< p, c >$, $p \in Pin,\ c \in Channel$.

$$avgCap = min_{\overrightarrow{channel} \in Channel^{|Pin|}} \frac{\sum_{P \in Pin} |P.channel.capability|}{|Pin|} \tag{8}$$

Objective 2: Minimize *#UsedChannelCards*, the number of channel cards used. This results in the maximal utilization of each channel card and will minimize the cost.

$$\#UsedChannelCards = min_{\overrightarrow{channel} \in Channel^{|Pin|}} |\bigcup_{P \in Pin} P.channel.card| \tag{9}$$

Constraints:

Constraint 1: Each channel can be assigned to only one pin.

$$\forall p_1, p_2 \in Pin \bullet p_1.channel = p_2.channel \Rightarrow p_1 = p_2 \tag{10}$$

Constraint 2: Only the pins with requirements should be assigned to channels.

$$\forall p \in Pin \bullet P.requirement = \emptyset \Rightarrow P.channel = \emptyset \tag{11}$$

Constraint 3: Channel capabilities must cover pin requirements.

$$\forall p \in Pin \bullet p.requirement.mapping \subseteq p.channel.capability \tag{12}$$

7.2 Structural Constraints in UML

A UML model of this problem is shown in Figure 4. `Design` represents the class of optimization problems discussed in this Section. Each problem (element of class `Design`) is an aggregate of one `UUT` and one or more instances of `ChannelCard` in an FBT (note that an FBT is not represented here explicitly, just its channel cards are). Each `UUT` has a number of instances of `Pin` and each `ChannelCard` has a number of instances of `Channel`. The associations of `Pin` with sets of `PinRequirement` and `Channel` with sets of `Capability` are captured by the associations `requirement` and `capability`, respectively. Moreover, the instances of

`Capability` are also associated with sets of `PinRequirement` via `mapping`. The multiplicity constraints on `requirement` show that a pin can have zero or more requirements, while a requirement can be associated with zero or more pins. A pin can be assigned to at most one channel and a channel can be assigned to at most one pin. Moreover, the multiplicity constraints on the `mapping` and the `capability` associations indicate that each `PinRequirement` must be covered by one `Capability`, although there may be some instances of `Capability` that are not associated with any of the instances of `PinRequirement`. This is possible, since not all of the capabilities of a channel need to be used by pin requirements. Moreover, the same capability may satisfy multiple pin requirements.

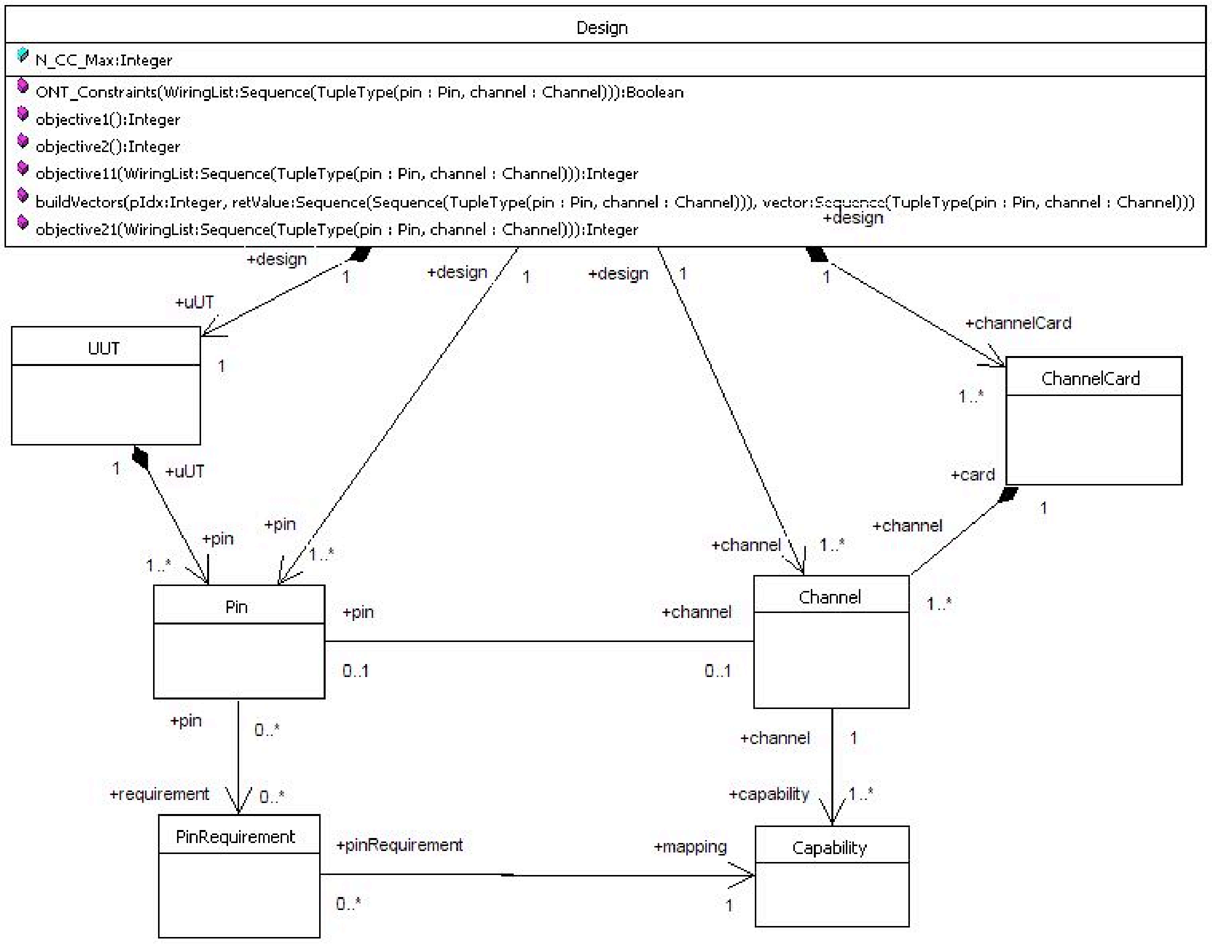

Design
N_CC_Max:Integer
ONT_Constraints(WiringList:Sequence(TupleType(pin : Pin, channel : Channel))):Boolean
objective1():Integer
objective2():Integer
objective11(WiringList:Sequence(TupleType(pin : Pin, channel : Channel))):Integer
buildVectors(pIdx:Integer, retValue:Sequence(Sequence(TupleType(pin : Pin, channel : Channel))), vector:Sequence(TupleType(pin : Pin, channel : Channel)))
objective21(WiringList:Sequence(TupleType(pin : Pin, channel : Channel))):Integer
+design
+design
+design
+design
1
1
1
1
+uUT
UUT
1
+channelCard
ChannelCard
1..*
+card
1
+uUT
1
+pin
+pin
1..*
1..*
+channel
1..*
+channel
1
1..*
Pin
+pin
+channel
Channel
0..1
0..1
+pin
0..*
+channel
1
+requirement
0..*
+capability
1..*
PinRequirement
+pinRequirement
+mapping
Capability
0..*
1

Figure 4: Class Diagram for the Fixture Design Problem

The association `channel` is central to this optimization problem; it is a decision variable that is instantiated as a result of the optimization process. It is shown that each instance of `Pin` is either assigned exactly to one instance of `Channel`, or is not assigned at all. The latter takes place when a pin does not have any requirements, i.e., when the `requirement` association is empty.

7.3 Goals in OCL

The specification of objective functions for this problem is very similar to the specification of objectives for the job scheduling case discussed in Section 3.3. In particular, we re-use the function `buildVectors` to construct a list of all possible assignments `ChannelMatrix` of channels to pins.

Objectives are represented in a similar fashion as in the job scheduling problem, too. First we define `objective11()` that returns the average number of capabilities that the used channels possess.

```
context Design::
objective11(channelVector:Sequence(TupleType(pin:Pin,channel:Channel))):Integer
 def: let WiredPinCount:Integer=channelVector->size()
      let CapabilityCount:Integer=channelVector->iterate(C;AccCap=0 |
         AccCap@pre+C.channel.capability->size())
post: result=CapabilityCount.div(WiredPinCount)
```

Then we define the function `objective1()` that invokes `objective11()`.

```
context Design::objective1():Integer
def: let minObjFunc1Value:Integer=ChannelMatrix->
  iterate(channelVector:Sequence(TupleType(pin:Pin,channel:Channel));minVal:Integer=20 |
   (if ONT_Constraints(channelVector) then
     (if objective11(channelVector)<minVal@pre then objective11(channelVector)
                                              else minVal@pre endif)
                                       else minVal@pre  endif))
 post: result=minObjFunc1Value
```

The specification of the second objective function follows the same pattern. `objective21` iterates over the channels, finds the channel cards that the used channels belong to, and returns the count of channel cards.

```
context Design::
objective21(channelVector:Sequence(TupleType(pin:Pin,channel:Channel))):Integer
 def: let
  UsedChannelCards:Set(ChannelCard)=channelVector->iterate(C; AccCC:Set(ChannelCard)=Set{} |
   AccCC->including(C.channel.card))
post: result = UsedChannelCards->size()
```

`objective2()` returns the value of the second objective function. It is very similar to `objective1()`, so we do not expand the definition here.

7.4 Constraints in OCL

OCL representations of constraints (Eq. 10 through 12) are shown below. They all are defined within the same context:

```
context Design::
ONT_Constraints(channelVector:Sequence(TupleType(pin:Pin,channel:Channel))):Boolean
```

The first constraint states that each UUT pin will be wired to a different channel.

```
let constraint1Value:Boolean = channelVector->forAll(C1,C2 |
 C1.pin<>C2.pin and C1.channel<>C2.channel)
```

The second constraint states that each UUT pin with a pin requirement must be wired to a channel.

```
let constraint2Value:Boolean =  Pins->forAll(P |
 if  P.requirement->notEmpty() then
  channelVector->select(pin = P)->notEmpty()
   else true endif)
```

The third constraint requires that all electrical requirements of a UUT pin need to be satisfied by the wired channel's capabilities.

```
let constraint3Value:Boolean =  Pins->forAll(P |
 P.requirement->forAll(R |
  channelVector->select(pin=P).channel.capability->includes(R.mapping)))
```

These three constraints then need to be connected by the logical `and`s, similarly as for the Job Scheduling case.

8 Evaluation

To assess the behavior of the generated code, test data were generated for each of the two MOCOPs described above. Since the problems are NP-complete [34], not only the generated code had to be tested whether it can find optimal solutions but also to see whether it can recognize that a solution does not exist or that the specific problem instance is "hard". The hardness is defined here as an instance being in the *phase transition* region [16, 35]. Intuitively, when a problem is under-constrained, it is easy to find a solution. If a problem is over-constrained, it is easy to find out that a solution does not exist. The most difficult cases are those that are neither over- nor under-constrained; these cases are in the phase transition region.

The test cases were prepared in such a way as to cover all types: (1) where solutions exist and are easy to find (under-constrained); (2) where solutions don't exist and it is easy to find out about this (over-constrained); (3) phase transition regions where solutions exist; (4) phase transition regions where solutions don't exist. In total, 320 cases were generated: 200 cases for the job scheduling problem and 120 for the fixture design problem.

For comparison, known benchmark programs available in the Mozart Programming System, version 1.3.1, were adopted. Both the benchmark programs and the code generated by our system were run on all of these cases. Since the two types of programs were developed using different approaches, their behaviors were different. The Oz benchmark programs were designed to search for at least one solution to the CSP problem indefinitely. In case it could not find a solution and exhausted its memory limits, it would generate a memory fault event and quit. Our program, on the other hand, was designed to detect that a given problem is in the phase transition region and then quit if the allotted number of iterations is exhausted. If it were able to find a solution before exhausting its time limit, it would provide the best solution found so far. Since the problem of detecting phase transitions is hard, our program would report a phase transition prematurely, when in fact it was not the case. Thus our program's behavior, at times, resulted in *false alarms*.

To compare the performance of the two systems we classify the results into the following five situations:

S/S: A solution exists and both programs find at least one solution.

N/N: A solution does not exist and both programs recognize this fact.

S/PT: A solution exists, but our program reports a phase transition (false alarm).

N/PT: A solution does not exist; the benchmark runs until fault, our program recognizes a phase transition.

PT/PT: It is not known whether a solution exists (it's a hard case); the benchmark runs until fault, our program recognizes a phase transition.

The results of experiments are summarized in Table 1. This table shows the distribution of the test cases for each of the above described classes for the two constraint satisfaction problems. Column 2 shows the number of test cases for each of the two problems. Column 3 shows for how many cases both programs were able to find a solution. Column 4 shows in how many cases the two programs concluded that a solution did not exist. The *S/PT* column shows in how many cases the benchmark programs were able to find a solution, while our program generated a false alarm and gave up on the search. Columns *N/PT* indicates how many of the test cases did not have a solution. Our program was able to recognize this and thus terminated gracefully, while the benchmark programs generated faults. Column *PT/PT* shows the counts of the cases where the programs behaved in the same way as in the former case. Those were the *hard* cases where it is not known whether a solution exists or not.

Problem	*Total*	*S/S*	*N/N*	*S/PT*	*N/PT*	*PT/PT*
Job Scheduling	200	102	20	22	0	56
Fixture Design	120	36	59	12	4	65

Table 1: Summary of types of test cases.

To summarize, the purpose of these tests was not to show superiority of our program over the benchmark programs in terms of performance, but just that our program, which was generated automatically, performs correctly on the easy cases of an NP-complete problem, i.e., that it was able to find solutions. Its behavior was inferior for some of the harder cases (column *S/PT*). However, it was compensated by a better performance of our program for the hard cases (columns *N/PT* and *PT/PT*).

9 Literature Review

A large number of Constraint Programming Languages have been developed [36] which could be selected as the target language for the CSP solver system discussed in this paper. SKETCHPAD [37], CONSTRAINT [38], and ThingLab [39] were some of the earlier languages in this group. Another group of languages can be classified under the label of *constraint logic programming* (CLP), e.g., CHIP [40], CLP($\Re$) [41], Prolog II [42] and Prolog III [43]. Later, the *CLP scheme* [44] generalized the fundamental ideas behind these languages. The scheme could be instantiated to produce a specific constraint logic programming language by defining a constraint system. It was later generalized into the *concurrent constraint* (*CC*) framework of concurrent constraint programming to enable capabilities such as concurrency control, and extensibility to be addressed at the language level [45].

Newer constraint programming languages are AKL [46], cc(FD) [47], CIAO [48], DETAIL [49], Prolog IV [50], Eclipse [51], Oz [52, 29, 53], HAL [54] and SALSA [55]. Their constraint vocabulary and solvers go beyond traditional linear and non-linear constraints and support logical and global constraints. Oz, CIAO, and AKL use the CC framework and implement distributed and concurrent systems. However, these languages mostly target computer scientists and have weaker abstractions for algebraic and set manipulation.

The Helios language [56] and Numerica [57], on the other hand, have been designed to specify and solve non-linear constraint systems using interval analysis techniques while CLP toolkits, like QOCA [58], EaCL [59] and Ultraviolet [60], implement graphical user interfaces to monitor the progress of the constraint solver and provide the user with a mechanism to interact with the solver at run-time.

Mathematical modeling languages are another kind of tool used in optimization. Modeling languages like GAMS [61], Claire [62, 63], CML [64], AMPL [65], XPRESS-MP and Visual Solver [66], provide high-level algebraic and set notations to express mathematical problems that can then be solved using the solvers mentioned above. There is also a new set of optimization programming languages, like OPL [67], and Modeler++ [68], which aim to unify modeling and constraint programming languages.

Today, there are also commercial products, e.g., ILOG (See http://www.kstec.co.kr/), which provide constraint programming and optimization libraries for main stream programming languages like C++, but they are proprietary solutions to which we did not have access.

In summary, although there is a large number of constraint programming languages, none of them provides the interface for expressing constraints like the UML/OCL-based interface discussed in this paper. However, the UML has been used for representing various types of knowledge related to manufacturing. For instance, [69] shows the use of UML for representing domain knowledge for diagnosis, maintenance and repair. UML was proposed as a part of a methodology for modeling supply networks [70]. It was also analyzed as a contributing methodology to a comprehensive modeling framework for networked organizations [71]. The use of a generic CSP for deriving product configurations was presented in [72]. The use of OCL for capturing cardinality constraints of feature models was reported in [73] and were also considered in [74] for representing different views of a system using diagrammatic models. The work that is similar in the approach, although quite different in the goals, was reported in [75, 76]. These papers describe an effort of translating models of software systems into CSPs and then using a CSP solver for the purpose of verification of some *correctness* properties of the models of software systems, such as weak and strong satisability or absence of constraint redundancies. Thus although there is an overlap in terms of the methods used for parsing UML/OCL and translating it into a CSP language, the goals are different. Although we are interested in MOCOPs rather than general software systems, the verification of the correctness properties of MOCOPs might be a useful additional step in the development of the whole solution proposed in this paper.

10 Conclusions

We can draw a number of conclusions from this study. First, it seems rather obvious that UML is a good language for representing the structural aspects of optimization problems. The UML diagrams developed for the two case studies are very easy to interpret and the mapping to the mathematical formalization of the optimization problems is straightforward. A big advantage of UML is that it is standardized and is so widely used in industry. While other approaches to the visualization of optimization problems exist, they are not standard and applicable only to the optimization domain.

Second, we can conclude that OCL can be used to supplement the expressiveness of UML for representing procedural aspects of optimization problems. The expressive power of OCL has been reported in [77] for OCL 1.4 and then in [78] for OCL 2.0. OCL went through a number of modifications, which included both an increase in its expressiveness and a decrease, as well. In [78] it is shown that "pure OCL expressions without auxiliary definitions (as in OCL 2.0) represent exactly the primitive recursive functions and that auxiliary definitions (as in OCL 1.4/5) increase the expressiveness making OCL Turing complete. In [77], OCL is also compared to relational calculus in order to assess its completeness with respect to query expressions. It was concluded that that version of OCL was not powerful enough to denote any query expression of the relational calculus, primarily due to the lack of the concept of "tuple". This was circumvented later by introducing Tuple as a primitive type in the latest versions of OCL.

Table 2 classifies and summarizes the types of constraints and the operations that can be represented using OCL. It shows that OCL is capable of expressing the most commonly used constraints that appear in the formulations of MOCOPs.

Numeric Constraints	*Logical Constraints*	*Set Constraints*	*Numeric Functions*	*Set Functions*
$=$	AND	Member	+	Intersection
$\neq$	OR	NotMember	-	Union
$\geq$	XOR	Subset	*	Product
$\leq$	NOT	AllDisjoint	/	Min/Max
$>$	IMPLIES	allDifferent	%	Count/Sum
$<$			Abs	Indexed access

Table 2: Summary of types of test cases.

In summary, it is save to state that OCL is a very powerful language with respect to both query formulation and expressing constraints. We do not claim, though, that the OCL representation is really simple and easy to write and comprehend. Instead, we show the details of the OCL expressions for two typical multiobjective optimization problems in UML/OCL so that the reader who is interested in pursuing this direction can use these details in his/her considerations.

However, we can claim that it is feasible to develop a generic translator of MOCOPs represented in UML/OCL to CSP solver languages, like Oz. Although we tested this approach only on two cases, the cases were significantly different and yet our translator generated Oz programs that behaved very reasonably as compared to hand-written benchmarks.

Finally, we need to mention that our system has some self-control capabilities to recognize that its search process is in a "hard area" of the search space, but we did not describe this feature in this paper. The self-control mechanisms are rather complicated and thus we intend to describe this aspect in a separate publication.

While working on the specification problems we identified some common patterns even for such diverse problems like the ones used in our study. This suggests that it should be possible to continue this work towards a standard high-level specification language for optimization problems that would be based on UML/OCL, but would also be specialized to this domain. In particular, a UML profile for optimization would be very useful and would help to alleviate some of the difficulties of representing procedural constraints in OCL.

References

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

[2] 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.

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

[4] Z. Xanthopulos, E. Melachrinoudis, and M.M. Solomon. Interactive multiobjective group decision making with interval parameters. *Management Science*, 46(12):1585–1601, December 2000.

[5] A. Kusiak, F. Tang, and G. Xu. Multi-objective optimizationof HVAC system with an evolutionary computation algorithm. *Energy*, 36:2440–2449, 2011.

[6] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2007.

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

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

[9] Karla L. Hoffman. Combinatorial optimization: Current successes and directions for the future. *Journal of Computational and Applied Mathematics*, 124:341–360, December 2000.

[10] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley and Sons., New York, 1990.

[11] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33:60–100, 1991.

[12] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Dover Publications, Inc., 1998.

[13] Z. Song and A. Kusiak. Multiobjective Optimization of Temporal Processes. *IEEE Transactions on Systems, Man and Cybernetics*, 40, no. 3:845–856, 2010.

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

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

[16] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the Really Hard Problems Are. In *Proc. IJCAI91*, volume 1, pages 331–337, 1991.

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

[18] S. Sen. Satisficing models. In *Proceedings of the 1998 AAAI Symposium. Technical Report SS-98-05*, Stanford, California, 1998.

[19] 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.

[20] 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.

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

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

[23] 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, September 2002.

[24] F. Rossi. Constraint (Logic) Programming: A Survey on Research and Applications. In *Proc. ERCIM/Compulog Net workshop on constraints*, pages 40–74. Springer-Verlag, LNAI 1865, 2000.

[25] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2.

[26] OMG. OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2.

[27] OMG. OCLE: Object Constraint Language Environment, version 2.0.4.

[28] C. Schulte and G. Smolka. *Finite Domain Constraint Programming in Oz.* Mozart Documentation, 1.3.1 edition, June 2004.

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

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

[31] W3C. XML Schema.

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

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

[34] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.

[35] J. Christopher Beck and W. Ken Jackson. Constrainedness and the phase transition in job shop scheduling, 1997.

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

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

[38] G.J. Sussman and G.L. Steele. CONSTRAINTS-a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14(1):1–39, 1980.

[39] 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.

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

[41] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In *Fourth International Conference in Logic Programming*, pages 196–218, Melbourne, Australia, May 1987.

[42] A. Colmerauer, H. Kanoui, and M. Van Caneghem. Last steps towards an ultimate prolog. In *Proceedings of the 7th international joint conference on Artificial intelligence - Volume 2*, pages 947–948, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

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

[44] J. Jaffar and J.L. Lassez. Constraint Logic Programming. In *POPl-87*, pages 111–119, Munich,FRG, January 1987.

[45] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the 17th ACM Symposium on Principals of Programming Languages*, pages 232–245, 1990.

[46] S. Haridi and S. Janson. Kernel Andorra Prolog and its computational model. In *ICLP'90*, pages 31–46. MIT Press, 1990.

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

[48] M. Hermenegildo and CLIP group. Some methodological issues in the design of CIAO - a generic, parallel concurrent constraint system. In *Principals and Practice of Constraint Programming (CP97)*, pages 123–133. Springer-Verlag, LNCS 874, 1994.

[49] Hosobe, Satoshi Matsuoka, and Akinori Yonezawa. Generalized local propagation: A framework for solving constraint hierarchies. In *In Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 237–251. Springer-Verlag, 1996.

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

[51] H. El Sakkout and M. Wallace. Probe Backtrack Search for Minimal Perturbation in Dynamic Scheduling. *Constraints*, 5(4):359–388, 2000.

[52] 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.

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

[54] 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.

[55] F. Laburthe and Y. Caseau. SALSA: A Language for Search Algorithms. In *Fourth International Conference on Principals and Practice of Constraint Programming (CP'98)*, pages 310–324, Pisa, Italy, October 1998.

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

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

[58] 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)*, pages 340–354. Springer-Verlag, 1998.

[59] E. Tsang, P. Mills, R. Williams, J. Ford, and J. Borrett. A computer aided constraint programming system. In *PACPL'99*, pages 88–93, 1999.

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

[61] 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)*, pages 107–120. Springer-Verlag, LNCS 1330, 1998.

[62] C. Le Pape and P. Baptiste. A constraint programming library for preemptive and non-preemptive scheduling. In *PACT97*, pages 23–25, 1997.

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

[64] K. Andersson and T. Hjerpe. Modeling constraint problems in CML. In *PACT'98*, pages 295–312, 1998.

[65] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 2003.

[66] 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, page 1. Springer-Verlag, LNCS 1330, 1997.

[67] S. Bistarelli, U. Montanari, and F. Rossi. Constraint solving over semirings. In *Proc. IJCAI95*, pages 624–630. Morgan Kauffman, 1995.

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

[69] I. Rasovska, B. Chebel-Morello, and N. Zerhouni. A mix method of knowledge capitalization in maintenance. *Journal of Intelligent Manufacturing*, 19:347–359, 2008.

[70] D. Stefanovic and N. Stefanovic. Methodology for modeling and analysis of supply networks. *Journal of Intelligent Manufacturing*, 19:485–503, 2008.

[71] L. M. Luis M. Camarinha-Matos and H. Afsarmanesh. A comprehensive modeling framework for collaborative networked organizations. *Journal of Intelligent Manufacturing*, 18:529–542, 2007.

[72] D. Yang and M. Dong. Applying constraint satisfaction approach to solve product configuration problems with cardinality-based configuration rules. *Journal of Intelligent Manufacturing*, 2011.

[73] K. Czarnecki and C. H. P. Kim. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *OOPSLA'05 Workshop on Software Factories*, pages 1–9, 2005.

[74] A. Haug. Managing diagrammatic models with different perspectives on product information. *Journal of Intelligent Manufacturing*, 21:811–822, 2010.

[75] J. Cabot, R. Claris, and D. Riera. UMLtoCSP: A tool for the formal verification of UML/OCL models. In *Proceedings of the 22nd ACM/IEEE International Conference on Automated Software Engineering (ASE'2007)*, pages 547–548. ACM/IEEE, November 2007.

[76] J. Cabot, R. Claris, and D. Riera. Verication of UML/OCL class diagrams using constraint programming. In *ICST Workshop on Model Driven Engineering, Verification and Validation: Integrating Verification and Validation in MDE (MoDeVVa'2008)*, pages 73–80, April 2008.

[77] L. Mandel and M. V. Cengarle. On the expressive power of OCL. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings World Congress on Formal Methods (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 854–874. Springer, 1999.

[78] M. V. Cengarle and A. Knapp. OCL1.4/5 vs. 2.0 expressions: Formal semantics and expressiveness. *Software and Systems Modeling*, 3:9–30, 2004.