# Modelling Destructive Assignments

Kathryn Francis  Jorge Navas  Peter Stuckey

National ICT Australia and the University of Melbourne

# Overview

## Symbolic Execution

- Translating procedural object-oriented code into constraints
- Key difficulty is efficiently modelling state changes
- We introduce a new technique for *modelling destructive update*

## Applications

- Test generation
- Program verification
- Optimization by Simulation

# The Context: Optimization by Simulation

## Widespread use of CP

- If CP is going to be used widely, it will need to be embedded in application specific software.
- Someone has to make that software, and there aren't enough CP enthusiasts to go around!
- So we need to create tools which are accessible to ordinary software developers.

## A CP tool for software developers

- Let developers use the skills and knowledge they already have.
- Support a problem definition suited to the application rather than the solver.

# The Context: Optimization by Simulation

## Ask the software developer to answer two questions:

1. If you had access to a perfect decision maker, how would you solve the problem?
2. Given a solution, how would you evaluate it?

## Then, using their answers (which they provide as code) we can:

1. Create an appropriate constraint model automatically and on demand.
2. Provide the solution exactly as their code would have created it.

## Some advantages:

- Don't have to switch mindset and create a mathematical model
- Can re-use existing types and code
- Solution can be provided in an immediately usable form

# Example - Pizza ordering

Each person gives number of slices required, and acceptable pizzas. The ordering program determines the cheapest order using full pizzas or individual slices.

```
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();
}
```

choiceMaker object records choices made in the simulation.

# Flattening

Only assignment statements change state. We can flatten a program (where loop iterations are bounded) to be a list of conditional assignment statements.

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

generates the sequence of conditional assignments

| | 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 |

## Local Variable Optimization

- Local variables value doesnt matter outside their scope
- Simplify sequence of assignments

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

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

# Modelling Assignments

## Existing Techniques

- CP Approach:
  - Replace Java variables by a sequence of CP variables
  - Hold the state of the Java variable in a CP variable
  - Update state
- SMT approach
  - Same as CP approach
  - But use theory of arrays for updating

## New approach

- Only model state that is read!

# Modelling Assignments: CP Approach

## Updating Local Variables

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

Example

(slicesCost>pizzaPrice) : slicesCost := pizzaPrice

element(bool2int(slicesCost0>pizzaPrice0)+1,[slicesCost0,pizzaPrice0], slicesCost1)

## Updating Fields

More complex since we dont necessarily know which object is updated. Assuming objectvar is one of $\{\text{Obj1}, \text{Obj2}, \text{Obj3}\}$:

***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

## Field References

Need to look up the field for the right object. Again assuming objectvar is one of $\{\text{Obj1}, \text{Obj2}, \text{Obj3}\}$:

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

- Two step process
  - Replace field assignments by equivalent local variable assignments
  - Translate local variable assignments to constraints
- Can result in VERY LARGE MODELS

# Modelling Assignments: SMT Approach

## SAT Modulo Theories (SMT)

- Driven significant advances in reasoning about programs

## Updating Local Variables

Similar to CP, but using if-then-else (ite)

***assignment:*** condition : localvar := expression

***constraint:*** localvar1 = ite(condition, expression, localvar0)

Example

(slicesCost>pizzaPrice) : slicesCost := pizzaPrice

slicesCost1 = ite(slicesCost0>pizzaPrice0,pizzaPrice0,slicesCost0)

# Modelling Assignments: SMT Approach

## Theory of Arrays

$\forall a, i, j, x$ (where $a$ is an array, $i$ and $j$ are indices and $x$ is a value )

$$i = j \rightarrow \mathit{read}(\mathit{write}(a, i, x), j) = x$$
$$i \neq j \rightarrow \mathit{read}(\mathit{write}(a, i, x), j) = \mathit{read}(a, j)$$

## Updating Fields

Using theory of arrays, where objects are indexes into an array object.

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

## Field References

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

# Comparing Existing Approaches

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

pizza1 . numSlices := pizza1.numSlices + 1

## CP Approach

element(index1, [Veg,Marg], pizza1)
element(index1, [Veg_numSlices0,Marg_numSlices0], pizza1_numSlices0)
temp1 = pizza1_numSlices0 + 1
element(bool2int(pizza1=Marg)+1,[Marg_numSlices0,temp1],Marg_numSlices1)
element(bool2int(pizza1=Veg)+1,[Veg_numSlices0,temp1],Veg_numSlices1)

## SMT Approach

numSlicesArray1 = write(numSlicesArray0, pizza1, read(numSlicesArray0,pizza1)+1)

# New Approach

## Key Idea

- Create an expression for every variable referenced in a state
- Not for every object+field in every state
- Expression based on relevant assignments

## Relevant Assignments to obj.field

- Occurs before reference, uses correct field .field
- Assigns to an object that may equal obj
- Not overwritten by unconditional assignment

## Encoding

- Encode which assignment applies to the reference!

# Encoding by Example

## Relevant Assignments

Reference Veg.fullPizzas, only 3 assignments to .fullpizzas out of 8 are relevant.

| | Cond | | Object | | Field/Var | | Assigned Value |
|---|---|---|---|---|---|---|---|
| 1. | | | Veg | . | fullPizzas | := | 0 |
| 18. | (b1) | : | pizza1 | . | fullPizzas | := | pizza1.fullPizzas + 1 |
| 23. | (b2) | : | pizza2 | . | fullPizzas | := | pizza2.fullPizzas + 1 |

## Constraint Encoding

Use indexvar to select which assignment applies

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

Constraint to enforce that the last applicable assignment holds.

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

## Boolean Expressions

Encode as a single Boolean expression:

***field reference:*** $q.\text{field}$

***assignments:*** $c_i : o_i.\text{field} := e_i \qquad i \in 1..n$

***expression:***

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

Advantages: no indexvar, easy to simplify.

## Sum Calculations

Usually programmed as an iteration of additions, e.g:

***relevant assignments***: total := 0
total := total + value1
total := total + value2
total := total + value3

***constraint***: finaltotal = sum([0,value1,value2,value3])

Works for conditional assignments as well.

| | | | | | | |
|---|---|---|---|---|---|---|
| | | Veg | . | fullPizzas | := | 0 |
| (b1) | : | pizza1 | . | fullPizzas | := | pizza1.fullPizzas + 1 |
| (b2) | : | pizza2 | . | fullPizzas | := | pizza2.fullPizzas + 1 |

Veg_fullPizzas =
sum([0,bool2int(b1 $\wedge$ pizza1=Veg), bool2int(b2 $\wedge$ pizza2=Veg)])

## Max/Min Calculations

Usually programmed as iterated maintenance of max so far.

***assignments***: $\quad$ max := init

$(\text{value1} > \text{max}) :$ $\quad$ max := value1

$(\text{value2} > \text{max}) :$ $\quad$ max := value2

***constraint***: $\quad \text{finalmax} = \max([\text{init}, \text{value1}, \text{value2}])$

Can be extended to conditional max calculations:

***assignments***: $\quad$ max := init

$(\text{b1} \wedge \text{value1} > \text{max}) :$ $\quad$ max := value1

$(\text{b2} \wedge \text{value2} > \text{max}) :$ $\quad$ max := value2

***constraint***: $\quad \text{finalmax} = \text{init} \quad \vee (\text{b1} \wedge \text{finalmax} = \text{value1})$

$\vee (\text{b2} \wedge \text{finalmax} = \text{value2})$

$\text{finalmax} \geq \text{init}$

$\text{b1} \rightarrow \text{finalmax} \geq \text{value1}$

$\text{b2} \rightarrow \text{finalmax} \geq \text{value2}$

# Experiments: Part I

Comparing: `smt`, `orig`inal CP approach, `orig+` with special cases, `new` CP approach, `new+` with special cases, and `hand`: CP solver chuffed, SMT solver Z3

| Problem | | Time (secs) | | | | | |
|---|---|---|---|---|---|---|---|
| | | `smt` | `orig` | `orig+` | `new` | `new+` | `hand` |
| `proj1` | 200 | 2.2 | 23.0 | 0.1 | 12.1 | 0.1 | 0.1 |
| | 225 | 2.4 | 3.2 | 0.1 | 1.5 | 0.1 | 0.1 |
| | 250 | 1.6 | 61.9 3 | 0.1 | 61.7 3 | 0.1 | 0.1 |
| `proj2` | 22 | 115.6 2 | 84.8 1 | 42.7 1 | 51.7 2 | 23.7 1 | 7.6 |
| | 24 | 221.1 7 | 286.9 9 | 167.6 5 | 170.9 6 | 129.2 4 | 92.2 4 |
| | 26 | 262.7 8 | 376.2 16 | 293.3 10 | 255.9 11 | 137.9 6 | 128.9 5 |
| `pizza` | 3 | 56.0 | 37.4 1 | 25.1 | 7.0 | 3.1 | 2.0 |
| | 4 | 226.4 8 | 180.9 7 | 175.7 7 | 138.0 4 | 79.3 2 | 2.1 |
| | 5 | 480.9 22 | 411.8 18 | 407.5 18 | 343.4 13 | 298.3 12 | 2.2 |

Summary: `hand` < `new+` < `orig+` < `smt` < `orig`

# Collections

## Collections Variables

- Most java code uses collection variables
- We can extend the original CP approach to collections by:
  - Iterating an (unknown) bounded number of iterations over collections
  - Maintaining the state of the collection after each assignment
- Inherits problems of fields (and then some)!

## Extending to Collections

- Update operations treated like assignments
- Query operations treated like field references.
- We support: Lists, Sets and Maps

## Operations

- Updates: add (at end), replace (at index)
- Queries: get (from index), size

```
L1 = [A,B]; L2 = [C]; L3 = [];
if(cond) {
  list1.add(A);
  list1.replace(0, item);
}
if(list2.size() > ind)
  item = list2.get(ind);
```

Relevant Updates for list2.get(ind)

| 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)* |

# Lists

Result getresult of query list2.get(ind) is computed like field access:

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,$\neg$(sizeresult$>$ind)], true)
element(indexvar, [A,B,C,A,item,A], getresult)

| | |
|---|---|
| (list2=L1) $\wedge$ (ind=1) | $\rightarrow$ indexvar $\geq$ 2 |
| (list2=L2) $\wedge$ (ind=0) | $\rightarrow$ indexvar $\geq$ 3 |
| (list2=list1) $\wedge$ (ind=size1) $\wedge$ cond | $\rightarrow$ indexvar $\geq$ 4 |
| (list2=list1) $\wedge$ (ind=0) $\wedge$ cond | $\rightarrow$ indexvar $\geq$ 5 |
| $\neg$(sizeresult$>$ind) | $\rightarrow$ indexvar $\geq$ 6 |

Note the last entry is a dummy value (if the query is not reached)

# Experiments: Collections (secs)

| Benchmark | | orig+ | | new | | new+ | | hand | |
|---|---|---|---|---|---|---|---|---|---|
| bins | 12 | 2.6 | | 5.3 | | 1.1 | | 1.2 | |
| | 14 | 82.8 | 1 | 129.6 | 3 | 7.6 | | 18.0 | |
| | 16 | 327.2 | 15 | 391.6 | 15 | 84.8 | | 141.6 | 5 |
| golf | 4,3 | 0.7 | | 0.2 | | 0.2 | | 21.3 | |
| | 4,4 | 3.4 | | 2.0 | | 0.3 | | 0.1 | |
| | 5,2 | 2.4 | | 0.8 | | 0.3 | | 1.5 | |
| golomb | 8 | 1.3 | | 1.2 | | 1.2 | | 1.2 | |
| | 9 | 14.0 | | 12.9 | | 12.9 | | 13.7 | |
| | 10 | 161.5 | | 144.1 | | 151.8 | | 178.8 | |
| knap2 | 70 | 20.9 | | 23.2 | | 22.4 | | 34.7 | |
| | 80 | 88.4 | 2 | 87.7 | 2 | 93.9 | 2 | 117.5 | 3 |
| | 90 | 223.6 | 5 | 229.9 | 5 | 230.9 | 5 | 207.0 | 5 |
| knap3 | 40 | 26.2 | | 0.9 | | 0.3 | | 0.2 | |
| | 50 | 81.1 | | 2.2 | | 1.3 | | 0.1 | |
| | 60 | 295.2 | 6 | 4.2 | | 1.8 | | 0.4 | |
| proj3 | 10 | 153.9 | 5 | 2.3 | | 2.4 | | 0.1 | |
| | 12 | 509.4 | 24 | 28.0 | | 20.7 | | 0.1 | |
| | 14 | 600.0 | 30 | 133.9 | 2 | 102.9 | 1 | 0.1 | |
| talent | 3,8 | 11.1 | | 3.4 | | 0.9 | | 0.8 | |
| | 4,9 | 170.8 | | 42.7 | | 8.8 | | 7.3 | |
| | 4,10 | 545.5 | 22 | 223.0 | 1 | 77.9 | | 54.6 | |

# Conclusions and Future Work

## Summary

- New encoding of destructive assignment
- Combines:
  - Reasoning over disjunction of CP, with
  - Compact encoding of SMT
- Avoids maintaining all state
- Allows more special cases to be detected

## Future Work

- Apply the encoding to test generation
  - Requires treatment of unknown initial values
  - Requires redundant constraints to relate queries to other queries
- Further improve optimization by simulation

Questions?