# Generating Distinguishing Tests using the MINION Constraint Solver

**Franz Wotawa** Mihai Nica Bernhard K. Aichernig

Institute for Software Technology
Technische Universität Graz
Inffeldgasse 16b/2, A-8010 Graz, Austria
{wotawa,nica,baichern}@ist.tugraz.at

April 9, 2010

The work described in the paper was partially funded by the Austrian Science Fund (FWF) under contract number P20199-N15, and the EU FP7 project MOGENTES ICT-216679, Model-based Generation of Tests for Dependable Embedded Systems.

# Content

1. Motivation
2. Basic definitions
3. Constraint representation
4. Computing distinguishing tests
5. Experimental results
6. Conclusion

# Motivation

- In *Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs*, ICST 2010, Session 2 Mutation Testing the authors introduce the notation of **possible fixes**.
- There might be many of them!
- How to minimize the number of possible fixes?

# Motivation

```
1.  begin
2.      i = 2 * x;
3.      j = 2 * y;
4.      o1 = i + j;
5.      o2 = i * i;
6.  end;
```

```
x = 1, y = 2, o1 = 8, o2 = 4
```

↘ ↙

**Debugger**

↓

Diagnosis candidates: `3.j=2*y` and `4.o1=i+j`

*How to distinguish the diagnosis candidates?*

# Motivation - Distinguishing tests

- Use new (distinguishing) test cases for removing diagnosis candidates!
- Note:
  - A diagnosis candidate can be eliminated if the new test case is in contradiction with its behavior.
- (Remark: We have to compute mutants for each diagnosis candidate!)
- Hence, we compute distinguishing test cases for each pair of candidates and ask the user (or another oracle) for the expected output values.
- The problem of *distinguishing diagnosis candidates* is reduced to the problem of *computing distinguishing test cases*!

# Preliminaries

$\Pi \ldots$ Program written in any programming language

**Variable environment** is a set of tuples $(x, v)$ where $x$ is a variable and $v$ is its value

$[\![\Pi]\!](I) \ldots$ Execution of $\Pi$ on input environment $I$

$$[\![\Pi]\!](I) \supseteq O \Leftrightarrow \Pi \text{ passes test case}(I, O)$$

$$\neg(\Pi \text{ passes test case}(I, O)) \Leftrightarrow \Pi \text{ fails test case}(I, O)$$

# Distinguishing test case

### Definition (Distinguishing test case)

Given programs $\Pi$ and $\Pi'$. A test case $(I, \emptyset)$ is a distinguishing test case if and only if there is at least one output variable where the value computed when executing $\Pi$ is different from the value computed when executing $\Pi'$ on the same input $I$.

$$(I, \emptyset) \text{ is distinguishing } \Pi \text{ from } \Pi' \Leftrightarrow$$
$$\exists\, x : (x, v) \in [\![\Pi]\!](I) \wedge (x, v') \in [\![\Pi']\!](I) \wedge v \neq v'$$

# Example

```
begin
    i = 2 * x;
    j = 3 * y;
    o1 = i + j;
    o2 = i * i;
end;
```

```
begin
    i = 2 * x;
    j = 2 * y;
    o1 = i + j + 2;
    o2 = i * i;
end;
```

Orignal test case: `x = 1, y = 2, o1 = 7, o2 = 4`

Distinguishing test case: `x = 1, y = 1`

`o1 = 5, o2 = 4` `o1 = 6, o2 = 4`

# Computing distinguishing test cases

- Given two programs $\Pi_1$, $\Pi_2$
- Basic idea:
  1. Convert programs into their constraint representation
  2. Add constraints stating that the inputs have to be equivalent
  3. Add constraints stating that at least one output has to be different
  4. Use the constraint solver to compute the distinguishing test case
- *How to represent programs using constraints?*

# Converting Programs into Constraints

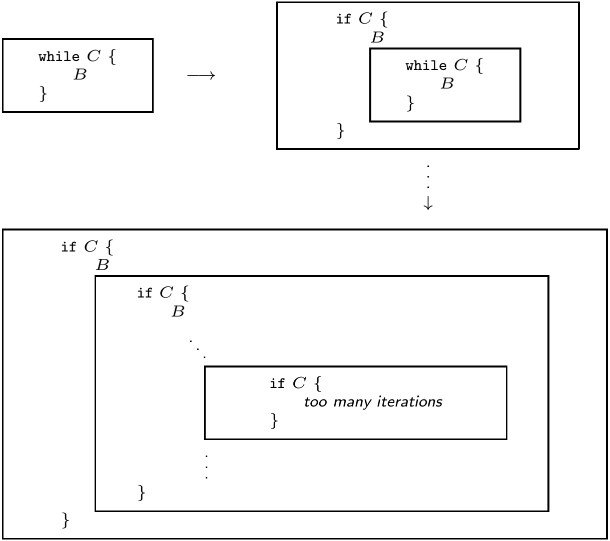
- Automated process
- Unrolling loops; Number of possible/considered iterations known in advance
- Algorithm **convert($\Pi$,$\#It$)**
  1. Unrolling the loops
  2. Computing the Static Single Assignment form (SSA)
  3. Converting the SSA program into constraints
- References:

# Constraint representation – Example

- *Original program:*

```
int power(int a, int exp)
1.   int e = exp;
2.   int res = 1;
3.   while (e > 0) {
4.     res = res * a;
5.     e = e - 1; }
6.   return res;
```

# Step 1 – Loop unrolling



# Constraint representation – Example cont.

- *Loop-free program (2 iterations):*

```
int power_loopfree(int a, int exp)
    int e = exp;
    int res = 1;
    if (e > 0) {
      res = res * a;
      e = e - 1;
      if (e > 0) {
        res = res * a;
        e = e - 1;
    return res;
    }}
```

# Step 2 – SSA representation

- Static Single Assignment form (SSA):
  - Property: Not two left-side variables have the same name!
  - Rename variables and make them unique (index).
  - Conversion of conditionals:

`if` $C$ `then` $B_1$ `else` $B2$ `end if`

    - Assign the value of $C$ to a variable, i.e., $x\_C$ `=` $C$ `;`.
    - Convert $B_1$ and $B_2$ separately (using different variable names).
    - Introduce a function $\Phi$ for each target variable:

$$x_i = \Phi(x_{index(B_1)}, x_{index(B_2)}, x\_C)$$

# Step 2 cont.

- Semantics of $\Phi$:

$$\Phi(\texttt{v\_j}, \texttt{v\_k}, \texttt{cond\_i}) \stackrel{\text{def}}{=} \begin{cases} \texttt{v\_j} & \text{if } \texttt{cond\_i} = true \\ \texttt{v\_k} & \text{otherwise} \end{cases}$$

```
int power_SSA(int a, int exp)
    int e_0 = exp;
    int res_0 = 1;
    bool cond_0 = (e_0 > 0);
    int res_1 = res_0 * a;
    int e_1 = e_0 - 1;
    bool cond_1 = cond_0 ∧ (e_1 > 0);
    int res_2 = res_1 * a;
    int e_2 = e_1 - 1;
    int res_3 = Φ(res_2, res_1, cond_1);
    int e_3 = Φ(e_2, e_1, cond_1);
    int res_4 = Φ(res_3, res_0, cond_0);
    int e_4 = Φ(e_3, e_0, cond_0);
```

# Step 3 – Conversion into Constraints

| SSA Statement | MINION Constraint |
|---|---|
| e_0 = exp; | auxVar = **ComputeExpression**(exp),<br>*eq*(e_0, auxVar) |
| cond_0 = (e_0 > 0); | *reify*(*ineq*(0,e_0,-1 ),cond_0) |
| cond_1 = cond_0 ∧ (e_1 > 0); | *reify*(*ineq*(0,e_1,-1 ),cond_aux)<br>*reify*(*watchsumgeq*([cond_0,cond_aux], 2),cond_1) |
| res_4 = Φ(res_3, res_0, cond_0); | *watched-or*(*eq*(cond_0,0), eq(res_4,res_3))<br>*watched-or*(*eq*(cond_0,1), eq(res_4,res_0)) |

# Step 3 cont.

**Algorithm ComputeExpression($\mathtt{E_{expr}}$)**

*Input:* An expression $\mathtt{E_{expr}}$ and an empty set $M$ for storing the MINION constraints.
*Output:* A set of minion constraints representing the expression stored in $M$, and a variable or constant where the result of the conversion is finally stored.

1. If $\mathtt{E_{expr}}$ is a variable or constant, then return $\mathtt{E_{expr}}$.
2. Otherwise, $\mathtt{E_{expr}}$ is of the form $\mathtt{E^1_{expr}}\ op\ \mathtt{E^2_{expr}}$.
3. Let $aux_1 =$ **ComputeExpression** ($\mathtt{E^1_{expr}}$)
4. Let $aux_2 =$ **ComputeExpression** ($\mathtt{E^2_{expr}}$)
5. Generate a new MINON variable $result$ and create MINON constraints accordingly to the given operator $op$, which define the relationship between $aux_1$, $aux_2$, and $result$, and add them to $M$.
6. Return $result$.

# Step 3 cont.

- *Example:* Given expression `a_0 + b_0 - c_0`
- Minion constraints:

```
sumleq([a_0,b_0],aux1)
sumgeq([a_0,b_0],aux1)
weightedsumleq([1,-1],[aux1,c_0], aux2)
weightedsumgeq([1,-1],[aux1,c_0], aux2)
```

# Summary conversion process

- Handles loop, conditionals, assignments, and function calls as well as arrays
- Currently not for OO constructs
- Completely automated
- To be used for testing and debugging (with some extensions)
- Correct under given restricting assumptions

*But how to compute distinguish test cases?*

# Algorithm: Compute distinguishing test case

*Inputs:* Two programs $\Pi_1$ and $\Pi_2$ having the same input variables ($IN$) and output variables ($OUT$), and a maximum number of iterations $\#It$.

*Outputs:* A distinguishing test case.

1. Call **convert(**$\Pi_1$**,**$\#It$**)** and store the result in $M_1$.
2. Call **convert(**$\Pi_2$**,**$\#It$**)** and store the result in $M_2$.
3. Rename all variables $x$ used in constraints $M_1$ to $x$_P1.
4. Rename all variables $x$ used in constraints $M_2$ to $x$_P2.
5. Let $M$ be $M_1 \cup M_2$.
6. For all input variables $x \in IN$ do:
   1. Add the constraint $x$_P1 $=$ $x$_P2 to $M$.
7. For all output variables $x \in OUT$ do:
   1. Add the constraint $x$_P1 $\neq$ $x$_P2 to $M$.
8. Return the values of the input variables obtained when calling a constraint solver on $M$ as result.

TU Graz

# Experimental results

- MINION version 0.8 constraint solver
- Maximum time for computing solutions set to 2 hours
- Only integer variables (range -250 to 250)
- Intel Pentium Dual Core 2 GHz computer, 4 GB RAM, Windows Vista
- No out-of-memory exceptions observed
- Iterations: 2, 4, and 7
- Only small programs (Note: For debugging we used programs up to 1kLOC)

# Experimental results cont.

| Name | LOC | #I/O | #It | V1 | V2 | V3 | V4 | #CO | #Var |
|---|---|---|---|---|---|---|---|---|---|
| MultATC | 12 | 2/1 | 2 | K (0,07s) | K(0,06s) | K(0,04s) | K(0,03s) | 47 | 32 |
| | | | 4 | K (0,04s) | K(0,08s) | K(0,07s) | K(0,07s) | 87 | 56 |
| | | | 7 | K (0,01s) | K(0,10s) | K(0,11s) | K(0,11s) | 151 | 92 |
| SumATC | 13 | 2/1 | 2 | K (0,4s) | K(0,03s) | K(0,4s) | K(0,4s) | 49 | 34 |
| | | | 4 | K (0,4s) | K(0,07s) | K(0,49s) | K(0,47s) | 89 | 58 |
| | | | 7 | K (0,67s) | K(0,11s) | K(0,62s) | K(0,09s) | 149 | 94 |
| MultV2ATC | 18 | 2/1 | 2 | K (0,2s) | K(0,12s) | K(0,21s) | K(0,18s) | 132 | 86 |
| | | | 4 | K (0,34s) | K(0,23s) | K(0,31s) | K(0,31s) | 418 | 258 |
| | | | 7 | K (2,09s) | K(2,09s) | K(2,15s) | K(2,15s) | 1144 | 696 |
| DivATC | 22 | 2/1 | 2 | K (0,06s) | K(0,06s) | K(0,06s) | K(0,06s) | 65 | 52 |
| | | | 4 | K (0,08s) | K(0,08s) | K(0,6s) | K(0,08s) | 105 | 76 |
| | | | 7 | K (0,10s) | K(0,10s) | K(0,09s) | K(0,12s) | 165 | 112 |
| GcdATC | 24 | 2/1 | 2 | K (0,07s) | K(0,35s) | K(46s/0,6s) | X/K(0,15s) | 126 | 90 |
| | | | 4 | K (0,08s) | K(0,08s) | X/K(0,12s) | X/K(0,5s) | 206 | 138 |
| | | | 7 | K (0,10s) | K(0,10s) | X/K(0,4s) | X/K(0,65s) | 333 | 220 |
| RandomATC | 52 | 3/1 | 2 | K (0,25s) | K(0,25s) | K(0,24s) | K(0,24s) | 303 | 213 |
| | | | 4 | K (0,8s) | K(0,8s) | K(0,8s) | K(0,8s) | 667 | 433 |
| | | | 7 | K (3,5s) | K(3,47s) | K(3,6s) | K(3,59s) | 1513 | 943 |

# Conclusions

- Computing inputs that distinguishes two implementations due to different outputs
- Automated test case generation
- Use constraints to represent the implementations
- Limitations:
  - No object-oriented constructs
  - The expected output values are not computed (oracle problem)
  - Computational complexity – Not for large programs
- Variable order has an influence on computation!
- For extending test suites
- An extension to debugging

Questions?

# Debugging using Model-based Diagnosis

- The debugging problem comprising:
  - A program:

```
begin
    i = 2 * x;
    j = 2 * y;
    o1 = i + j;
    o2 = i * i;
end;
```

  - At least one test case:

```
x = 1, y = 2, o1 = 8, o2 = 4
```

- Basic idea: Introduce a predicate allowing to state correctness / incorrectness of programs.

# Debugging cont.

- Introduce predicates

```
      [ x_0 = 1, y_0 = 2 ]
1.    begin
2.       ¬AB₂ ∨ i_1 = 2 * x_0;
3.       ¬AB₃ ∨ j_1 = 2 * y_0;
4.       ¬AB₄ ∨ o1_1 = i_1 + j_1;
5.       ¬AB₅ ∨ o2_1 = i_1 * i_1;
6.    end;
      [ o1_1 = 8, o2_1 = 4 ]
```

- When considering assignment as equations / constraints, we can use a constraint solver to set values for $AB_i$ such that all constraints are fulfilled.
- Debugging becomes constraint solving.

# Debugging and distinguishing test cases

- *But is this all we need in order to use distinguishing test cases?*
- **NO!** But we can do the following:
  - Focus on statements identified to be diagnosis candidates and ignore all others.
  - Compute (all) mutations for the interesting statements.
  - Distinguish mutants using distinguishing test cases.