Optimisation Modelling for Software Developers

Kathryn Francis, Sebastian Brand, and Peter J. Stuckey

National ICT Australia, Victoria Research Laboratory,
The University of Melbourne, Victoria 3010, Australia
{kathryn.francis,sebastian.brand,peter.stuckey}@nicta.com.au

Abstract. Software developers are an ideal channel for the distribution of Constraint Programming (CP) technology. Unfortunately, including even basic optimisation functionality in an application currently requires the use of an entirely separate paradigm with which most software developers are not familiar.
We suggest an alternative interface to CP designed to overcome this barrier, and describe a prototype implementation for Java. The interface allows an optimisation problem to be defined in terms of procedures rather than decision variables and constraints. Optimisation is seamlessly integrated into a wider application through automatic conversion between this definition and a conventional model solved by an external solver.
This work is inspired by the language CoJava, in which a simulation is automatically translated into an optimal simulation. We extend this idea to support a general interface where optimisation is triggered on-demand. Our implementation also supports much more advanced code, such as object variables, variable-sized collections, and complex decisions.

1 Introduction

This paper is concerned with the usability of Constraint Programming (CP) tools, specifically for general software developers. The reason we have chosen to target general software developers is because they have the potential to pass on the benefits of CP technology to many more end users by incorporating optimisation functionality into application-specific software.

Much of the work on usability for CP is aimed at expert users involved in research or the development of large scale industrial applications. For example, modelling languages such as OPL [12] and MiniZinc [10] are designed to address the requirement of these advanced users to easily experiment with different models and solving strategies, with a fine level of control.

Software developers aiming to incorporate basic optimisation functionality into an application have no such requirement for experimentation and control, as they do not have the expertise to benefit from this. Furthermore, it is highly inconvenient to have to learn and use a separate paradigm in order to implement a single application feature.

We propose an alternative interface to CP technology which hides from the user all reference to CP specific concepts. The problem is defined within the

procedural paradigm, by specifying the procedure used to combine individual decisions and evaluate the outcome. Automatic translation from this definition to a conventional CP model allows the problem to be solved using an external constraint solver. The solution giving optimal values for decision variables is then translated back into a structured representation of the optimal outcome.

This sort of interface would greatly improve the accessibility of CP for general software developers. Procedural programmers are well practiced at defining procedures, so most software developers will find this form of problem definition easy to create and understand. Integration of optimisation functionality into a wider application is also straightforward, as the programmer is freed from the burden of writing tedious and error-prone code to manually translate between two different representations of the same problem (one using types and structures appropriate for the application, and another using primitive decision variables).

This project is inspired by the language CoJava [5] and uses the same core technique to transform procedural code into declarative constraints. We extend significantly beyond CoJava in a number of ways.

2 Background and Related Work

Constraint Programming In order to apply CP techniques to a given satisfaction or optimisation problem, it must first be modeled as a set of constrained decision variables. The model can be defined using a special purpose modelling language, or a CP library embedded within a host programming language; see e.g. [11, 8, 9]. Such libraries typically provide data types to represent decision variables, constraints, the model, and the solver. Although one does not have to use a foreign language, it is necessary to understand CP modelling concepts and to work directly with these to build a declarative model.

Simulation An alternative approach to optimisation is to model the situation using a simulation, and then experiment with parameters, searching for a good selection. This technique, called *simulation optimisation*, treats the simulation as a black box. It is inherently heuristic and thus unable to find provably optimal solutions. For a survey of simulation optimisation, see e.g. [6, 7].

Most research into simulation optimisation assumes that the true objective function is not known: the simulation incorporates some non-determinism and gives a noisy estimate of this function. If the simulation is actually deterministic given a choice of parameters, then it is possible to convert the simulation code into an equivalent constraint model and use this to find a provably optimal set of parameters. Performing this conversion automatically was the goal of CoJava.

CoJava The system described in this paper is inspired by and builds on techniques developed for the language CoJava, introduced in [4] and described further in [5]. CoJava is an extension to the Java language, intended to allow programmers to use optimisation technology to solve a problem modelled as a simulation.

The simulation is written in Java using library functions to choose random numbers, assert conditions which must hold, and nominate a program variable

as the optimisation objective. The CoJava compiler transforms the simulation code into a constraint model whose solution gives a number to return for each random choice so that all assertions are satisfied and the objective variable is assigned the best possible value. The original code is then recompiled with the random choices replaced by assignments to these optimal values. The result is a program which executes the optimal execution path of the original simulation.

CoJava was later extended to SC-CoJava and CoReJava [2, 1, 3], introducing simpler semantics, but also narrowing in focus to supply chain applications.

3 Contributions

Architecture Despite using the same basic technique to translate procedural code into a declarative constraint model, the purpose and architecture of our system are fundamentally different to CoJava.

- Optimisation is triggered explicitly as required during program execution, and problem instance data is not required at compile time. In CoJava, optimisation is implicit and takes place at compile time.
- Performing optimisation at run time allows us to support interactive applications, while CoJava is restricted to simulation-like programs. This is an important distinction, as interactive applications can enable non-technical end users to access optimisation technology independently.
- The code written by the programmer is pure Java and does not take on any new semantics. CoJava on the other hand is an extension of the Java language, with semantics which depend on the mode of compilation.

Supported Code We dramatically expand on the type of code supported, allowing a much more natural coding style.

- We extend support for non-determinism from arithmetic numeric and Boolean types to arbitrary object types.
- We allow variable collections of objects such as sets, lists and maps to be constructed and manipulated.
- We introduce generic higher level decisions to aid modelling, in contrast to the approach taken in SC-CoJava of supplying application-specific library components.

4 A Prototype for Java: the Programmer's Perspective

This paper proposes an alternative interface to CP technology which allows an optimisation problem to be specified through code which constructs a solution using the results of pre-supplied decision making procedures, and code which evaluates that solution, determining whether or not it is valid and calculating a measure of its quality. This idea is applicable to any host programming language,

although the precise design would obviously depend on the features of the language. We describe here a proof-of-concept implementation for Java, consisting of a Java library and a plug-in for the Eclipse IDE.

The library includes only three public classes/interfaces. To include optimisation capability in an application, the programmer implements the Solution interface, defining two methods: build and evaluate. The build method is passed a ChoiceMaker object which provides decision making procedures, and uses these to build a solution. The evaluate method calculates and returns the value of the current solution, or throws an exception if the current solution is invalid.

Optimisation is triggered using the buildMinimal and buildMaximal methods provided by the Optimiser class. These take a Solution object and apparently call its build method with a special ChoiceMaker able to make optimal decisions, so that a subsequent call to evaluate will return the minimal or maximal value without encountering any exceptions. Additional versions of these methods with an extra parameter allow the programmer to request the best solution found within a given time limit.

The Eclipse plug-in performs compile time program manipulation to support run time conversion into a conventional constraint model. This step is performed transparently during program launch, or as an independent operation.

As an illustration of the system we consider a simple project planning application.[1] Input to the program is a list of tasks, each of which may have dependencies indicating other tasks which must be scheduled at least a given number of days earlier. The application chooses a day to schedule each task so that all dependencies are satisfied and the project finishes as early as possible. The generated project plan is displayed to the user, who is then allowed to repeatedly reschedule the project after adjusting the tasks and dependencies. Note that this sort of interactive application cannot be achieved using the CoJava architecture.

The Solution interface is implemented by a class ProjectPlan, whose build and evaluate methods are shown below. A ProjectPlan is initialised with a reference to a list of tasks alltasks. The build method chooses a day for each task and passes this to the task to be recorded. The evaluate method first checks that all task dependencies are satisfied, and then calculates and returns the latest scheduled day as the value of the solution.

```
void build(ChoiceMaker chooser) {
  for(Task task : alltasks) {
    int day = chooser.chooseInt(1, max);
    task.setScheduledDay(day);
  }
}
```

```
Integer evaluate() throws Exception {
  for(Task task : alltasks)
    if(!task.dependenciesSatisfied())
      throw new Exception();
  return getFinishDay();
}
```

An optimal project plan is obtained by passing a ProjectPlan to the buildMinimal method of Optimiser (as we wish to find the solution with smallest evaluation result). After this method is called the tasks currently recorded in tasklist will

[1] Full code for examples is available online: www.csse.unimelb.edu.au/ pjs/optmodel/

have scheduled days corresponding to an optimal project plan. The application displays the plan using the Task objects, and is then free to make changes to the task list and dependencies (based on user input) before calling buildMinimal again, which will update the scheduled days to once again represent an optimal solution given the new tasks and dependencies.

5 A Prototype for Java: Implementation

To solve the optimisation problem specified by the build and evaluate methods of a Solution object, an equivalent constraint model must be constructed and sent to a solver. As the complete problem specification depends on the program state when optimisation is requested, the constraint model cannot be constructed at compile time. Our approach is to generate at compile time a transformed version of the original code which can be used at run time to create a complete model based on the current state.

5.1 Compile Time

The transformed version of the original code works with symbolic expressions rather than concrete values. It builds an expression for the new value of each possibly affected field (thus capturing all possible state changes), and an expression for the value of the solution.

The compilation process is illustrated in Figure 1. Fortunately it is not necessary to create a transformed version of the entire program: only code used within the build and evaluate methods is relevant to optimisation. Transformed versions of the relevant methods are added to the project, and then this generated code is compiled along with the original source code using a regular Java compiler.

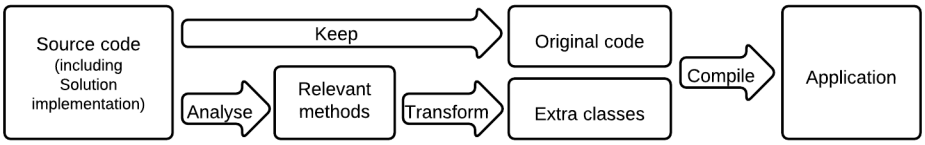

Fig. 1. The compile time operation.

The code transformation is based on that described for CoReJava [3]. The logic of the original code is captured by executing all reachable statements, while ensuring that assignments are predicated on the conditions under which normal execution would reach the assignment. A variable assigned a new value is constrained to equal the assigned value if the *path constraint* holds and the old value otherwise. Pseudo-code showing the code generated for an assignment statement is shown below, where P is the current path constraint, and C and V are respectively the constraints and solver variables collected so far.

	$x' := \text{newVar}();\ V := V \cup \{x'\}$
$x = e;$	$C := C \wedge (x' = \textbf{if}\ P\ \textbf{then}\ e\ \textbf{else}\ x)$
	$x := x'$

Where the original code contains a branch whose condition depends on the outcome of some decision, the transformed code executes both branches.

if(*condition*){	$P_0 := P$
<then-block>	$P := P_0 \wedge \mathit{condition}$
}	<translation of then-block>
else{	$P := P_0 \wedge \neg \mathit{condition}$
<else-block>	<translation of else-block>
}	$P := P_0$

A throw statement introduces a constraint that the current path constraint must evaluate to false (as normal execution is not allowed to reach this point).

We have found it helpful to introduce a separate type to represent variables (local variables, method parameters, and fields), rather than using the expression type directly. The Variable type provides a method to look up an expression for the current value of the variable, and an assign method to replace assignment statements. The types of affected local variables and method parameters are changed to this Variable type, so that assignments can be handled correctly as discussed above. The types of fields are not changed, instead when a field is accessed for the first time a Variable is created and recorded against the object and field name, and future uses of the same field refer to this Variable.

One complication not discussed in the CoJava work is return statements. A return statement provides an expression for the return value of a method, but it also indicates that the remainder of the method should not be executed. To correctly handle methods with multiple return statements, at each return a condition is added to the path constraint for the remainder of the method ensuring that all further assignments are predicated on not reaching this return statement. The condition added is the negation of the part of the current path constraint falling within the scope of the current method. The loop exit statements break and continue are supported using the same technique. Return handling is illustrated by the translation shown below (where a is a field of the current object):

int exampleMethod() {	$P_0 := P$
if(a > 5) {	$P := P_0 \wedge (a > 5);$
return a;	$r := a$ *// introduce variable r for return value*
}	$P := P_0 \wedge \neg(a > 5)$ *// add return constraint*
a = a + 1;	$a' := \text{newVar}();\ V := V \cup \{a'\}$
	$C := C \cup (a' = \textbf{if}\ P\ \textbf{then}\ (a+1)\ \textbf{else}\ a);\ a := a'$
return a;	$r' := \text{newVar}();\ V := V \cup \{r'\}$
	$C := C \cup (r' = \textbf{if}\ P\ \textbf{then}\ a\ \textbf{else}\ r);\ r := r'$
}	$P := P_0;$ **return** r

The transformed versions of affected methods are added to new classes, leaving the original classes unchanged. The transformed methods are made static but are given an extra argument for the object on which the method is called. A further object of type ModelBuilder is threaded through all method calls to keep track of state information. It maintains the path constraint, and also records variables introduced by decision procedures and as intermediate variables, and constraints caused by throw statements or introduced to constrain intermediate variables.

5.2 Run Time

At run time, optimisation is triggered by a call to the Optimiser method buildMinimal or buildMaximal, with an object implementing the Solution interface passed as an argument. The Optimiser first uses reflection techniques to find the transformed build and evaluate methods corresponding to the type of the received Solution object, and then executes these methods.

The variables and constraints recorded in the ModelBuilder during execution of the transformed methods, along with the objective expression returned by the evaluate method, are used to generate a constraint problem which is then sent to an external solver. We chose to express the problem in MiniZinc [10] because it gives us an easy choice of constraint problem solvers.

The ModelBuilder also records a Variable object for each potentially updated field. When a solution is obtained, the optimal decisions are substituted into the expression giving the final value for each of these Variable objects, and reflection techniques are used to update the corresponding fields accordingly. The run time process is illustrated in Figure 2.

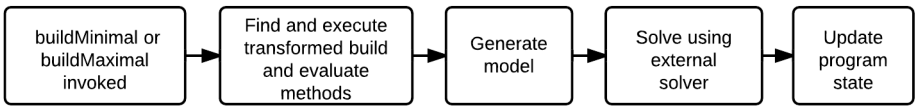

Fig. 2. The optimisation process, triggered on demand during application execution.

Returning to the project planning example, consider the method getFinishDay (Figure 3), which is called by the evaluate method of ProjectPlan. In the transformed version of this method, when getScheduledDay is called on a task the result is an integer expression which is not constant (as the task's scheduled day is assigned in the build method to the result of chooseInt). The greater-than operator is therefore replaced with a method which creates a comparison expression. This means that the if statement has a non-constant condition. As discussed above, the body of the if statement is executed, but the condition (the comparison expression) is added to the current path constraint at the beginning of the then block, and removed at the end.

```
int getFinishDay() {
    int finishDay = 1;
    for(Task task : alltasks)
        if(task.getScheduledDay() > finishDay)
            finishDay = task.getScheduledDay();
    return finishDay;
}
```

Fig. 3. ProjectPlan getFinishDay method, called from within evaluate.

This means that when the assignment of finishDay is reached, the path constraint will not be constant, so finishDay is given a Variable type, and the assign method is used in place of the assignment. The assign method creates a new intermediate variable for the current value of finishDay, and records a constraint that this new expression is equal to the assigned expression (this task's scheduled day) if the path constraint (the greater-than comparison) evaluates to true, and otherwise it is equal to the previous value. The expression finally returned by getLatestDay is an intermediate variable constrained via a series of these varUpdate constraints to equal the latest scheduled day.

The definition of varUpdate, and an example MiniZinc model for a project with 3 tasks is shown in Figure 4. The three day variables in this model are core decision variables introduced by calls to chooseInt. The finishDay variables are intermediate variables used to represent the value stored in the finishDay variable in the getFinishDay method discussed above. The varUpdate constraints also originate from here, while the two constraints enforcing task dependencies originate from the evaluate method, each being the negation of the path constraint when the throw statement was reached.

```
var 1..6: day0;
var 1..6: day1;
var 1..6: day2;
var {1,2,3,4,5,6}: finishDay31;
var {1,2,3,4,5,6}: finishDay33;
var {1,2,3,4,5,6}: finishDay35;
constraint (not (day1 < day0));
constraint (not (day2 < (1 + day1))) /\ (not (day2 < (2 + day0)));
constraint varUpdate(finishDay31,(day0 > 1),day0,1);
constraint varUpdate(finishDay33,(day1 > finishDay31),day1,finishDay31);
constraint varUpdate(finishDay35,(day2 > finishDay33),day2,finishDay33);
solve :: int_search([day0, day1, day2], input_order, indomain_split, complete)
        minimize finishDay35;

predicate varUpdate(var int: out, var bool: update, var int: new, var int: old) =
    (out = [old,new][1+bool2int(update)]);
```

Fig. 4. MiniZinc model for project planning example.

5.3 Refining the Translation

Many values computed within the build and evaluate methods are unaffected by the results of decisions. Measures to take this into account can reduce unnecessary complexity in the model. At compile time, translation is only required for a method if at some time it is passed a non-constant argument, or if the code within the method uses decision procedures provided by the ChoiceMaker, changes the state of some object, or reads a field which is updated elsewhere in translated code. Within a method, a variable that is never assigned to a value that depends on the outcome of decisions, and is never assigned conditionally depending on the outcome of a decision, does not need its type changed.

At run time, if the condition for a branching statement is constant, only the corresponding branch is executed. Also, if the current path constraint is constant, an assignment can overwrite the previous value of a variable unconditionally. Actually, it is only required that the part of the path constraint which falls within the variable's scope is constant. If the part of the path constraint outside this scope is not satisfied, then normal execution would never reach the declaration of the variable, making its value irrelevant.

An example of this is illustrated in the translation below:

if(X) {	$P_0 := P$
	$P := P_0 \wedge X$
int $sum = 0$;	$sum := 0$
for(**int** $item$: list) {	**for** $item$ in list **do**
$sum = sum + item$;	$sum' :=$ newVar(); $V := V \cup \{sum'\}$
}	$C := C \cup (sum' = sum + item)$; $sum := sum'$
$a = sum$;	$a' :=$ newVar(); $V := V \cup \{a'\}$
	$C := C \cup (a' =$ **if** P **then** sum **else** $a)$; $a := a'$
}	$P := P_0$

The Boolean expression X is added to the path constraint for the duration of the then block. The assignment to a (which is declared outside the if statement) is conditional on X, but all assignments to sum are unconditional.

6 Object Variables

For real-world problems, natural code will almost always involve decisions at the object level. We describe here our support for this, including a procedure to choose one object from a collection. Note that this is a major extension over CoJava, which restricts non-determinism to primitive typed variables only.

Let us consider a new version of the project planning application. This time, resources are no longer infinite. Instead only a given number of hours are available on each day. We introduce a Day class, with fields recording the day number and the maximum number of hours available, as well as the number of hours currently assigned. Each Task now also has a duration, and we need to ensure that the total duration of all tasks assigned to a day does not exceed the hours available.

The new build method for ProjectPlan is shown below. For each task a Day object is chosen, and the task is assigned to this Day using the addTask method. This method updates the hours assigned to the day, returning false if the total is now greater than the available hours. If the return value is false an exception is thrown to indicate that the solution is not acceptable.

```
public void build(ChoiceMaker chooser) throws Exception {
    for(Task task : alltasks) {
        Day chosenDay = chooser.chooseFrom(allDays);
        if(!chosenDay.addTask(task))
            throw new Exception("Failed to add task");
    }
}
```

The object represented by chosenDay depends on the outcome of the chooseFrom decision. This means that in the transformed version of the code we need to be able to call addTask without knowing which Day object is the target.

In order to support this sort of code, we first need to be able to represent the choice of an object using primitive solver variables. This is achieved by assigning an integer key to each distinct object. Each expression with a non-primitive type has a domain of possible objects, which can be translated into a corresponding integer domain. This representation allows straightforward equality comparisons between object expressions. The chooseFrom method simply creates a new expression whose domain is given by the provided collection, with a corresponding integer decision variable.

The next consideration is field accesses. As with single objects, each accessed field of an object expression is assigned a Variable object. This Variable must be able to produce an expression for the current value of the field, and to accept an expression for a newly assigned value. For object expressions, a special type of Variable is used which has a reference to the Variable for the corresponding field of each object possibly represented by the expression, as well as an integer expression for the index of the chosen object.

When an expression for the current value is requested, an intermediate solver variable is created and constrained to equal the current field expression for the object at the chosen index. This is a simple element constraint.

An assignment must update the corresponding field for every object in the expression's domain. A new intermediate variable is created for each, and a user-defined predicate fieldUpdate is used to ensure that all except the one at the correct index are equal to the previous values, while the one at the correct index is equal to the assigned expression if the current path constraint holds, or the old value otherwise.

We also need to handle the calling of methods on object expressions. Fortunately, the only way the target object affects the outcome of a method invocation is through the values stored in its fields. This means that all uncertainty can be pushed down to the field level. Translated methods are already converted to be static with an argument indicating the object called on, so it is straightforward

to allow this argument to be non-constant. Then, for each field access the Variable retrieved is of the special type discussed above.

7 Variable Collections

Combinatorial problems commonly involve collections such as sets or lists. It is therefore valuable to allow the use of these in the Solution code. It should be possible not only to store variable objects in collections, but also to make arbitrary changes to the collection in variable contexts, so that the resulting size and composition of the collection depends on the outcome of decisions. Furthermore, it should be possible to iterate over these variable collections.

Returning to the project planning example, imagine we are now allowed to hire an external contractor to perform some tasks, paying an hourly rate plus a callout fee for each day the contractor's services are required. Instead of finishing as early as possible we wish to minimise cost while meeting a deadline. Extracts from the revised build and evaluate methods for ProjectPlan are shown in Figure 5.

```
Day day = chooser.chooseFrom(days);
if(chooser.chooseBool()) {
    contractedTasks.add(task);
    contractorDays.add(day);
}
```

```
int cost = dayFee * contractorDays.size();
for(Task t : contractedTasks) {
    cost += hourlyRate * t.duration();
}
return cost;
```

Fig. 5. Extracts from revised build and evaluate methods making use of collections.

Note first that the chosen day for the task, a decision variable, is added to the contractorDays set. Second, this set and the list of contracted tasks are both updated conditionally depending on whether or not this task is to be contracted out (as decided by the chooseBool method). The crucial aspect here is that the for loop in the evaluate code iterates over a collection whose size depends on the values of decision variables.

We have implemented a special purpose translation for the Set, List and Map interfaces in order to support this kind of code. The VariableSet, VariableList and VariableMap classes provide specialised transformed versions of (almost) all methods included in these collection interfaces, using a special-purpose representation for the state of the collection.

7.1 Sets, Lists and Maps

The VariableSet class represents a set using a list of possible members of the set, and a corresponding list of Boolean expressions indicating whether or not each item is actually in the set. Each possible item is also an expression which may represent a choice between several actual objects. When a VariableSet is

initialised, all items are constant expressions, and the Boolean conditions are true. It is only through operations on the set that variability is introduced.

As an example of an operation we consider add. This method is required to add the given item to the set if it is not already present, and return true if the set has changed. The pseudo-code below defines the effect of calling add on a set s having current state $vs = \langle n, x_{1..n}, c_{1..n} \rangle$, where n is the number of possible members of the set, x_i is the possible member at index i, and c_i is the Boolean condition indicating whether or not x_i is actually in the set.

boolean *result* = s.add(y);

$a := \bigwedge_{i=1..n} \neg(x_i = y \wedge c_i)$
$b := P \wedge a$ // *P: path constraint*
$vs := \langle n + 1, \langle x_{1..n}, y \rangle, \langle c_{1..n}, b \rangle \rangle$
result := a

The Boolean expression a represents the condition that y was not already in the set. This expression is also used as the return value. It is possible for the set to remain unchanged even if y was not already present, as the current path constraint may not be satisfied. However, in this case execution would not reach this method call, so the return value is irrelevant.

Clearly it is important to handle constants well, as otherwise the model becomes unnecessarily complex. For example, the add method does not add a new possible item if one of the existing possible items is identical to the added item. In this case the existing item's Boolean condition is updated instead, and if this is already constant and true, no change is required.

The VariableList class maintains a list of possible members in the same fashion as VariableSet, but instead of a Boolean expression indicating whether or not the item is present, an integer expression for each possible item indicates its (0-based) index in the list, with all items not actually in the list having indices greater than the length of the list. A separate integer expression is maintained for the current length of the list.

The VariableMap class is implemented as an extension of the VariableSet class, with an added expression for each possible key giving its currently assigned value.

7.2 Iteration

Iteration in Java is performed using the Iterator interface, with two methods: hasNext to check whether there are remaining items, and next to retrieve the next item. We support iteration over variable collections using a VariableIterator class which implements the transformed versions of these operations. That is, the hasNext method returns a Boolean expression which evaluates to true if at least one of the remaining items is actually in the collection, while the next method returns an expression for the next item.

Enhanced for loops (for example that on the right of Figure 5) are converted into the equivalent while loop using an explicit iterator. At the beginning of each loop iteration the hasNext method is called, and the resulting Boolean expression is added to the path constraint, to be removed at the end of the loop body. When

the hasNext method returns an expression which is constant and false (as it does when it runs out of possible members of the collection), the loop is terminated.

Although the VariableSet state includes a list of expressions for possible members of the set, the iterator cannot simply return these in order. To correctly reflect loop exit logic all items which are actually in the set must be returned before any which are not. For this reason, each item returned by the VariableIterator is actually a new expression which may represent any of the possible members of the set. An integer variable is created for each returned item, giving the corresponding index into the list of possible members. These indices are constrained to ensure that an item which is not in the set is never returned before an item which is in the set, and further (to avoid symmetry) that within these two groups items are returned in order of index.

Iteration over lists is implemented similarly except that the order in which items are returned is determined by the indices stored as part of the VariableList state. For both lists and sets, the Boolean expression returned by hasNext is simply a comparison between the number of items already returned and an expression for the size of the collection.

Obviously constant detection is very important to avoid excessive complexity. We have implemented some simplifications, such as returning all items which are definitely in the set first, and excluding entirely any which are definitely not in the set, but have not yet investigated all possible simplifications.

8 Complex Decisions

With support for variable collections, it becomes possible to provide more complex decision procedures allowing decisions to be specified at a higher level. As an illustration, let us return to the project planning example. Imagine that instead of choosing a single worker for each task, we assign a team of workers. For each task there are a set of allowed team sizes, and the task duration varies according to the size chosen. Furthermore, tasks may be performed across multiple days, as long as an integral number of hours is assigned to each day and the total number of hours matches the task duration.

Below is a build method appropriate for this situation, making use of complex decision procedures.

```
public void build(ChoiceMaker chooser) {
    for(Task task : allTasks) {
        int teamSize = chooser.chooseFrom(task.allowedTeamSizes());
        int taskDuration = task.getDuration(teamSize);
        Set<Worker> team = chooser.chooseSubset(allWorkers, teamSize);
        Map<Day,Integer> chosenDays = chooser.allocate(taskDuration, allDays);
        for(Worker worker : team)
            worker.assignTask(task, chosenDays);
    }
}
```

The procedure first chooses a team size and a corresponding team. Then the total task duration is allocated to days using the allocate method. This method decides how much of the given quantity (in this case the task duration) should be allocated to each object in the given collection (in this case the list of days). Finally, the workers' schedules are updated appropriately.

Not only do complex decision procedures greatly simplify the build method, they also provide opportunities to make use of global constraints. For example, the allocate method can make use of a global sum constraint to ensure that the total quantity allocated is correct.

9 Experimental Results and Future Work

Having developed a working system, our next concern is performance. We present in Table 1 preliminary experimental results demonstrating the relative performance of the system compared with equivalent hand-written models. It is unrealistic to expect that automatically generated models will be able to compete with models produced by an expert. Our aim is to be able to handle problems arising for small businesses or individuals. The results show that we still have some work to do to achieve this goal. Note that compilation time is not shown as this was insignificant: the entire suite compiles in 20 seconds (with around 15 seconds spent performing code transformation).

For most problems the vast majority of the total time is spent solving the model (rather than generating it). This suggests the potential to greatly decrease the running time by improving the model. Our initial analysis of the automatically generated models has led to the identification of two easily detected programming patterns for which stronger constraints are available. The table below shows the original Java code for each of these patterns, the constraints which would be generated using the standard transformation, and the alternative stronger constraints.

Java code	**Standard translation**	**Specialised translation**
if(c) x++;	var x' = [x, x+1][bool2int(c)+1];	var x' = x + bool2int(c);
if(a > x) x = a;	var x' = [x, a][bool2int(a > x)+1];	var x' = max(x, a);

After adding a step during the model generation phase to automatically detect just these two programming patterns and replace the constraints, we have observed a significant improvement in performance on several benchmarks. We anticipate further gains can be made by identifying other patterns for which straightforward model refinements are beneficial. Some patterns may involve larger portions of the code, for example a loop which sums a collection of values or counts the number of objects satisfying some property.

A complementary approach is to attempt to exploit the structure of the generated models, which tend to be quite unidirectional. We may be able to take advantage of this property for efficient propagation scheduling or through the development of a specialised search strategy.

Other future work could design global constraints to improve the treatment of collection operations as well as variable and field updates. There is also room

for further exploration of alternative representations which could be used for the state of variable collections. The current implementations are correct but certainly not as efficient as possible.

Problem	Size	Total	Solving	Improved	Hand	Factor
project planning 1	250 tasks	1.13	78.1%	0.69 (39%)	0.16	4.4
project planning 2	18 tasks	16.22	97.6%	13.31 (18%)	1.18	11.3
project planning 3	14 tasks	32.38	96.6%	26.38 (19%)	0.27	96.9
bin packing	8 items	8.67	98.2%	5.81 (33%)	0.34	17.2
golomb ruler	7 ticks	3.45	33.6%	3.41 (0.9%)	2.03	1.7
knapsack (0-1)	30 items	1.97	80.4%	1.95 (0.9%)	0.15	13.2
knapsack (bounded)	30 items	6.40	95.2%	6.36 (0.6%)	1.10	5.8
routing (pickup-del)	8 stops	6.48	96.5%	6.45 (0.6%)	0.33	19.7
social golfers	9 golfers	2.65	75.0%	2.60 (1.9%)	0.14	18.3
talent scheduling	8 scenes	39.47	99.7%	20.87 (47%)	2.20	9.5

Table 1. Experimental results for project planning example and various well-known problems. In order the figures give the total time (secs) for the optimisation step, the percentage of this total used by the solver, the new total time using basic model refinement with percentage improvement in brackets, the solving time for an equivalent hand-written model, and the number of times faster this hand-written model is compared with the improved total time. Timing figures are the average over 30 instances.

10 Conclusion

If software developers with no specialised CP expertise could easily incorporate CP technology into their applications, this would greatly increase its impact.

We have designed an alternative interface to CP which aims to be more intuitive and convenient for software developers. The necessary translation between paradigms is automated, allowing the programmer to work exclusively with a procedure based definition of the optimisation problem. This significantly reduces the burden on the programmer and allows straightforward integration of optimisation functionality within a wider application.

A natural coding style is allowed with support for object variables, variable collections, and high level decision procedures, building on and significantly extending techniques used to implement the language CoJava.

Preliminary experiments show that further work is required to achieve our goal of performance sufficient for small business and personal applications, but that there are gains to be made using very simple local adjustments to the model. We have also identified several other avenues of investigation which may lead to further improvements in performance.

Acknowledgments NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

References

1. M. Al-Nory and A. Brodsky. Unifying simulation and optimization of strategic sourcing and transportation. In *Winter Simulation Conference (WSC)*, pages 2616–2624, 2008.
2. A. Brodsky, M. Al-Nory, and H. Nash. Service composition language to unify simulation and optimization of supply chains. In *Hawaii International Conference on System Sciences*, page 74, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
3. A. Brodsky, J. Luo, and H. Nash. CoReJava: learning functions expressed as Object-Oriented programs. In *Machine Learning and Applications*, pages 368–375, Los Alamitos, CA, USA, 2008. IEEE Computer Society.
4. A. Brodsky and H. Nash. CoJava: a unified language for simulation and optimization. In *Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 194–195, New York, NY, USA, 2005. ACM.
5. A. Brodsky and H. Nash. CoJava: optimization modeling by nondeterministic simulation, in constraint programming. In *Principles and Practice of Constraint Programming (CP)*, pages 91–107, 2006.
6. Y. Carson and A. Maria. Simulation optimization: methods and applications. In *Winter Simulation Conference (WSC)*, pages 118–126, Atlanta, Georgia, United States, 1997. IEEE Computer Society.
7. M. C. Fu, F. W. Glover, and J. April. Simulation optimization: a review, new developments, and applications. In *Winter Simulation Conference (WSC)*, 2005.
8. E. Hebrard, E. O'Mahony, and B. O'Sullivan. Constraint programming and combinatorial optimisation in Numberjack. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, pages 181–185, 2010.
9. N. Jussien, G. Rochart, and X. Lorca. The CHOCO constraint programming solver. In *CPAIOR08 Workshop on OpenSource Software for Integer and Constraint Programming OSSICP08*, 2008.
10. N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. *Principles and Practice of Constraint Programming (CP)*, pages 529–543, 2007.
11. C. Schulte, M. Lagerkvist, and G. Tack. GECODE – an open, free, efficient constraint solving toolkit. http://www.gecode.org/.
12. P. Van Hentenryck, L. Michel, L. Perron, and J.-C. Régin. Constraint programming in OPL. *Principles and Practice of Declarative Programming (PPDP)*, pages 98–116, 1999.