Loop Untangling

Kathryn Francis and Peter J. Stuckey

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

Abstract. An effective translation from procedural code into equivalent constraints is necessary in order to facilitate automated reasoning about the behaviour of programs. We consider the translation of bounded loops, proposing a new form of loop unwinding called *loop untangling*. In comparison to standard loop unwinding the constraints representing each iteration of the loop are greatly simplified. This is achieved by decoupling the execution order from the representation of each individual iteration. We illustrate this new technique using two different examples and provide experimental results verifying that the technique produces simpler models which result in much better solver performance.

1 Introduction

A translation from procedural code into equivalent constraints is a prerequisite for various applications based on automatic reasoning about program behaviour, such as program testing [15], test generation [19] and program verification [10, 14]. This paper is concerned specifically with the treatment of loops (**for** loops and **while** loops) during this translation.

We focus on bounded loops, where a limit on the number of iterations is assumed or can be computed. Bounded loops arise in bounded model checking (e.g.[6]), simulation optimization (e.g. [11, 5]), and other forms of symbolic execution. The typical approach to handling bounded loops is *loop unwinding*, which involves flattening the loop by creating a copy of the body for each potential iteration. This is used in e.g. [11, 12, 5, 6, 3, 4].

The key insight of this paper is that the iterations of a loop do not necessarily need to be identified by the order of execution. That is, when creating copies of the loop body we do not have to label them as the iteration reached by execution first, second, and third, as is done in standard loop unwinding. Instead we can choose a different way of identifying each potential iteration, and then link them together using a separate representation of the execution order.

We describe here a new technique called *loop untangling* which does just that. Instead of execution order, iterations are identified by the value taken by a key expression within the loop body. This can vastly simplify the constraints for each copy of the loop body as the value of this key expression is known. As shown in Section 4, the result is greatly improved solver performance.

2 Motivating Examples

We give here two example programs where standard loop unwinding produces a particularly inefficient model, and sketch how loop untangling can provide a better translation. We will later show how this can be achieved automatically.

Our motivating examples come from a tool which allows combinatorial optimisation problems to be defined procedurally [11, 12]. A programmer with no modelling experience can define an optimisation problem by writing a Java method which uses provided non-deterministic library methods to build a random solution to the problem, and then evaluates that solution, returning a measure of its quality or throwing an exception if it is invalid. The tool automatically finds the values to be returned by the library functions in order to produce the best return value. This is achieved by translating the code into equivalent constraints and passing the resulting model to a constraint solver. More details can be found in [11] but are not important for this work.

Our first example (below) is a routing problem which was one of the original benchmarks from [11]. Given a set of jobs, each of which has a pickup stop and a delivery stop, the problem is to choose the shortest Hamiltonian route visiting all stops, with no delivery stop visited before the corresponding pickup. Note that the ChoiceMaker argument provides the non-deterministic decision making methods. In this case the method chooseOrder is used, which returns a permutation of the given list.

```
int buildRoute(ChoiceMaker chooser) {
  List<Stop> route = chooser.chooseOrder(allStops);
  // compute arrival times
  int currentLocation = startLocation;
  int currentTime = 0;
  for(Stop stop: route) {
    int nextLocation = stop.getLocation();
    currentTime += travelTime(currentLocation, nextLocation);
    stop.arrivalTime = currentTime;
    currentLocation = nextLocation;
  }
  tripFinishTime = currentTime + travelTime(currentLocation, startLocation);
  // check no pickup is after the corresponding delivery
  for(Job j : jobs) {
    if(j.pickupStop.arrivalTime > j.deliveryStop.arrivalTime)
      throw new Exception();
  }
  return tripFinishTime;
}
```

Fig. 1: Java code defining a routing problem.

Consider the first loop, which computes the arrival time for each stop. Let us assume the list allStops contains three stops [A,B,C], which means these three stops also occur exactly once in the route list, but in an unknown order. Using standard loop unwinding we would create a copy of the loop body for the first, second, and third iteration. For each of these, the value of `stop` may be A, B, or C. All of the other variables depend on `stop`, so their values are also unknown. Furthermore, when we later look up the arrival time for the pickup and delivery stop for each job, the value retrieved could be the `currentTime` value computed in any of the three iterations.

Figure 2(a) shows the (idealized) MiniZinc [16] produced by loop unwinding. The decisions are the permutation of the stops, enforced by `alldifferent`. Expressions computed within the loop body are represented using arrays indexed by iteration time `Ite` $= 1..n$ (where n is the number of iterations), or `Ite0` $= 0..n$ for those having a version before the loop. Constraints simulate the calculation within the loop, using the `locations` and `dist` arrays to look up parameter values referenced within the `getLocation` and `travelTime` methods. To constrain the final arrival time for each stop s we need to determine which iteration was the last where we changed the `arrivalTime` field of the stop s, encoded using `which`. We then can lookup the `currentTime` in that iteration to give the `arrivalTime`.

```
array[Ite] of var Stop: route;
constraint alldifferent(route);

array[Ite] of var Location: nextL;
array[Ite0] of var Location: currL;
array[Ite] of var int: travT;
array[Ite0] of var int: currT;
constraint currL[0] = startL;
constraint currT[0] = 0;
constraint forall (i in Ite) (
  nextL[i] = locations[route[i]] ∧
  travT[i] = dist[currL[i-1],nextL[i]] ∧
  currT[i] = currT[i-1] + travT[i] ∧
  currL[i] = nextL[i]
);
array[Stop] of var Ite: which;
constraint forall(s in Stop) (
  which[s] = max(i in Ite)
        (i∗bool2int(route[i] = s)));

constraint forall (j in Job) (
  currT[which[pickup[j]]] <=
  currT[which[delivery[j]]]
);
```
(a)

```
array[Stop] of var Stop0: prevS;
constraint path(prevS,_,0);

array[Stop] of var Location: nextL;
array[Stop0] of var Location: currL;
array[Stop] of var int: travT;
array[Stop0] of var int: currT;
constraint currT[0] = 0;
constraint currL[0] = startL;
constraint forall (s in Stop) (
  nextL[s] = locations[s] ∧
  travT[s] = dist[currL[prevS[s]],nextL[s]] ∧
  currT[s] = currT[prevS[s]] + travT[s] ∧
  currL[s] = nextL[s]
);

constraint forall (j in Job) (
  currT[pickup[j]] <= currT[delivery[j]]
);
```
(b)

Fig. 2: (Idealized) Constraints generated using (a) unwinding, and (b) untangling.

The observation we make in this paper is that instead of creating copies of the loop body for each iteration in order of execution, it would be much better to create a copy for *the iteration where* `stop` *equals A, B, and C.* We know that each of these iterations will be executed, the only uncertainty is the order in which this will happen. With a known value for stop, the value of nextLocation is fixed, and crucially the stop whose arrival time we set in each iteration is also known, so when we later look up the arrival time for each stop we know which version of currentTime is relevant in each case. Obviously we still need to link the iterations to each other, as expressions used within the loop depend on the previous iteration. Actually the value of e.g. currentLocation for a given iteration is exactly the value of nextLocation from the previous iteration. So this linking can be achieved with a `path` [13] or `DomReachability` constraint [18].

The (again idealized) MiniZinc produced using loop untangling is shown in Figure 2(b). Since we are identifying iterations by the value of stop, the arrays for expressions calculated within the loop are indexed by Stop or `Stop0` (which includes an artificial initial stop 0). The decisions are `prevS`, that is for each iteration/stop, what is the previous iteration/stop (or 0 for the first iteration). This is used to look up values that depend on the previous loop iteration (or initialization), while a `path` constraint ensures that these predecessor variables correspond to a Hamiltonian path starting anywhere and ending at the artificial stop 0. The arrivalTime for a stop is now simply equal to the currentTime computed in the iteration corresponding to that stop.

Our second example is a pizza ordering problem which was the running example from [12], part of which is shown in Figure 3. The task is to find the cheapest pizza order which will satisfy a group of discriminating pizza eaters. The code computes the acceptable pizzas for each person, then chooses from these for each slice up to the number the person requires. Once the slices are chosen the cost of the order is calculated taking into account a discount for ordering whole pizzas.

```
 1 int buildOrder() {
 2   order = new Order(menu);
 3   for(Person person : people) {
 4     // Find acceptable pizzas
 5     pizzas.clear();
 6     for(OrderItem item : order.items)
 7       if(person.willEat(item))
 8         pizzas.add(item);
 9     // Choose type for each slice
10     for(int i=0; i<person.slices; i++) {
11       OrderItem pizza =
12         chooser.chooseOne(pizzas);
13       pizza.addSlice();
14   } }
15   return order.totalCost();
16 }

17 class OrderItem
18 {
19   int fullPizzas = 0;
20   int numSlices = 0;
21
22   void addSlice() {
23     numSlices = numSlices + 1;
24     if(numSlices == slicesPerPizza) {
25       numSlices = 0;
26       fullPizzas = fullPizzas + 1;
27   } }
28
29   ...
30 }
```

Fig. 3: Extract from a Java simulation of a pizza ordering optimisation problem.

The loop we consider this time is the one on lines 10–14, within which we make the decisions and tally up the number of slices and pizzas for each pizza type. Here the order of iterations is actually irrelevant to the final result (the cost of the order). It is only the number of times each type of pizza is chosen which matters. Unwinding the loop introduces symmetries and also creates a lot of added uncertainty as the pizza type whose numSlices and numPizzas field is changed in each iteration is unknown. It would be much better to create a variable giving the number of times each type of pizza is chosen, and then constrain the final value of numSlices and numPizzas for each pizza type to be a function of this variable.

Loop untangling achieves this by labelling the iterations by the return value of chooseOne (assigned to the pizza variable on line 11/12). Note that in this case the label is not unique, so we will need a copy of the body for e.g. the first time Vegetarian is chosen, and the second time, up to the maximum times possible. In each of these iterations the value of numSlices and numPizzas will be fixed. Furthermore, when we later look up these values for a particular pizza type, we know that the result will be the value computed in one of the iterations corresponding to that pizza type. Which iteration will depend only on the number of times that pizza type is chosen. The resulting constraint system is far simpler and propagates much more efficiently.

We describe in the following sections our loop untangling technique which can be applied to any loop, and which when applied to the examples above results in much better performance than standard loop unwinding. It is not necessary to detect specifically that in the first example the value of currentLocation is exactly the value of nextLocation from the previous iteration, nor to detect in the second example that the order of iterations is irrelevant. Provided the appropriate choice of labelling scheme for iterations our generalised implementation automatically produces a model which is equivalent to the better model in both cases.

3 General Loop Untangling Technique

This section explains the process of converting code into equivalent constraints using loop untangling rather than loop unwinding. The underlying translation technique is the query based approach described in [12]. The key feature of this technique is that rather than modelling the current state of the program at each execution step, we simply constrain the value of each state query to correspond correctly to the preceding state changes. This is a necessary prerequisite for loop untangling because it allows the execution order of state changes to be viewed as a decision.

The translation is broken into two phases. First the code is flattened into a list of *basic steps*: state changes, state queries, and path control points. Then the result of each state query in this list is constrained to correspond correctly to the changes and control points. The difference between the technique we describe here and that used in [12] is a new approach to making copies of loop bodies while flattening, and a different representation for the constraints defining which state changes occur before which state queries.

3.1 Programs as Ordered State Changes and State Queries

We consider a Java program to consist of a sequence of *basic steps*, each of which is a *state change*, *state query*, or *path control point*. At the lowest level all state changes are assignments and all state queries are variable references. However, since our application of interest (defining combinatorial optimisation problems using imperative code) tends to make heavy use of collections (sets, lists and maps), we treat the core collection operations as atomic state changes (e.g. add item to list) and state queries (e.g. length of list). Path control points are points in the code where execution branches or merges. That is, **break**, **continue** and **return** statements, plus the beginning and end of **then** blocks, **else** blocks and loop bodies, and the end of methods (if there are multiple **return** statements).

3.2 Flattening

The first step in our translation is to convert the code into a list of basic steps. For example the code in lines 5-12 of the routing example (Figure 1) is flattened as shown in Figure 4. To save space we have not separated compound queries into individual parts. For example stop.getLocation() is actually a query for the value of the stop variable, and then a query for the result of the getLocation method called on that stop variable, which is itself a query for the location field of the stop.

Note that this list is not really *flat* yet, as items inside loop bodies may occur more than once in an execution of the program. To solve this we need to create copies of the loop body in such a way that each copy is executed at most once.

c_1:	currentTime := 0	(5)
c_2:	i := 0	(6)
q_1:	i < route.size()	(6)
p_1:	*start loop* (q_1)	(6)
p_2:	*start loop body*	(6)
q_2:	route.get(i)	(6)
c_3:	stop := q_2	(6)
q_3:	stop.getLocation()	(7)
c_4:	nextLocation := q_3	(7)
q_4:	currentTime+travelTime(..)	(8)
c_5:	currentTime := q_4	(8)
q_5:	currentTime	(9)
q_6:	stop	(9)
c_6:	q_6.arrivalTime := q_5	(9)
q_7:	nextLocation	(10)
c_7:	currentLocation := q_7	(10)
q_8:	i+1	(11)
c_8:	i := q_8	(11)
q_9:	i < stopsInOrder.size()	(11)
p_3:	*end loop body* (q_9)	(11)
p_4:	*end loop*	(11)
q_{10}:	currentTime+travelTime(..)	(12)
c_9:	tripFinishTime := q_{10}	(12)

Fig. 4: Flattened loop

3.3 Creating Iterations

When standard loop unwinding is used (as in [12]), we create a copy of the loop body for each potential iteration and label them as the first, second, third etc. The execution order is fixed, but each individual iteration can have a large amount of uncertainty. The idea behind loop untangling is to instead create and label our iterations in a way that reduces the uncertainty within each individual iteration.

The first step is to choose a state query inside the body to be used as the *label query*. The label query is how we will refer to the loop iteration, and ideally knowing the value of the label query will make the loop body much easier to model. Currently this choice is specified via annotation, although it seems clear that some simple static analysis should give us good choices. In our illustrative examples, we choose the iteration argument stop (q_2 in Figure 4) and the choice of pizza type assigned to pizza (line 11 in Figure 3).

Given a label query q_L, we determine the maximum number of times the loop body may be executed (n), and for each iteration $i \in 1..n$ we compute the set of possible values D_i which could be taken by q_L. This is exactly the same calculation as would be done as part of standard loop unwinding.

We then create copies of the loop body as follows. For each value v in the union of the domains D_i computed above, we create k copies of the loop body, where k is the number of iterations in which D_i contains v. For each copy, we add a constraint that if this iteration is reached by execution then the value of q_L is v. This means that we can assume a fixed value for each iteration. If execution reaches the iteration then we know its value will be v, and if execution does not reach this iteration then the value of any query contained in it is irrelevant. When multiple copies are created for value v we also impose a fixed execution order on these to eliminate symmetry, and number them accordingly.

Note that the added constraint setting q_L to take value v does not replace the constraints ordinarily used to define the result of the query based on the preceding state changes. These are still needed but they will now impose a constraint on the (no longer fixed) execution order rather than the query result.

Note also that we may create more than n copies of the loop body. A good choice of label query will remove a lot of uncertainty from individual iterations without introducing too many extra iterations. If for the chosen label query every computed domain D_i contains only a single value, then no uncertainty can be removed and loop untangling is equivalent to loop unwinding.

In the routing example we create a single copy of the loop body for each stop in the allStops list, as we know that each occurs exactly once in stopsInOrder and therefore will occur in exactly one iteration. The new list of basic steps will have three copies of the body (assuming there are 3 stops A,B,C). We will add subscripts a, b, c to the listed step ids in Figure 4 to refer to them.

In the pizza example, we need multiple copies for each pizza type. The number of copies for each is the number of iterations in which that pizza type may be contained in the pizzas list, which is calculated by unwinding as described above. For nested loops such as this one, we create copies of the bodies separately. That is, copies of the inner loop body are not associated with a particular outer iteration. However, there will be multiple copies of the start and end loop nodes for the inner loop, and each of these will belong to a particular iteration of the outer loop. Assuming people = [Ant,Bee], if Ant wants one slice of Veg or Capriciosa, and Bee wants two slices which could be Margherita or Veg, then there are 3 copies of the inner loop for Veg, two copies for Mar, and one for Cap.

3.4 Modelling State Queries

A state query is a function of the state changes occurring before it, while the path control points determine which state changes occur before which state queries. To achieve a correct translation from code to constraints we need to constrain each state query in our flattened list to correspond correctly to the state changes (including artificial state changes added at the beginning to set up the initial program state) and path control points. The constraints described below are the same as those used in [12].

Most types of state query (including variable references) are what we call *lookup queries*, which means they return a value which is a function of only the most recent matching state change. What is meant by *matching* depends on the specific query type. For lookup queries we create a variable *changeID* to represent the ID of the most recent matching state change, and then constrain this ID and the retrieved value appropriately. For example, field references are constrained as shown below. Note that only assignments to the queried field (not other state changes) are relevant.

query	qstep: var ref qobj.field
changes:	step$_1$: obj$_1$.field := expr$_1$
	...
	step$_n$: obj$_n$.field := expr$_n$
variables:	var 1..n: changeID; var int: changestep; var int: qresult;
constraints:	[obj$_1$, ..., obj$_n$][changeID] = qobj $\wedge$
	qresult = [expr$_1$, ..., expr$_n$][changeID] $\wedge$
	changestep= [step$_1$, ...,step$_n$][changeID] $\wedge$
	before(changestep, qstep) $\wedge$
	forall (i in 1..n) (
	(obj$_i$ = qobj $\wedge$ before(step$_i$, qstep)) $\rightarrow$ not before(changestep, step$_i$));

The first constraint requires the chosen assignment to use the same object as the query, which is the definition of *matching* for field references. The next constraint sets the result of the query to the value from the chosen assignment. The before constraint ensures that the change occurs before the query. A change which is skipped by the execution path or which occurs after the query cannot be chosen. We discuss the implementation of before in the next section. The final constraint is used to ensure that we choose the *latest* matching assignment by requiring that no other matching change overwrites our chosen one.

Other queries return a value which is a function of all matching state changes occurring before the query. We call these *aggregate queries*. For these we use before to constrain which changes should be included in the aggregate. For example, list length is constrained as follows (the length of the list is the number of matching add item changes before the query).

query:	qstep: qlist.length()
changes:	step$_1$: list$_1$.add(item$_1$)
	...
	step$_n$: list$_n$.add(item$_n$)
variables:	var int: qresult;
constraints:	qresult = sum (i in 1..n) (bool2int(list$_i$ = qlist $\wedge$ before(step$_i$,qstep)));

Sometimes a lookup query behaves like an aggregate query. This happens when the most recent relevant change itself depends on the previous changes, either because of its arguments or because earlier changes affect whether or not execution reaches the later changes. In these cases it is still possible to use the standard representation for lookup queries, but much better performance can be achieved using a specialised translation. In [12] this was called *special cases*. When untangling rather than unwinding loops we can use the same specialised translations described in [12], as in each special case discussed there the query result is not affected by the order in which the changes occur.

3.5 Modelling the Execution Path

When standard loop unwinding is used, path control points can only cause execution to skip state changes. The relative order is known. So in [12], before was implemented by calculating a Boolean expression $cond_a$ for the conditions under which execution reaches step a, and then defining before as follows.

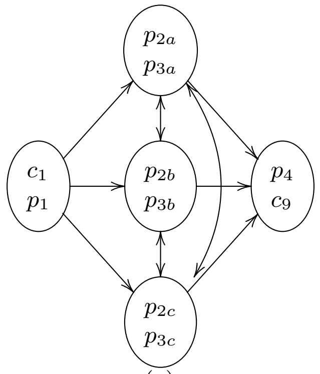
$$\mathsf{before}(step_a, step_b) = (a < b) \wedge cond_a$$

When iterations are identified by something other than execution order, the relative execution order of iterations, and therefore of the basic steps contained in them, is unknown. This means we need a new implementation of before. Our new implementation is based on a graph of the possible execution paths. This graph is very similar to a control flow graph, but it contains a node for every copy of each basic step, rather than a single node for each basic block. The edges are constructed as follows.

– A state query or state change has a single outgoing edge leading to the following step.
– The control point at the beginning of a **then** or **else** block has an edge leading to the first step in the block, and another edge leading directly to the end of the block.
– A **continue**, **break**, or **return** control point has a single outgoing edge to the end of the associated loop body, loop, or method respectively.
– A start loop control point has an edge to its associated end loop control point, and to every start body control point for its loop.
– An end loop body control point has an edge to the start body point for each other iteration of that loop, and to every version of the end loop control point.
– An exception step has no outgoing edges.

Any valid solution must correspond to a path through this graph (between the fixed start and end steps). Note that as required by our application this prevents exception points from being reached.

As we will be looking backwards from queries to the changes affecting them, we assign for each basic step s a predecessor $prev[s]$ which is the basic step



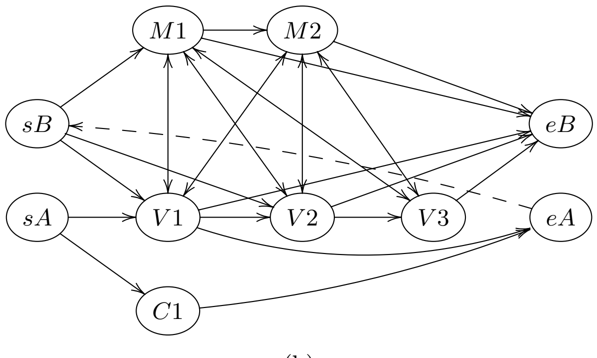


Fig. 5: Path control graphs: (a) lines 5-12 of Figure 1, (b) lines 10-14 of Figure 3.

executed immediately before s. Clearly for most steps this is simply the unique predecessor. The *prev* array can be constrained by a `subpath` constraint [13].

Not all paths through the graph represent valid execution paths. The use of certain edges (where execution branches) is conditional on the result of a Boolean state query referred to in that step. The edge leading into a then or else block can only be used if the **if** condition is *true* or *false* respectively. An edge leading into a loop body is only valid if a query for the loop entry condition returns *true*. In Figure 4 the query used to control the use of edges is shown in brackets next to the source node. For loops (both start loop and end loop body control points) if the query shown is *false* then the edge to the end loop control point must be used.

Figure 5(a) shows a portion of the execution graph for our routing example with basic blocks collapsed. The start can reach each loop iteration for stop = A, B or C, and these can each reach each other and the end of the loop. As an example of the conditions on edges, consider the edges leaving step p_{3a}. Setting $prev[p_4] = p_{3a}$ requires $q_{9a} = false$, as this edge represents exiting the loop, while $prev[p_{2b}] = p_{3a}$ (or $prev[p_{2c}] = p_{3a}$) requires that $q_{9a} = true$, as these edges represent re-entering the loop.

For the pizza example (Figure 5(b)), edges which can be discounted upfront due to false edge conditions or constraints on the label query are not shown. Since Ant runs first it can reach only the first instance of Veg or Cap, and each of these can reach its end since Ant only picks one slice. For Bee the start can reach the first Mar or the first or second Veg. Each of these nodes can reach only the next of the same category or any of the other category, and the end of Bee's loop. Outside this part of the graph is a mandatory path from the end of Ant's loop to the start of Bee's.

We need further constraints on the edges for nested loops, to ensure that we do not enter the inner loop from one outer iteration and leave to a different outer iteration. For example we cannot enter node $V1$ from sA and leave to eB. This is prevented by adding a constraint on the start and end loop body control points (s_i and e_i for i in the iteration set I) for each loop.

$$\forall i, j \in I, \neg(\mathsf{before}(s_i, e_j) \wedge \mathsf{before}(e_j, e_i))$$

This ensures that no other end loop body step from the outer loop can come between a pair of associated start and end body steps for that loop.

As mentioned previously, if we have created multiple iterations for a given value of the label query, we also impose a fixed order on those (using before) to eliminate symmetry. This means that (as shown in Figure 5(b)) edges leading from the start loop control point to the second or later copy of the body for each value of the label query are excluded immediately, and between iterations for the same value we only keep edges leading between successive copies.

3.6 Redefining before

The purpose of constructing the graph described in the previous section is to provide a new definition of the before relation used in our constraints. A simple implementation of before can be achieved by creating a time variable for each node in the graph of possible execution paths, and adding a constraint for each edge to say that if that edge is used then the time of the destination is one greater than the time of the source. Then before can be defined as follows.

$$\mathsf{before}(a, b) = \mathsf{time}[a] < \mathsf{time}[b]$$

In order to prevent changes which are not included on the execution path from affecting queries which are, we require that any step not on the path has a time greater than the number of steps.

While this implementation is correct, it does not provide very strong propagation. Consider the reference to currentTime which forms part of q_4 in the routing example (Figure 4). The options for the *latest matching change* are c_1 and c_5 (which has a version for each iteration of the loop). Imagine we are determining q_{4b} and we have already decided that the a iteration is followed by the b iteration $prev[p_{2b}] = p_{3a}$. Ideally we should know that q_{4b} refers directly to c_{5a}. But after this decision we have that $\mathsf{time}[q_{4b}] \in \{26, 42\}$, $\mathsf{time}[c_{5a}] = \{11, 27\}$, $\mathsf{time}[c_{5c}] = \{11, 43\}$, $\mathsf{time}[c_1] = 1$. So according to the above definition of before each of $\{c_1, c_{5a}, c_{5c}\}$ could be the latest matching state change.

Even harder to handle is when we decide that iteration b does not follow a. We know that all iterations have a matching change for q_{4b}. So if a is not immediately before b, then there must be another iteration in between with a matching change, even though no specific change is known to be between them.

More generally, the logic we would like to have is that whenever all paths between change c and query q go through another change which is known to match q, then change c cannot be chosen for q. Note that although this is related to dominance it is not pure dominance as we do not require that the same change overwrites c on all paths.

Ideally this would be achieved using a global constraint. Such a constraint does not exist, but we have found that including the logic it would use in our simplification phase is sufficient to provide good performance compared with loop unwinding. We intend to implement the global constraint and expect that even better performance should be possible.

3.7 Optimisations and Simplifications

In certain cases it is possible to create a simple expression $closest_c$ defining the conditions which must hold for change c to be the closest matching change to a given lookup query q. When this is possible, we can change the constraints used for q to the simpler form shown below.

query: qstep: var ref qobj.field
changes: step$_1$: obj$_1$.field := expr$_1$
...
step$_n$: obj$_n$.field := expr$_n$
variables: var 1..n: changeID; var int: qresult;
constraints: [obj$_1$, ..., obj$_n$][changeID] = qobj $\wedge$
[$closest_1$, ..., $closest_n$][changeID] $\wedge$
qresult = [expr$_1$, ..., expr$_n$][changeID];

If all changes potentially matching a lookup query are initialisation changes (added at the beginning to set up the initial program state), then only one can match the query, so we can use *true* as the *closest* expression for all of them.

If for a query q there exists a node in the execution graph n such that every edge leading in to n would create a fixed path between a matching change c for q and q, then we can use the edges into n to define the *closest* conditions. For example, consider the reference to currentTime discussed previously (part of q_4 in Figure 4). The query q_{4b} has a matching change in each iteration of the loop (c_{5a}, c_{5b}, c_{5c}), and one before the loop (c_1). All edges into the node p_{2b} (see Figure 5) create a fixed path between one of these changes and q_{4b}. So we can constrain q_{4b} as shown below. Note that since none of the edges leading in to p_{2b} correspond to the change c_{5b} we can discount this change immediately.

query: q_{4b}: currentTime
changes: c_1: currentTime := 0
c_{5a}: currentTime := q_{4a}
c_{5c}: currentTime := q_{4c}
variables: var 1..3: changeID; var int: qresult;
constraints: [prev[p_{2b}]=p_1, prev[p_{2b}]=p_{3a}, prev[p_{2b}]=p_{3c}][changeID] $\wedge$
qresult = [0, q_{4a}, q_{4c}][changeID];

The same can be done for query q_{10} outside the loop using the edges into p_4. This provides both the positive and negative reasoning discussed in the previous section. If we decide to use an edge then the *closest* condition for that change will become *true* and all others will become *false*. If we decide not to use an edge then that *closest* condition will become *false*, excluding the corresponding change. Although these constraints are more verbose than the idealised MiniZinc shown in Section 2, they provide the same propagation strength.

We can also take into account the known relationship between iterations with the same value for the label query. If all changes relevant to a query belong to iterations with the same label value, then their relative order is known (as we have fixed this to avoid symmetry), so we can use the original definition of before.

If in addition all possibly matching changes are known to actually match and the path through each iteration with a matching change is fixed, then we can do even better. In this case it is not possible for execution to skip the change in iteration i without also skipping those in later iterations. So the closest matching change is the one from the last iteration to be executed before the query. Ordering the changes by iteration version, for each change c_i except the last:

$$\mathit{closest}_{c_i} = \mathsf{before}(c_i, q) \wedge \neg\mathsf{before}(c_{i+1}, q)$$

The change from the last iteration is the closest whenever it is before the query. If the query is outside the loop, then we know that all iterations not skipped by execution will occur before the query, so we can simplify the condition further:

$$\mathit{closest}_{c_i} = \mathit{in}[c_i] \wedge \neg\mathit{in}[c_{i+1}] \qquad\qquad \mathit{in}[\mathit{step}_i] = (\mathit{prev}[\mathit{step}_i] \neq \mathit{step}_i)$$

where $\mathit{in}[\mathit{step}_i]$ means step i is included on the execution path. This can be defined as shown above since subpath sets unused nodes to point at themselves.

Finally, for queries which are also contained in an iteration with the same label value, we can assume that all changes in earlier iterations for this value are included on the execution path. If not, then the query will not be reached either so its value does not matter. Therefore for these queries the closest matching change is set to the change from the latest iteration before the query iteration.

The above simplifications are used in the translation of the pizza example. Consider the reference to numSlices on line 23 of the pizza code (Figure 3), in the first Veg iteration. Let us call this query q_{nv1}. The relevant changes are the initialisation assignment for Veg which sets numSlices to 0, and the assignments on lines 23 and 25. All versions of these assignments from non-Veg iterations are known not to match q_{nv1} as they refer to a different pizza object, and all versions from later Veg iterations are known to be after this query. The versions from the current iteration are also after this query, so actually only the initialisation assignment can be chosen. Therefore the value of q_{nv1} can be fixed to 0. This in turn will fix the value assigned to numSlices on line 23 to 1, and the condition on the following line (24) to *false* (assuming slicesPerPizza is fixed to say 2), which means that the assignment on line 25 is not reached by execution.

Now consider the reference to numSlices in the next Veg iteration, query q_{nv2}. As explained above, since the path through the earlier Veg iteration is fixed and the query is also inside a Veg iteration, we can simply use the change from the closest iteration to this one ($V1$). The value of q_{nv2} is therefore 1. The value assigned on line 23 will be 2, and the test on line 24 will succeed. Again the path through this iteration is fixed, but it goes through the assignment on line 25, setting numSlices back to zero. When we consider q_{nv3} applying the same simplification again gives a value of 0.

For the query to Veg.numSlices after the loop (inside the totalCost method), we can use the definition of *closest* described above for queries outside the loop, because all matching changes belong to Veg iterations (except for the init change) and are known to match, and the path through every Veg iteration is fixed. The constraint for this query is therefore:

query:	q: var ref Veg.numSlices
changes:	c_0: Veg.numSlices := 0
	c_{1v1}: Veg.numSlices := 1
	c_{2v2}: Veg.numSlices := 0
	c_{1v3}: Veg.numSlices := 1
variables:	var 1..4: changeID, var int: qresult
constraints:	$[\neg in[c_{1v1}], in[c_{1v1}] \wedge \neg in[c_{2v2}], in[c_{2v2}] \wedge \neg in[c_{1v3}], in[c_{1v3}]]$[changeID] $\wedge$ qresult = [0,1,0,1][changeID];

This is the constraint described in Section 2 which links the final value of Veg.numSlices to the number of times Veg is chosen (which is changeID-1).

4 Experimental Results

We have implemented the loop untangling technique described above, and show here experimental results for the two examples discussed. The constraint models for unwinding and untangling are produced fully automatically from the input Java code, the only additional information given to untangling was the choice of label query for each loop. Table 1 compares untangling to unwinding (the version called new+ in [12]) and to a hand written CP model for the same problem (also from [12]). Each figure is the average for 30 instances of the stated size. The times shown include instances which reached the timeout of 10 minutes, while the failures figures (shown in thousands) exclude them. All models were solved using G12 CPX on a 3.40GHz Intel i5-4670K with 16GB RAM.

The results clearly show the benefit of untangling. For these problems, we are clearly better off using a simple model for each iteration and deciding the order through them, rather than deciding what happens in the i^{th} loop iteration where we know the order. We expect that with a specialized global propagator for managing the before constraint this could be further substantially improved.

		Solving Time						Failures (000s)		
Problem		unwind		untangle		hand		unwind	untangle	hand
pizza	4	94.0s	(2)	1.3s		0.1s		127.8	17.2	0.8
	5	320.7s	(13)	5.8s		0.5s		250.5	49.0	5.7
	6	470.1s	(22)	149.7s	(6)	20.9s	(1)	240.1	480.2	12.1
routing	5	12.9s		1.5s		0.4s		24.5	4.4	2.1
	6	102.8s		8.1s		2.2s		112.4	18.4	9.9
	7	569.3s	(20)	40.8s		14.6s		343.4	67.4	45.7

Table 1: Comparative performance of unwinding and untangling.

5 Related and Further Work

Loop untangling is related to other forms of program analysis that reason about loops. For example, automatic parallelisation of code needs to reason about when iterations can be reordered [17]. We could improve loop untangling by co-opting methods from this area to detect cases where the execution order of iterations can be fixed arbitrarily. The technique described in [2] for detecting commutativity could be a good starting point as a similar query-based viewpoint is taken when considering whether or not reordering iterations changes the outcome.

Loop untangling could also be improved by employing more general forms of program analysis. Typically optimisations performed by compilers are designed to simplify the remaining code, which would in turn simplify our translation to the constraint model. For example, loop untangling implicitly requires reaching definitions, and can also be simplified by constant propagation. While our tool does a basic form of reaching definition analysis and constant propagation these could be improved by full program analysis techniques (e.g. [1]).

An interesting direction for future work is developing a program analysis which would automatically select the label query. By examining the reaching definitions graph and understanding what data in the program is dependent on decisions and what is not we can choose a label query that, when fixed, fixes much of the computation of the loop body. But we need to trade this off against the number of iterations it will create.

Finally our method, and indeed most methods based on symbolic execution, currently only handles bounded loops. Unbounded loops can be approximated by putting an artificial limit on the number of iterations, but otherwise they require techniques to generate loop invariants including interpolation [8] or abstract interpretation [7]. Loop untangling could possibly be extended to handle unbounded loops using an approach similar to that in [9]. There constraints were added for each iteration of the loop lazily as needed when it became known that the previous iteration was entered. We could do something similar, lazily adding nodes to our execution path graph.

6 Conclusion

Standard loop unwinding unnecessarily ties the actual execution order of iterations with the way an individual iteration is identified. The idea behind loop untangling is to decouple these two things by modelling the execution path explicitly. The labelling scheme can be used to reduce the uncertainty in each copy of the loop body. Although it may be necessary to create more copies of the loop body than through standard loop unwinding, with a good choice of labelling scheme this is far outweighed by the relative simplicity of the constraints required for each copy. The final result is a model which is much easier to solve.

Acknowledgments NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

1. A.V. Aho, R. Sethi, J.D. Ullman, and M.S. Lam. *Compilers: Principles, Techniques, and Tools: second edition.* Addison-Wesley, 2006.
2. F. Aleen and N. Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. *ACM Sigplan Notices*, 44(3):241–252, 2009.
3. A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009.
4. P. Brandwijk. Verifying software with SMT and random testing using a single property specification. Master's thesis, University of Amsterdam, 2012.
5. A. Brodsky and H. Nash. CoJava: optimization modeling by nondeterministic simulation. In *Principles and Practice of Constraint Programming (CP)*, pages 91–107, 2006.
6. H. Collavizza, M. Rueher, and P. Van Hentenryck. CPBPV: a constraint-programming framework for bounded program verification. *Constraints*, 15(2):238–264, 2010.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
8. W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.
9. T. Denmat, A. Gotlieb, and M. Ducassé. An abstract interpretation based combinator for modelling while loops in constraint programming. In *Principles and Practice of Constraint Programming (CP)*, pages 241–255, 2007.
10. R. W. Floyd. Assigning meanings to programs. In *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, volume 19, pages 19–31, 1967.
11. 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.
12. K. Francis, J. Navas, and P.J. Stuckey. Modelling destructive assignments. In C. Schulte, editor, *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming*, volume 8124 of *LNCS*, pages 315–330. Springer, 2013.
13. K. Francis and P.J. Stuckey. Explaining circuit propagation. *Constraints*, 19(1):1–29, 2014.
14. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
15. J. King. Symbolic Execution and Program Testing. *Com. ACM*, pages 385–394, 1976.
16. 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.
17. W. Pottenger. The role of associativity and commutativity in the detection and transformation of loop-level parallelism. In *Proceedings of the 12th international conference on Supercomputing*, pages 188–195. ACM, 1998.

18. L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *Practical Aspects of Declarative Languages, 8th International Symposium, PADL 2006, Charleston, SC, USA, January 9-10, 2006, Proceedings*, volume 3819 of *Lecture Notes in Computer Science*, pages 73–87, 2006.
19. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference*, pages 263–272. ACM, 2005.