Does testing help to reduce the number of potentially faulty statements in debugging?

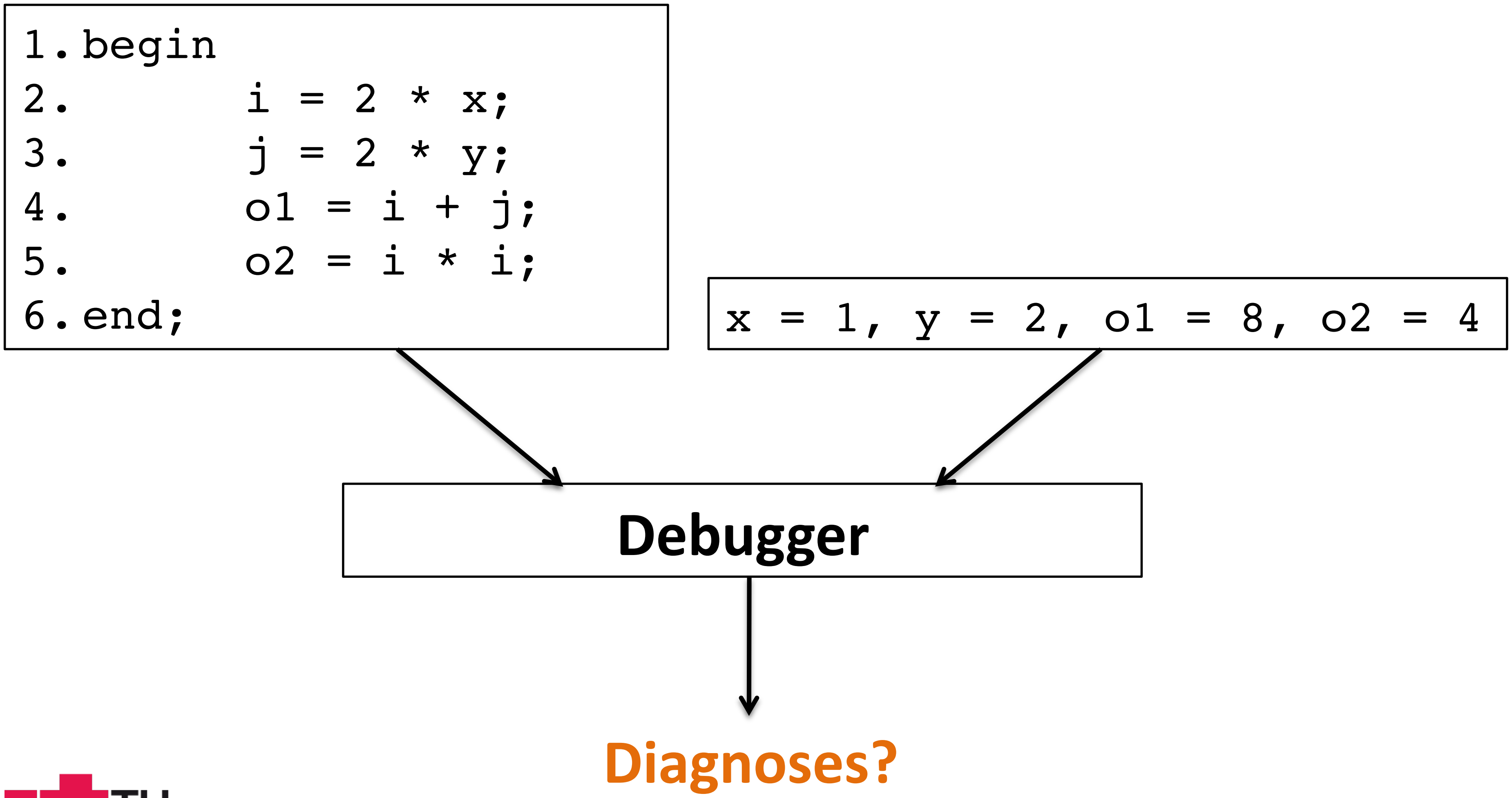
Mihai Nica, Simona Nica, and **Franz Wotawa**
Technische Universität Graz
Institute for Software Technology
{mnica,snica,wotawa}@ist.tugraz.at

The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian FederalMinistry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsf¨orderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT) and the Austrian Science Fund (FWF) under contract number P20199-N15

Motivation

- A lot of research in automated debugging (but maybe not enough), e.g.,
  - *Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs, ICST 2010 introducing* **possible fixes**.
- Using mutations or genetic programming
- There are too many possible fixes!
- Reducing the number of possible fixes via testing

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
Diagnoses?

Debugging using constraints

```
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
```

Programm execution

$Ab(2) \vee \; i = 2 * x;$
$Ab(3) \vee \; j = 2 * y;$
$Ab(4) \vee \; o1 = i + j;$
$Ab(5) \vee \; o2 = i * i;$

$x = 1$
$y = 2$
$o1 = 8$
$o2 = 4$

Constraint solving / equation solving

Finding bugs using constraints

$Ab(2) \vee$ ~~$i = 2 * x;$~~
$Ab(3) \vee$ ~~$j = 2 * y;$~~
$Ab(4) \vee \; o1 = i + j;$
$Ab(5) \vee \; o2 = i * i;$

$x = 1$
$y = 2$
$o1 = 8$
$o2 = 4$

$Ab(2) \wedge \neg Ab(3) \wedge \neg Ab(4) \wedge \neg Ab(5)$

j = 2 * 2 = 4
o1 = i + j = 8 = i + 4 → i = 4
o2 = 4 = i * i = 4 * 4 → **FAIL!!!!**

$\neg Ab(2) \wedge Ab(3) \wedge \neg Ab(4) \wedge \neg Ab(5)$

i = 2 * 1 = 2
o1 = 8 = 2 + j → j = 6
o2 = 4 = i * i = 2 * 2

And so on ... finally leading to 2 possible diagnoses statement 3 and statement 4

What we have reached...

- Automated debugging using constraints and the Ab predicates

- But: How to handle recursions, loops, conditionals, multiple definitions of the same variable,...???

Handling loops

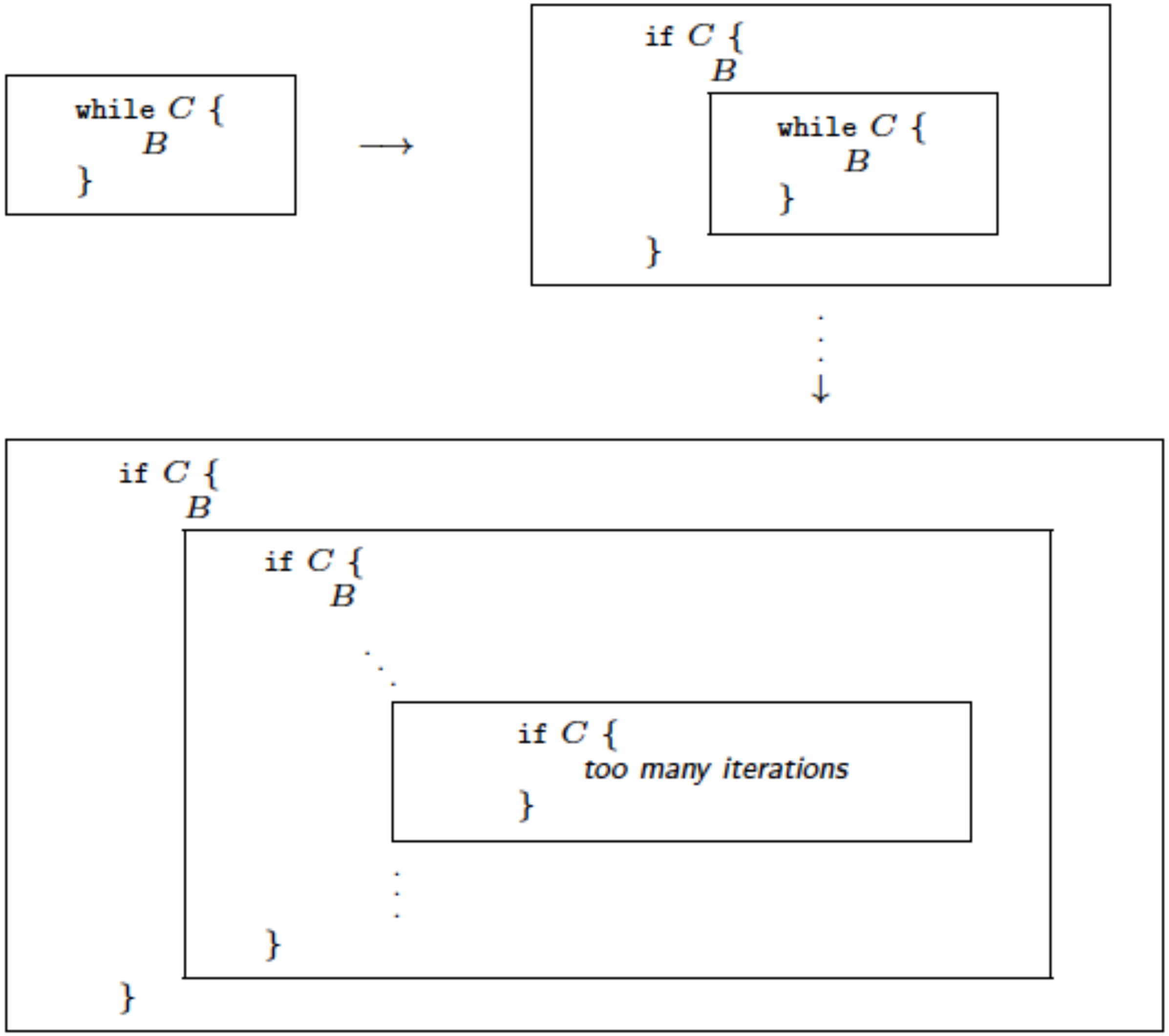
- Execution of

```
while (e > 0) { ... }
```

leads to:

```
if (e > 0) { ...
    if (e > 0) { ...
        if (e > 0) { ... }}}
```

Loop unrolling



Static single assignment form (SSA form)

- In order to convert programs to constraints every variable is only allowed to be defined once!

- **Solution**: convert the loop-free program into its SSA form

SSA form

- **Property**: No two left-side (=defined) variables have the same name

- Assign each defined variable an unique index.
- If a variable is used afterwards in the program, refer to the last given index.

Conditional statements

- Statement of the form

  `if` C `then` B_1 `else` B_2 `end if;`

- Convert B_1 and B_2 separately using a distinguished set of indices

Conditional statements

- Introduce a new function Φ.
- Add a new statement

```
x_C = C;
```

- For each defined variable x in either B1 or B2 add the following assignment:

x_i = Φ(x_index(B_1),x_index(B2),x_C);

Semantics of Φ

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

So debugging using constraints is possible for general programs...

- But there are some challenges remaining:
  - OO constructs
  - Reducing the number of bug candidates
  - Providing information about how to correct programs
  - ...

Correcting programs ...
... using mutations

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

```
j = 3 * y;

 ...

o1 = i + j + 2;
```

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

Possible fixes can be used to reduce the number of possible diagnoses!

- Given:
  – Program
  – Test suite
  – Mutations of the program wrt. given diagnoses
- If there is no mutation of a diagnosis that passes the test suite, remove the diagnosis from the list of possible diagnoses!

Other possiblity for removing diagnoses is to use distinguishing test cases

- 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.
  - 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!

Some definitions

$\Pi \ldots$ 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 Π on input environment I

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

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

Def. distinguishing test case

Given programs Π and Π'. 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 Π is different from the value computed when executing Π' 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 (cont.)

Mutant 1

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

Mutant 2

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

`x = 1, y = 2, o1 = 8, o2 = 4` **Original test case**

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

`o1 = 5, o2 = 4` (Mutant 1)

`o1 = 6, o2 = 4` (Mutant 2)

Computing distinguishing test cases

- Given two programs.

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

Bringing it all together...

1. Convert the program into its constraint representation
2. Compute all possible diagnoses using the given test suite
3. Compute the mutations for the obtained diagnosis and remove those mutants that are in contradiction with at least one test case.
4. Filter the obtained diagnoses using the remaining mutations.
5. Select two mutations and compute the distinguishing test case.
6. Ask the user about the expected output values.
7. Add the distinguishing test cases including the expected outputs to the test suite
8. Remove all mutants that do not pass the new test. If the number of remaining diagnoses is sufficently small stop. Otherwise, go to 5.

Empirical results

Name	It	Var_{Π}	LOC_{Π}	Inputs	Outputs	LOC_{SSA}	$\|CO\|$	Var_{CO}	$\|Diag\|$	$\|Diag_{filt}\|$	#UI	$\|Diag_{TC}\|$
DivATC_V1	2	5	21	2	1	32	33	29	3	2	1	2
DivATC_V2	2	5	21	2	1	32	33	29	5	3	1	1
DivATC_V3	2	5	21	2	1	32	33	29	3	2	1	2
DivATC_V4	2	5	21	2	1	32	33	29	4	4	1/2	3(1)/1
GcdATC_V1	2	6	35	2	1	49	61	46	2	2	1	1
GcdATC_V2	2	6	35	2	1	49	61	46	10	3	1/2/3/4/5	3/3/2/2/1
GcdATC_V3	2	6	35	2	1	49	61	46	2	2	1	1
MultATC_V1	2	5	16	2	1	26	24	19	2	2	1	1
MultATC_V2	2	5	16	2	1	26	24	19	2	2	1	1
MultATC_V3	2	5	16	2	1	26	24	19	2	2	1	1
MultATC_V4	2	5	16	2	1	26	24	19	5	2	1	1
MultV2ATC_V1	2	6	20	2	1	49	67	46	6	2	1	1
MultV2ATC_V2	2	6	20	2	1	49	67	46	2	1	1	1
MultV2ATC_V3	2	6	20	2	1	49	67	46	6	1	1	1
SumATC_V1	2	5	18	2	1	27	24	20	2	2	1	1
SumATC_V2	2	5	18	2	1	27	24	20	3	2	1	1
SumATC_V3	2	5	18	2	1	27	24	20	5	2	1	1
SumPowers_V1	2	11	36	3	1	72	87	70	16	6	1/2/3/4	4/4/2/2
SumPowers_V2	2	11	36	3	1	72	87	70	11	6	1/2	2/1
SumPowers_V3	2	11	36	3	1	72	87	70	11	1	1	1
tcas08	1	48	125	12	1	125	98	132	27	13	1/2/3/4	11/11/11/10
tcas03	1	48	125	12	1	125	98	132	27	13	1/2/3/4	13/12/9/9

Conclusion

- **Does testing help to reduce the number of potentially faulty statements in debugging?**
- **Answer: YES!**

- Debugging = Constraint solving
- Mutations for obtaining corrections
- Distinguishing test cases for reducing diagnoses

Related Literature

- Wotawa, F., Nica, M., Aichernig, B.K.: Generating distinguishing tests using the minion constraint solver. In: CSTVA 2010: Proceedings of the 2nd Workshop on Constraints for Testing, Verification and Analysis, IEEE (2010).
- Ceballos, R., Nica, M., Weber, J., Wotawa, F.: On the complexity of program debugging using constraints for modeling the program's syntax and semantics. In: In *Proc. Conference of the Spanish Association for Artificial Intelligence (CAEPIA), Seville, Spain (2009).*
- Wotawa, F., Nica, M., Moraru, I., Automated Debugging based on a Constraint Model of the Program and a Test Case, Currently under Review.

Thank you for your attention!

http://paginas.fe.up.pt/~tebug2011/