Modelling Destructive Assignments

Kathryn Francis[1,2], Jorge Navas[2], and Peter J. Stuckey[1,2]

[1] National ICT Australia, Victoria Research Laboratory
[2] The University of Melbourne, Victoria 3010, Australia

Abstract. Translating procedural object oriented code into constraints is required for many processes that reason about the execution of this code. The most obvious is for *symbolic execution* of the code, where the code is executed without necessarily knowing the concrete values. In this paper, we discuss translations from procedural object oriented code to constraints in the context of solving optimisation problems defined via simulation. A key difficulty arising in the translation is the modelling of state changes. We introduce a new technique for modelling destructive assignments that outperforms previous approaches. Our results show that the optimisation models generated by our technique can be as efficient as equivalent hand written models.

1 Introduction

Symbolic reasoning has been the crux of many software applications such as verifiers, test-case generation tools, and bug finders since the seminal papers of Floyd and Hoare [6, 8] in program verification and King [9] in symbolic execution for testing. Common to these applications is their translation of the program or some abstraction of it into equivalent *constraints* which are then fed into a *constraint solver* to be checked for (un)satisfiability.

The principal challenge for this translation is effective handling of destructive state changes. These both influence and depend on the flow of control, making it necessary to reason disjunctively across possible execution paths. In object oriented languages with field assignments, the disjunctive nature of the problem is further compounded by potential aliasing between object variables.

In this paper we introduce a new, *demand-driven* technique for modelling destructive assignments, designed specifically to be effective for the difficult case of field assignments. The key idea is to view the value stored in a variable not as a function of the current state, but as a function of the *relevant assignment statements*. This allows us to avoid maintaining a representation of the entire program state, instead only producing constraints for expressions which are actually required.

The particular application we consider for our new technique is the tool introduced in [7], which aims to provide Java programmers with more convenient access to optimisation technology. The tool allows an optimisation problem to be expressed in simulation form, as Java code which computes the objective value given the decisions. This code could be used directly to solve the optimisation problem by searching over possible combinations of decisions and comparing the computed results, but this is likely to be very inefficient. Instead, the tool in [7] translates the simulation code into constraints, and then uses an *off-the-shelf* CP solver to find a set of decisions resulting in the optimal return value.

Experimental results using examples from this tool demonstrate that our new technique for modelling destructive assignments is superior to previous approaches, and can produce optimisation models comparable in efficiency to a simple hand written model for the same problem.

1.1 Running Example

As a running example throughout the paper we consider a smartphone app for group pizza ordering. Each member of the group nominates a number of slices and some ingredient preferences. The app automatically generates a joint order of minimum cost which provides sufficient pizza for the group, assuming that a person will only eat pizza with at least one ingredient they like and no ingredients they dislike. After approval from the user, the order is placed electronically.

Our focus is on the optimisation aspect of the application: finding the cheapest acceptable order. We assume that for each type of pizza both a price per pizza and a price per slice is specified. The order may include surplus pizza if it is cheaper to buy a whole pizza than the required number of individual slices.

Figure 1 shows a Java method defining this optimisation problem, called buildOrder. The problem parameters are the contents of the people list and the details stored in the menu object when buildOrder is called. Each call to the method choiceMaker.chooseFrom indicates a decision to be made, where the possible options are the OrderItem objects included in the list pizzas (the Order constructor creates an OrderItem for each pizza on the menu, all initially for 0 slices). The objective is to minimise the return value, which is the total cost of the order.

1.2 Translating Code into Constraints

To evaluate different possible translations from procedural code to constraints we use examples from the tool in [7]. This tool actually performs the translation on demand at run-time (not as a compile time operation), which complicates the translation process somewhat. For the purpose of this paper we will ignore such implementation details, using the following abstraction to simplify the description of the different translations.

We consider the translation to be split into two phases. In the first phase the code is flattened into a linear sequence of assignment statements, each of which has some conditions attached. We describe this transformation briefly in Section 2. In the second phase, which is the main focus of the paper, the flattened sequence of assignments is translated into constraints.

2 Flattening

In the Java programming language only the assignment statement changes the state of the program. All other constructs simply influence which other statements will be executed. It is therefore possible to emulate the effect of a piece of

```
int buildOrder() {
  order = new Order(menu);
  for(Person person : people) {
    // Narrow down acceptable pizzas
    pizzas.clear();
    for(OrderItem item : order.items)
      if(person.willEat(item))
        pizzas.add(item);
    // Choose from these for each slice
    for(int i = 0; i < person.slices; i++) {
      OrderItem pizza =
        choiceMaker.chooseFrom(pizzas);
      pizza.addSlice();
  } }
  return order.totalCost();
}

class Order {
  List<OrderItem> items;
  int totalCost() {
    int totalcost = 0;
    for(OrderItem item : items)
      totalcost += item.getCost();
    return totalcost;
  }
}
```

```
class OrderItem
{
  int pizzaPrice;
  int slicePrice;
  int fullPizzas = 0;
  int numSlices = 0;

  void addSlice() {
    numSlices = numSlices + 1;
    if(numSlices == slicesPerPizza) {
      numSlices = 0;
      fullPizzas = fullPizzas + 1;
  } }

  int getCost() {
    int cost = fullPizzas * pizzaPrice;
    if(numSlices > 0) {
      int slicesCost =
        numSlices * slicePrice;
      if(slicesCost > pizzaPrice)
        slicesCost = pizzaPrice;
      cost = cost + slicesCost;
  } }
}
```

Fig. 1: A Java simulation of a pizza ordering optimisation problem.

Java code using a sequence of assignment statements, each with an attached set of conditions controlling whether or not it should be executed. The conditions reflect the circumstances under which this statement would be reached during the execution of the original code.

The flattening process involves unrolling loops[3], substituting method bodies for method calls, and removing control flow statements after adding appropriate execution conditions for the child statements. As an example, consider the method getCost shown in Figure 1. To flatten an if statement we simply add the if condition to the execution conditions of every statement within the then part. The body of getCost can be flattened into the following sequence of conditional assignment statements.

	Conditions		Variable		Assigned Value
1.			cost	:=	fullPizzas × pizzaPrice
2.	(numSlices > 0)	:	slicesCost	:=	numSlices × slicePrice
3.	(numSlices > 0, slicesCost>pizzaPrice)	:	slicesCost	:=	pizzaPrice
4.	(numSlices > 0)	:	cost	:=	cost + slicesCost

[3] The tool in [7] only supports loops with exit conditions unaffected by the decisions, or iteration over bounded collections. This means the number of loop iterations is always bounded. For unbounded loops partial unrolling can be performed, and the final model will be an under-approximation of the behaviour of the program.

Note that each assignment statement applies to a specific variable. This may be a local variable identified by name (as above), or an object field o.f where o is a variable storing an object, and f is a field identifier. We call an assignment to an object field a *field assignment*. The value of the object variable o may depend on the decisions, so the concrete object whose field is updated by a field assignment is not necessarily known.

An important optimisation is to consider the declaration scope of variables. For example, if a variable is declared inside the then part of an if statement (as is the case for the slicesCost variable above), assignments to that variable need not depend on the if condition. In any execution of the original code where the if condition does not hold, this variable would not be created, and therefore its value is irrelevant. This means assignments 2 and 3 above do not need the condition numSlices > 0.

We also need to record the initial program state. For variables which exist outside the scope of the code being analysed, we add an unconditional assignment at the beginning of the list setting the variable to its initial value. We call this an *initialising assignment*. For object fields we add an initialising assignment for each concrete object.

Figure 2 shows the sequence of assignments produced by flattening our example function buildOrder for an instance with two people and three pizza types. Note that calls to ChoiceMaker methods are left untouched (these represent the creation of new decision variables), and expressions which do not depend on the decisions are calculated upfront. For example, the code used to find acceptable order items for each person does not depend on any decisions, so rather than including assignments originating in this part of the code in the flattened sequence, we simply calculate these lists and then use them as constants. Where these expressions are used as if conditions or loop exit conditions we exclude from the translation any unreachable code.

In the following sections, we assume our input is this flattened list of conditional assignment statements. We also use the notation $Dom(v, i)$ to refer to the set of possible values for variable v at (just before) assignment i. This is easily calculated from the list of assignments. A conditional assignment adds values to the domain of the assigned-to variable, while an unconditional assignment replaces the domain.

3 Modelling Assignments: Existing Techniques

Using the flattening transformation described above and a straightforward translation of mathematical and logical expressions, we reduce the problem of representing Java code by constraints to that of modelling (conditional) assignment statements. In this section we describe two existing approaches to this, while in the next section we introduce a new proposed approach.

3.1 Typical CP Approach

One obvious technique for modelling assignments, and that used in [7, 2, 4], is to create a new version of the assigned-to variable for each assignment, and then

	Cond		Object		Field/Var		Assigned Value
1.			Veg	.	fullPizzas	:=	0
2.			Marg	.	fullPizzas	:=	0
3.			Mush	.	fullPizzas	:=	0
4-12.	***other initialisation assignments (for numSlices, pizzaPrice, slicePrice)***						
13.					pizzas1	:=	[Veg,Marg]
14.					pizza1	:=	chooseFrom(pizzas1)
15.			pizza1	.	numSlices	:=	pizza1.numSlices + 1
16.					b1	:=	pizza1.numSlices == slicesPerPizza
17.	(b1)	:	pizza1	.	numSlices	:=	0
18.	(b1)	:	pizza1	.	fullPizzas	:=	pizza1.fullPizzas + 1
19-23.	***repeat assignments 14-18 for 2nd slice (using vars pizza2 and b2)***						
24.					pizzas2	:=	[Marg,Mush]
25.					pizza3	:=	chooseFrom(pizzas2)
26.			pizza3	.	numSlices	:=	pizza3.numSlices + 1
27.					b3	:=	pizza3.numSlices == slicesPerPizza
28.	(b3)	:	pizza3	.	numSlices	:=	0
29.	(b3)	:	pizza3	.	fullPizzas	:=	pizza3.fullPizzas + 1
30-34.	***repeat assignments 25-29 for 2nd slice (using vars pizza4 and b4)***						
35-39.	***repeat assignments 25-29 for 3rd slice (using vars pizza5 and b5)***						
40.					totalcost	:=	0
41.					cost1	:=	Veg.fullPizzas × Veg.pizzaPrice
42.					b6	:=	Veg.numSlices > 0
43.					slicesCost1	:=	Veg.numSlices × Veg.slicePrice
44.					b7	:=	slicesCost1>Veg.pizzaPrice
45.	(b7)	:			slicesCost1	:=	Veg.pizzaPrice
46.	(b6)	:			cost1	:=	cost1 + slicesCost1
47.					totalcost	:=	totalcost + cost1
48-54.	***repeat assignments 41-47 for 2nd order item (Marg)***						
55-61.	***repeat assignments 41-47 for 3rd order item (Mush)***						
62.					objective	:=	totalcost

Fig. 2: Flattened version of buildOrder method. We assume an instance where the menu lists three different types of pizza (vegetarian, margharita and mushroom), meaning the order will contain three OrderItems [Veg, Marg, Mush], and where the people list contains two Person objects, the first willing to eat vegetarian or margharita and requiring two slices, and the second willing to eat margharita or mushroom and requiring three slices. The b variables have been introduced to store branching conditions. Variables from methods called more than once and those used as the iteration variable in a loop are numbered to distinguish between the different versions.

use the latest version whenever a variable is referred to as part of an expression. If the assignment has some conditions, the new version of the variable can be constrained to equal either the assigned value or the previous version, depending on whether or not the conditions hold. This is easily achieved using a pair of implications, or alternatively using an element constraint with the condition as the index. The element constraint has the advantage that some propagation is possible before the condition is fixed, so we will use this translation.

The constraint arising from a local variable assignment is shown below, where localvar0 is the latest version of localvar before the assignment, and localvar1 is the new variable that results from the assignment, which will become the new latest version of localvar. Note that we assume arrays in element constraints are indexed from 1. For convenience, in the rest of the paper we use a simplified syntax for these constraints (also shown below).

assignment:	condition : localvar := expression
constraint:	element(bool2int(condition)+1, [localvar0, expression], localvar1)
simple syntax:	localvar1 = [localvar0, expression][condition]

This translation is only correct for local variables. Field assignments are more difficult to handle due to the possibility of aliasing between objects. However, if the set of concrete objects which may be referred to by an object variable is finite (which is the case for our application), then it is possible to convert all field assignments into equivalent assignments over local variables, after which the translation above can be applied.

For each concrete object, a local variable is created to hold the value of each of its fields. In the following we name these variables using the object name and the field name separated by an underscore. Then every field assignment is replaced by a sequence of local variable assignments, one for each of the possibly affected concrete objects. These new assignments retain the original conditions, and each also has one further condition: that its corresponding concrete object is the one referred to by the object variable. Where necessary to avoid duplication, an intermediate variable is created to hold the assigned expression.

An example of this conversion is shown below, where we assume the assignment is on line n and $Dom(\textsf{objectvar}, n) = \{\textsf{Obj1}, \textsf{Obj2}, \textsf{Obj3}\}$.

field assignment:	condition : objectvar.field := expression
assignments:	condition $\wedge$ (objectvar = Obj1) : Obj1_field := expression
	condition $\wedge$ (objectvar = Obj2) : Obj2_field := expression
	condition $\wedge$ (objectvar = Obj3) : Obj3_field := expression

The final requirement is to handle references to object fields. We need to look up the field value for the concrete object corresponding to the current value of the object variable. To achieve this we use a pair of element constraints sharing an index as shown below, where fieldrefvar is an intermediate variable representing the retrieved value. We assume the same domain for objectvar.

field reference:	objectvar.field
constraints:	element(indexvar, [Obj1,Obj2,Obj3], objectvar)
	element(indexvar, [Obj1_field,Obj2_field,Obj3_field], fieldrefvar)

In summary, this approach involves two steps. First the list of assignments is modified to replace field assignments with equivalent local variable assignments, introducing new variables as required. Then the new list (now containing only local variable assignments) is translated into constraints, with special handling for field references. This approach is quite simple, but can result in a very large model if fields are used extensively. To see the result of applying this translation to a portion of our running example, see Figure 3(a).

3.2 Typical SMT Approach

One of the main reasons for the significant advances in program symbolic reasoning (e.g. verification and testing) during the last decade has been the remarkable progress in modern SMT solvers (we refer the reader to [1] for details).

When using SMT, local variable assignments can be translated in the same way as for the CP approach (adding a new version of the variable for each assignment), but using an if-then-else construct (ite below) instead of an element constraint.

assignment: condition : localvar := expression
formula: localvar1 = ite(condition, expression, localvar0)

For field assignments, it is more convenient to use the theory of arrays. This theory extends the theory of uninterpreted functions with two interpreted functions *read* and *write*. McCarthy proposed [10] the main axiom for arrays:

$$\begin{aligned}
&\forall a, i, j, x \text{ (where } a \text{ is an array, } i \text{ and } j \text{ are indices and } x \text{ is a value)}\\
&\qquad i = j \rightarrow \mathit{read}(\mathit{write}(a, i, x), j) = x\\
&\qquad i \neq j \rightarrow \mathit{read}(\mathit{write}(a, i, x), j) = \mathit{read}(a, j)
\end{aligned}$$

Note that since we are not interested in equalities between arrays we only focus on the non-extensional fragment.

Following the key idea of Burstall [3] and using the theory of arrays, we define one array variable for each object field. Conceptually, this array contains the value of the field for every object, indexed by object. Note however that there are no explicit variables for the elements.

An assignment to a field is modelled as a write to the array for that field, using the object variable as the index. The result is a new array variable representing the new state of the field for all objects. This is much more concise and efficient than creating an explicit new variable for each concrete object.

We still need to handle assignments with conditions. If the condition does not hold all field values should remain the same, so we can simply use an ite to ensure that in this case the new array variable is equal to the previous version.

field assignment: cond : objectvar.field := expression
formula: field1 = ite(cond, *write*(field0, objectvar, expression), field0)

A reference to an object field is represented as a read of the latest version of the field array, using the object variable as the lookup index.

field reference: objectvar.field
formula: *read*(field0, objectvar)

For a more complete example, see Figure 3(b). This example clearly demonstrates that the SMT formula can be much more concise than the CP model arising from the translation discussed in the previous section. Its weakness is its inability to reason over disjunction (compared to *element* in the CP approach). The approaches are compared further in Section 4.3.

```
pizza1 in {Veg, Marg}
element(index1, [Veg,Marg], pizza1)
element(index1, [Veg_numSlices0,Marg_numSlices0], pizza1_numSlices0)
temp1 = pizza1_numSlices0 + 1
Marg_numSlices1 = [Marg_numSlices0,temp1][pizza1 = Marg]
Veg_numSlices1 = [Veg_numSlices0,temp1][pizza1 = Veg]
element(index2, [Veg,Marg], pizza1)
element(index2, [Veg_numSlices1,Marg_numSlices1], pizza1_numSlices1)
b1 = (pizza1_numSlices1 == slicesPerPizza)
Marg_numSlices2 = [Marg_numSlices1,0][b1 ∧ pizza1 = Marg]
Veg_numSlices2 = [Veg_numSlices1,0][b1 ∧ pizza1 = Veg]
```

(a) CP Translation: Constraints

```
(pizza1 = Marg) ∨ (pizza1 = Veg)
numSlicesArray1 = write(numSlicesArray0, pizza1, read(numSlicesArray0,pizza1)+1)
b1 = ( read(numSlicesArray1,pizza1) = slicesPerPizza )
numSlicesArray2 = ite(b1, write(numSlicesArray1,pizza1,0), numSlicesArray1)
```

(b) SMT Translation: Formula

Fig. 3: Translation of assignments 14-17 of the running example (Figure 2) using (a) the obvious CP approach, and (b) the SMT approach.

4 A New Approach to Modelling Assignments

The main problem with the CP approach presented earlier is the excessive number of variables created to store new field values for every object possibly affected by a field assignment. Essentially we maintain a representation of the complete state of the program after each execution step.

This is not actually necessary. Our only real requirement is to ensure that the values retrieved by variable references are correctly determined by the assignment statements. Maintaining the entire state is a very inefficient way of achieving this, since we may make several assignments to a field using different object variables before ever referring to the value of that field for a particular concrete object. To take advantage of this observation, we move away from the state-based representation, instead simply creating a variable for each field reference, and constraining this to be consistent with the relevant assignments.

4.1 The General Case

We first need to define which assignment statements are *relevant* (i.e. may affect the retrieved value) for a given variable reference. Let a_i be the assignment on line i of the flattened list, and o_i, f_i and c_i be the object, field identifier and set of conditions for this assignment. For a reference to variable obj.field occurring on line n, assignment a_j is *relevant* iff the following conditions hold.

$j < n$ and f_j=field	(occurs before the reference, uses the correct field)
$Dom(\mathsf{obj}, n) \cap Dom(o_j, j) \neq \emptyset$	(assigns to an object which may equal obj)
$\nexists u : o_u$=obj,f_u=field,c_u=$\emptyset$,j<u<n	(not overwritten by an unconditional assignment)

As an example, consider the reference to Veg.fullPizzas on line 41 in Figure 2. Of the eight assignments to the fullPizzas field (all of which occur before this reference), the following three are relevant. The others cannot affect the retrieved value as they use object variables (e.g. pizza3) whose domains do not include Veg.

```
 1.              Veg   .  fullPizzas  :=  0
18.   (b1)  :  pizza1  .  fullPizzas  :=  pizza1.fullPizzas + 1
23.   (b2)  :  pizza2  .  fullPizzas  :=  pizza2.fullPizzas + 1
```

For a correct model we need constraints ensuring that the retrieved value (Veg_fullPizzas) corresponds to the most recent assignment which updated the read variable. To achieve this we introduce a new integer variable indexvar whose value indicates which of the relevant assignments this is. We use three element constraints to ensure that the selected assignment applies to the correct object, has true execution conditions, and assigns a value equal to the result.

```
element(indexvar, [Veg,pizza1,pizza2], Veg)
element(indexvar, [true,b1,b2], true)
element(indexvar, [0, pizza1_fullPizzas + 1, pizza2_fullPizzas + 1], Veg_fullPizzas)
```

Note that pizza1_fullPizzas and pizza2_fullPizzas are the variables introduced for the field references used as part of the assigned values. These would be constrained using their own list of relevant assignments.

The only remaining requirement is that we must choose the *latest* applicable assignment. Using the natural order for the arrays this corresponds to the greatest index. We therefore add constraints stating that if at index i the object variables are equal and the execution condition is true, then the selected index must be no less than i.

$$(\mathsf{b1} \wedge \mathsf{pizza1} = \mathsf{Veg}) \rightarrow \mathsf{indexvar} \geq 2$$
$$(\mathsf{b2} \wedge \mathsf{pizza2} = \mathsf{Veg}) \rightarrow \mathsf{indexvar} \geq 3$$

The general form of the constraints used for field references is shown below. References to local variables are treated as field references where the object variable is the same as that used for all relevant assignments. When this is the case the first element constraint is not required (as it is trivially satisfied), and the implications can be simplified. Other obvious simplifications are also applied.

```
field reference:                         queryobj.field
relevant assignments:               cond1 : obj1.field := expr1
                                              ...
                                    condn : objn.field := exprn
constraints:              element(indexvar, [obj1, ..., objn], queryobj)
                          element(indexvar, [cond1, ..., condn], true)
                          element(indexvar, [expr1, ..., exprn], queryobj_field)
                             (cond2 ∧ queryobj = obj2) → indexvar ≥ 2
                                              ...
                             (condn ∧ queryobj = objn) → indexvar ≥ n
```

As an optimisation, when the code contains more than one reference to some variable v, we insert an unconditional assignment to v at the time of the earlier read, using the read result as the assigned value. This will become the earliest relevant assignment for the later read, which helps to avoid duplication of expressions and constraints.

There is some similarity between our extraction of relevant assignments and the dynamic slicing technique proposed in [?]. Note however that slicing is used only to reduce the number of statements to be translated into constraints. The actual translation is still based on the standard CP approach.

4.2 Special Cases

In some cases, we can detect a pattern to the relevant assignments which allows us to use a more specialised constraint. The three special cases we look for are Boolean variables, sequences of assignments representing a sum calculation, and sequences of assignments representing a maximum or minimum calculation. The translation of assignments automatically detects the cases described below and uses the more efficient translation.

Boolean Variables When the referenced variable is of type bool, we can define a Boolean expression for the retrieved value rather than using element constraints and implications. The expression (shown below) is true if some assignment with a true value applies and no later assignment with a false value applies.

field reference: $q.\text{field}$
assignments: $c_i : o_i.\text{field} := e_i \qquad i \in 1..n$
expression:
$$\bigvee_{i \in 1..n} \left(c_i \wedge e_i \wedge (o_i = q) \wedge \bigwedge_{j \in i+1..n} (e_j \vee \neg c_j \vee (o_j \neq q)) \right)$$

Local variables are handled in the same way except without the object equalities. Using this constraint instead of the generic constraint eliminates the need to introduce an index variable, and allows simplifications to be performed when objects are known to be equal, an assigned value is fixed, or an assignment is unconditional.

Sum Calculations Computations often involve taking the sum of a set of numbers. In a procedural language, sums are commonly calculated by iteratively adding each number to a variable representing the total. This coding pattern results in a sequence of writes where each written value is an addition of the previous value of this variable and some other number. When this pattern is detected, we can replace the usual constraints with a sum constraint.

***relevant assignments*:** total := 0
total := total + value1
total := total + value2
total := total + value3
***constraint*:** finaltotal = sum([0,value1,value2,value3])

In the example above all assignments were unconditional and used exactly the same variable (the local variable total). It is also possible to use a general form of this constraint for sequences of assignments which do not represent a pure sum but a related calculation, such as counting the objects which satisfy some condition. Consider again the relevant writes for the reference to Veg.fullPizzas discussed earlier. A better constraint for Veg_fullPizzas is shown below.

***assignments*:**	Veg . fullPizzas := 0
	(b1) : pizza1 . fullPizzas := pizza1.fullPizzas + 1
	(b2) : pizza2 . fullPizzas := pizza2.fullPizzas + 1
***constraint*:**	Veg_fullPizzas = sum([0, bool2int(b1 $\wedge$ pizza1=Veg), bool2int(b2 $\wedge$ pizza2=Veg)])

A major advantage of this constraint is that it removes the need to create and constrain the variables pizza1_fullPizzas and pizza2_fullPizzas. These can be excluded from the model entirely as they were only used to re-assign to the same variable, and are now not required to define the values retrieved from this field.

The general form of the alternative constraint used for sums is given below. The first assignment gives the initial value of qobj.field. If not already present it is created from the initialisation assignments for objects in the domain of qobj.

field reference:	qobj.field
assignments:	qobj.field := init
	$\text{cond}_i : \text{obj}_i.\text{field} := \text{obj}_i.\text{field} + \text{expr}_i \qquad i \in 1..n$
constraint:	$\text{q\_field} = \text{sum}([\text{init}, \text{expr}_i \times \text{bool2int}(\text{cond}_i \wedge \text{obj}_i = \text{qobj}) \mid i \in 1..n])$

Max/Min Calculations Another common coding pattern is to calculate a maximum or minimum by iterating through a list of values overwriting a variable each time a smaller/larger value is found. As with sum, we can detect this pattern when building the constraints for the final read of the variable. This time in order for the alternative constraint to apply, every non-initialisation assignment must have a condition which compares the current value of the variable with the assigned value. When there are no other assignment conditions, and the variable is a local variable (or the assigned-to object is known to equal the read object for all relevant assignments), we can use a max/min constraint as shown below.

***assignments*:**	max := init
	(value1 > max) : max := value1
	(value2 > max) : max := value2
***constraint*:**	finalmax = max([init, value1, value2])

We can again extend this to apply to field assignments and assignments with additional conditions. When extra conditions are present, we are calculating the maximum or minimum value for which these additional conditions hold. For a maximum, we constrain the result to be no less than any value for which the extra conditions hold, and to equal one of the values for which the conditions hold. Minimum is handled equivalently.

field reference:	queryobj.field
assignments:	$(\text{cond}_i \wedge \text{value}_i > \text{obj}_i.\text{field}) : \text{obj}_i.\text{field} := \text{value}_i \qquad i \in 1..n$
constraints:	$\bigvee_{i \in 1..n} cond_i \wedge (obj_i = queryobj) \wedge (queryobj\_field = value_i)$
	$\bigwedge_{i \in 1..n} (cond_i \wedge obj_i = queryobj) \rightarrow queryobj\_field \geq value_i$

Table 1: Comparing three approaches to modelling destructive assignments.

		Time (secs)						Failures (000s)				
Problem		`smt`	`orig`	`orig+`	`new`	`new+`	`hand`	`orig`	`orig+`	`new`	`new+`	`hand`
`proj1`	200	2.2	23.0	0.1	12.1	0.1	0.1	56	0	34	0	0
	225	2.4	3.2	0.1	1.5	0.1	0.1	9	0	4	0	0
	250	1.6	61.9 3	0.1	61.7 3	0.1	0.1	99	0	127	0	0
`proj2`	22	115.6 2	84.8 1	42.7 1	51.7 2	23.7 1	7.6	39	31	110	35	22
	24	221.1 7	286.9 9	167.6 5	170.9 6	129.2 4	92.2 4	92	89	368	239	280
	26	262.7 8	376.2 16	293.3 10	255.9 11	137.9 6	128.9 5	120	144	583	251	452
`pizza`	3	56.0	37.4 1	25.1	7.0	3.1	2.0	175	118	30	14	0
	4	226.4 8	180.9 7	175.7 7	138.0 4	79.3 2	2.1	544	541	377	252	1
	5	480.9 22	411.8 18	407.5 18	343.4 13	298.3 12	2.2	1170	1216	865	945	7

4.3 Comparison with Earlier Approaches

We compared the three presented translation techniques experimentally, using the pizza ordering example plus two benchmarks used in [7] (the other benchmarks require support for collection operations, as discussed in the next section). We used 30 instances for each of several different sizes to evaluate scaling behaviour. For the `orig`inal and `new` CP approaches we show the effect of adding special cases (`orig+` and `new+`). Special cases can be detected in the original method, but only for local variables. Using the new translation makes these cases also recognisable for fields. As a reference we also include a fairly naive `hand` written model for each problem. The Java code defining the problems and all compared constraint models are available online at `www.cs.mu.oz.au/~pjs/optmodel`.

The CP models were solved using the lazy clause generation solver Chuffed. The SMT versions were solved using Z3 [5]. Z3, like most SMT solvers, does not have built-in support for optimisation. We used a technique similar to [11] to perform optimisation using SMT: in incremental mode, we repeatedly ask for a solution with a better objective value until a not satisfiable result is returned.

Table 1 shows average time to solve and failures for the different models. The small number next to the time indicates the number of timeouts (> 600s). These were included in the average calculations. The results show that while the SMT approach does compete with the original approach, with special case treatment it does not. The `new` approach is quite superior and in fact has a synergy with special cases (since more of them are visible). `new+` competes with `hand` except for `pizza` where it appears that the treatment of the relationship between slices and full pizzas used in `hand` is massively more efficient than the iterative approach in the simulation.

5 Collection Operations

The code for the pizza ordering example makes use of collection classes from the Java Standard Library: Set, List and Map. In this case no special handling is required as all collection operations are independent of the decisions, but often it is more natural to write code where that is not the case. For example, say we wished to extend our application to choose between several possible pizza outlets, each with a different menu. We could do this by adding one extra line at the beginning of the buildOrder function.

```
menu = chooseFrom(availableMenus);
```

This change means the contents of the OrderItem list in the Order class will depend on the decisions, so the for loop iterating over this list (in buildOrder) will perform an unknown (though bounded) number of iterations, and the result of any query operation on this list will also depend on the decisions.

In [7], collection operations were supported by introducing appropriately constrained variables representing the state of each collection after each update operation (e.g. List.add). Query operations (e.g. List.get) were represented as a function of the current state of the relevant collections. This is analogous to the way field assignments were handled, with the same drawbacks.

Fortunately our new technique can also be extended to apply to collection operations, resulting in a much more efficient representation. Where previously the flattened list of state changing operations contained only assignments, we now also include collection update operations. Then every query operation on a collection is treated analogously to a field reference. That is, a new variable is created to hold the returned value, and constraints are added to ensure that this value is consistent with the relevant update operations.

Below we provide details of the constraints used for List operations. Set and Map operations are treated similarly; a detailed description is omitted for brevity. We then give experimental results using collection-related benchmarks from [7].

5.1 Example: List

For the List class we support update operations add (at end of list) and replace (item at index), and query operations get (item at index) and size.

A code snippet containing one of each operation type is shown below. Also shown are the assumed possible variable values and initial list contents, and the flattened list of collection update operations. Each update operation has an associated condition, list, index and item. For the add operation, the index is a variable size1 holding the current size of list1. The first three operations in the table reflect the original contents of the lists.

```
if(cond) {
 list1.add(A);
 list1.replace(0, item);
}
if(list2.size() > ind)
 item = list2.get(ind);
```

(a) Code

```
list1 ∈ {L1,L2,L3}
ind   ∈ {0,1,2}
list2 ∈ {L1,L2,L3}
item ∈ {A,B,C}
cond ∈ {true,false}
L1:[A,B], L2:[C], L3:[]
```

(b) Variables

Cond		List	Index		Item	
		L1	[0]	:=	A	*(add)*
		L1	[1]	:=	B	*(add)*
		L2	[0]	:=	C	*(add)*
cond	:	list1	[size1]	:=	A	*(add)*
cond	:	list1	[0]	:=	item	*(repl)*

(c) Update Operations

With our limited set of supported update operations (which is nevertheless sufficient to cover all code used in the benchmarks from [7]), the size of a list is simply the number of preceding add operations applying to this list and having true execution conditions. Note that the replace operation is not relevant to size.

```
query:       sizeresult := list2.size()
constraint:  sizeresult = sum([ bool2int(list2=L1), bool2int(list2=L1),
                                bool2int(list2=L2), bool2int(list2=list1∧cond) ])
```

A get query is treated almost exactly like a field reference. The value returned must correspond to the most recent update operation with true execution condition which applied to the correct list and index. There is however one extra complication to be considered. Constraining the get result to correspond to an update operation has the effect of forcing the index to be less than the size of the list. This is only valid if the get query is actually executed.

In the constraints shown below, the final element of each array has been added to leave the index unconstrained and assign an arbitrary value A to our result variable when the get would not be executed (sizeresult>ind is false). Without this the constraints would force ind to correspond to an operation on list2 regardless of whether or not the get query is actually executed, incorrectly causing failure when list2 is empty. We fix the result rather than leaving it unconstrained to avoid searching over its possible values.

```
query:        getresult := list2.get(ind)
constraints:  element(indexvar, [L1,L1,L2,list1,list1,list2], list2)
              element(indexvar, [0,1,0,size1,0,ind], ind)
              element(indexvar, [true,true,true,cond,cond,¬(sizeresult>ind)], true)
              element(indexvar, [A,B,C,A,item,A], getresult)
              (list2=L1) ∧ (ind=1)                → indexvar ≥ 2
              (list2=L2) ∧ (ind=0)                → indexvar ≥ 3
              (list2=list1) ∧ (ind=size1) ∧ cond  → indexvar ≥ 4
              (list2=list1) ∧ (ind=0) ∧ cond      → indexvar ≥ 5
              ¬(sizeresult>ind)                   → indexvar ≥ 6
```

5.2 Comparison on Benchmarks with Collections

Table 2 compares the various translation approaches (excluding `smt` and `orig` which were shown to be not competitive in Table 1) and hand written models, using problems involving collections from [7]. It is clear that the new translation substantially improves on the old in most cases, and is never very much worse (`bins`,`knap1`). With the addition of special case treatment the new translation is often comparable to the hand written model, though certainly not always (`proj3`,`route`). In a few instances it is superior (`bins`,`golf`), this may be because it uses a sequential search based on the order decisions are made in the Java code, or indeed that the intermediate variables it generates give more scope for reusable nogood learning.

6 Conclusion

Effective modelling of destructive assignment is essential for any form of reasoning about procedural code. We have developed a new encoding of assignment and state that gives effective propagation of state-related information. We

Table 2: Comparison on examples which use variable collections.

		Time (secs)				Failures (000s)			
Benchmark		`orig+`	`new`	`new+`	`hand`	`orig+`	`new`	`new+`	`hand`
`bins`	12	2.6	5.3	1.1	1.2	8.1	32.5	6.2	13.8
	14	82.8 1	129.6 3	7.6	18.0	95.4	612.9	75.1	169.9
	16	327.2 15	391.6 15	84.8	141.6 5	315.1	1617.6	749.5	1355.0
`golf`	4,3	0.7	0.2	0.2	21.3	0.7	0.8	0.7	159.7
	4,4	3.4	2.0	0.3	0.1	0.8	6.7	0.0	0.0
	5,2	2.4	0.8	0.3	1.5	0.4	0.3	0.0	12.3
`golomb`	8	1.3	1.2	1.2	1.2	10.8	10.4	10.4	24.0
	9	14.0	12.9	12.9	13.7	55.4	51.9	51.9	149.4
	10	161.5	144.1	151.8	178.8	281.1	284.5	284.5	1211.0
`knap1`	70	2.1	8.3	2.8	1.8	33.3	2.2	1.9	33.3
	80	7.5	18.4	7.1	6.8	95.7	3.5	3.5	95.7
	90	14.2	31.9	12.7	13.9	180.2	4.5	4.5	180.2
`knap2`	70	20.9	23.2	22.4	34.7	247.8	245.4	245.4	425.8
	80	88.4 2	87.7 2	93.9 2	117.5 3	935.2	901.8	915.0	1253.1
	90	223.6 5	229.9 5	230.9 5	207.0 5	2263.9	2182.7	2199.3	2085.5
`knap3`	40	26.2	0.9	0.3	0.2	14.3	0.5	0.4	1.3
	50	81.1	2.2	1.3	0.1	25.0	0.8	0.6	2.4
	60	295.2 6	4.2	1.8	0.4	58.7	1.4	1.2	10.2
`proj3`	10	153.9 5	2.3	2.4	0.1	289.3	9.3	11.6	0.1
	12	509.4 24	28.0	20.7	0.1	778.5	83.5	92.1	0.2
	14	600.0 30	133.9 2	102.9 1	0.1	807.3	299.5	394.5	0.5
`route`	5	34.2	1.7	1.7	0.2	34.0	6.3	6.3	2.3
	6	338.3 3	43.7	43.1	0.8	195.8	57.5	57.5	7.6
	7	600.0 30	536.9 20	502.5 17	2.7	263.2	286.9	333.1	19.2
`talent`	3,8	11.1	3.4	0.9	0.8	25.9	17.2	5.0	8.9
	4,9	170.8	42.7	8.8	7.3	159.7	127.3	31.1	52.4
	4,10	545.5 22	223.0 1	77.9	54.6	459.7	510.8	178.1	212.5

demonstrate the effectiveness of this encoding for the automatic generation of optimisation models from simulation code, showing that the resulting model can be comparable in efficiency to a hand-written optimization model.

In the future we will investigate the use of this encoding for applications such as test generation. The main difference is the lack of a known initial state. This will require the creation of variables to represent unknown initial field values, with constraints ensuring that if a pair of object variables are equal then their corresponding initial field variables are also equal. Uncertainty about the initial state will also affect the number of relevant assignments for field references. For a query object with unbounded domain all assignments to the same field occurring prior to the read are relevant, unless one of these is an unconditional assignment using this exact variable. These differences may mean that redundant constraints relating reads to each other (which we have not discussed due to their lack of impact for our application) become more important for effective propagation.

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. Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185. February 2009.
2. 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.
3. Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
4. H. Collavizza, M. Rueher, and P. Van Hentenryck. CPBPV: a constraint-programming framework for bounded program verification. *Constraints*, 15(2):238–264, 2010.
5. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
6. R. W. Floyd. Assigning meanings to programs. In *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, volume 19, pages 19–31, 1967.
7. K. Francis, S. Brand, and P.J. Stuckey. Optimization modelling for software developers. In M. Milano, editor, *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming*, number 7514 in LNCS, pages 274–289. Springer, 2012.
8. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
9. James C. King. Symbolic Execution and Program Testing. *Com. ACM*, pages 385–394, 1976.
10. John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.
11. Roberto Sebastiani and Silvia Tomasi. Optimization in SMT with $LA(Q)$ cost functions. In *IJCAR*, pages 484–498, 2012.