TU Graz

Automated debugging – the past, the now, and the future

Part 2: Debugging based on models

Franz Wotawa

TU Graz, Institute for Software Technology

wotawa@ist.tugraz.at

FWF Der Wissenschaftsfonds.

FFG

Christian Doppler Forschungsgesellschaft

Content

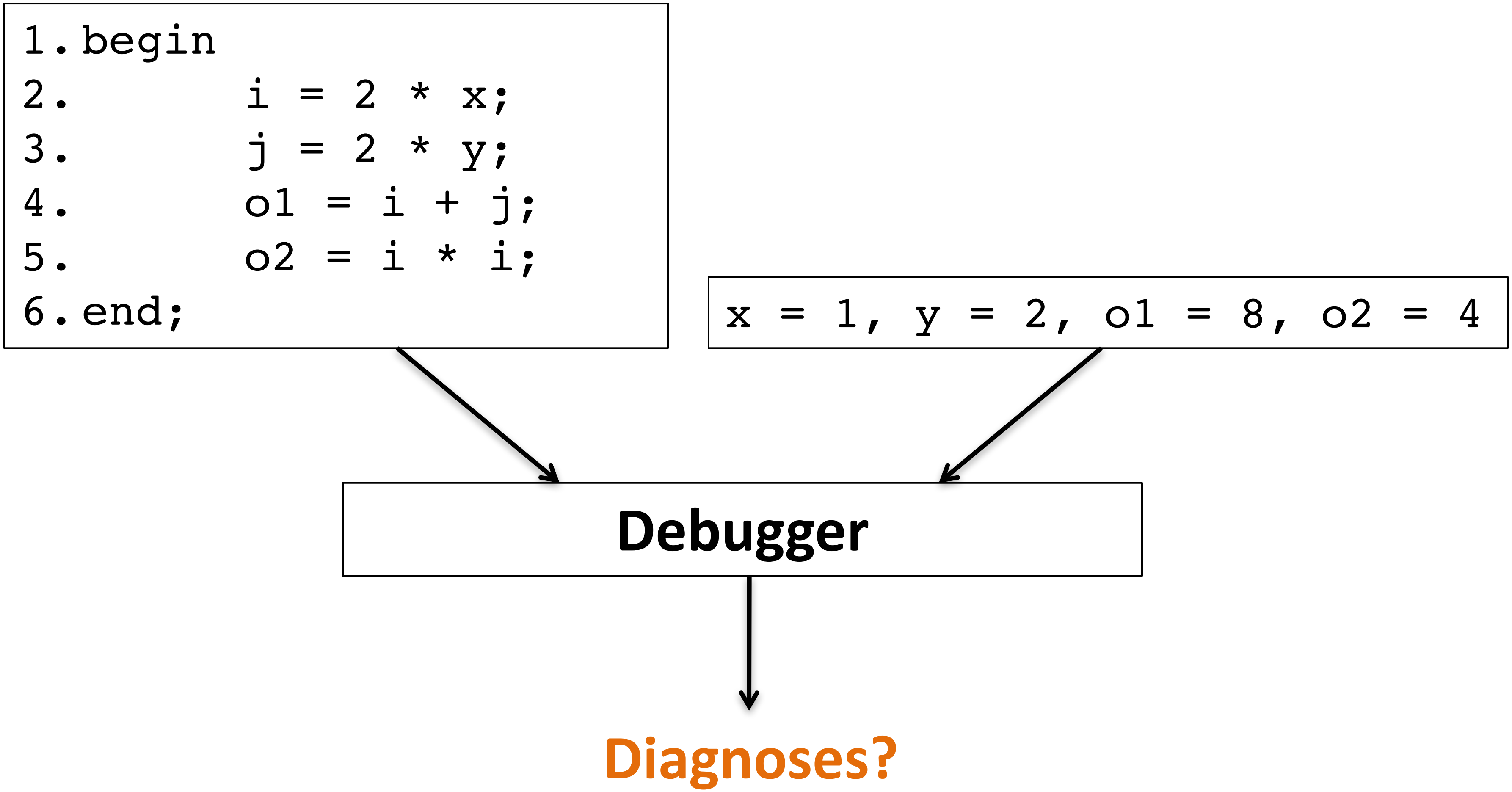
- The debugging problem
- Conversion of programs into constraints
- Specification knowledge / handling functions
- Testing

- Joint work with: Bernhard Aichernig, R. Ceballos, Gerhard Friedrich, Wolfgang Maier, Julia Nica, Mihai Nica, Simona Nica, Ingo Pill, Markus Stumptner, Jörg Weber, Dominik Wieland.

The debugging problem

- **Given**:
  - Source code of a program
  - A test suite comprising at least one failing test case
- **Wanted**:
  - Root cause for the detected misbehavior (statement, expression,..)

Debugging – A (very) short intro



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

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

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

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

$j = 2 * 2 = 4$
$o1 = i + j = 8 = i + 4 \rightarrow \; i = 4$
$o2 = 4 = i * i = 4 * 4 \rightarrow$ **FAIL!!!!**

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

$i = 2 * 1 = 2$
$o1 = 8 = 2 + j \rightarrow \; j = 6$
$o2 = 4 = i * i = 2 * 2$

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

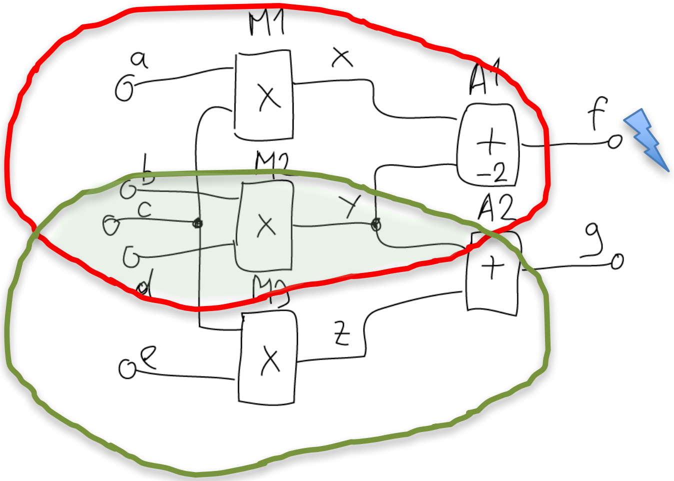
Observations

- Reasoning in „all directions“ (from input to outputs and vice versa)

- Make assumptions about the correctness of „components“

- Use in-consistencies for accepting or refuting assumptions

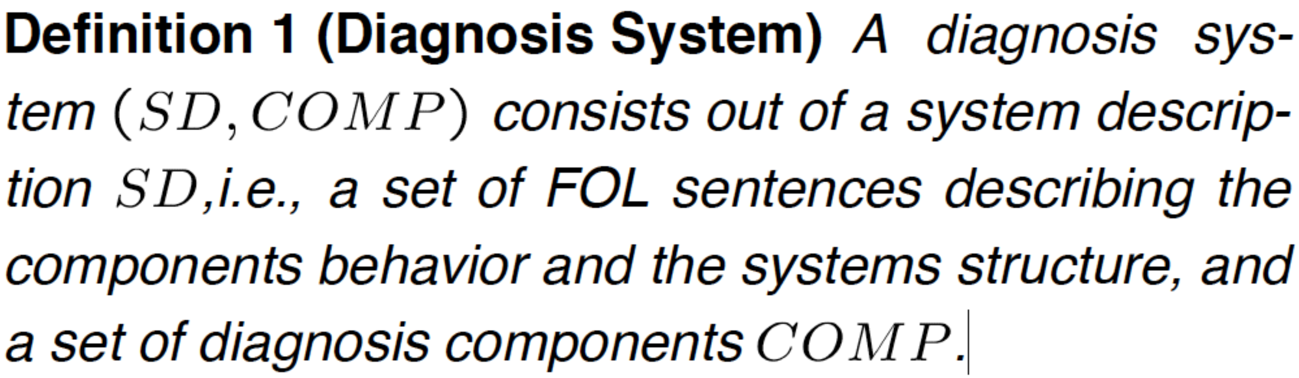
Additional remarks



We might try to find root causes by tracing back dependencies and eliminating candidates that also contribute to correct output values.

This should NOT be done because of failure masking:

Basic definitions



Definition 1 (Diagnosis System) *A diagnosis system* $(SD, COMP)$ *consists out of a system description* SD*,i.e., a set of FOL sentences describing the components behavior and the systems structure, and a set of diagnosis components* $COMP$*.*

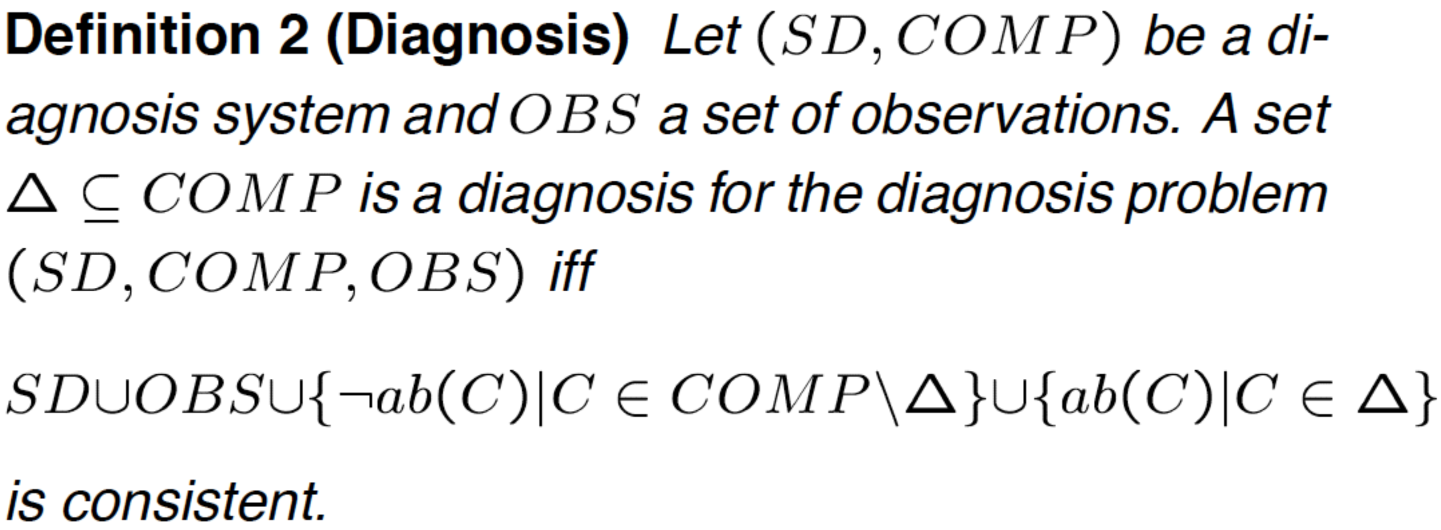
$\neg AB(M1) \rightarrow x = a * c$ (or $AB(M1) \vee x = a * c$)

$\neg AB(M2) \rightarrow y = b * d$

....

AB…Abnormal / Assumption

Diagnosis



Definition 2 (Diagnosis) *Let* $(SD, COMP)$ *be a diagnosis system and* OBS *a set of observations. A set* $\Delta \subseteq COMP$ *is a diagnosis for the diagnosis problem* $(SD, COMP, OBS)$ *iff*

$$SD \cup OBS \cup \{\neg ab(C) | C \in COMP \setminus \Delta\} \cup \{ab(C) | C \in \Delta\}$$

is consistent.

What is needed?

- Mapping of programs to a model!

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

$Ab(2) \vee\ i = 2 * x;$

$Ab(3) \vee\ j = 2 * y;$

$Ab(4) \vee\ o1 = i + j;$

$Ab(5) \vee\ o2 = i * i;$

Debugging / Testing

CONVERTING PROGRAMS TO CONSTRAINTS

Assumptions

- Sequential programming language without OO constructs
- The program terminates
- No exception handling

Challenges

- Loops / recursive function calls
- Variables defined more than once in a program

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

Handling loops

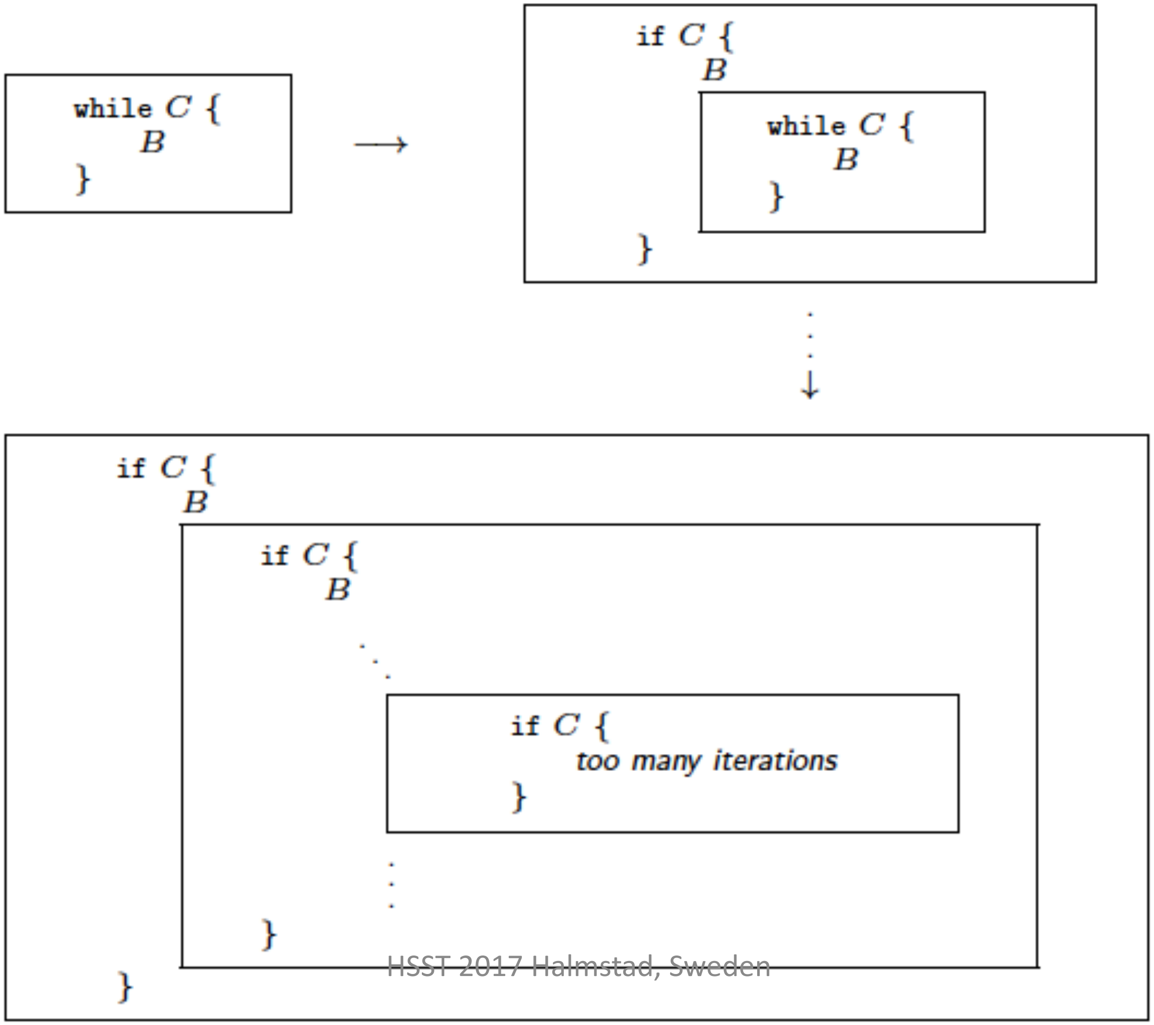
- Execution of

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

  leads to:

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

Loop unrolling



```
while C {
    B
}
```

→

```
if C {
    B
    while C {
        B
    }
}
```

↓

```
if C {
    B
    if C {
        B
          ⋱
            if C {
                too many iterations
            }
            ⋮
    }
}
```

Summary loop unrolling

- No influence on semantics if nesting depth set appropriately
  - Nesting depth > maximum number of iterations caused by a test case
- Increase in size of the pogram (accordingly to the complexity of the program)

Example (cont.)

```
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;
```

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(B1),x_index(B2),x_C);
```

Semantics of Φ

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

Example (cont.)

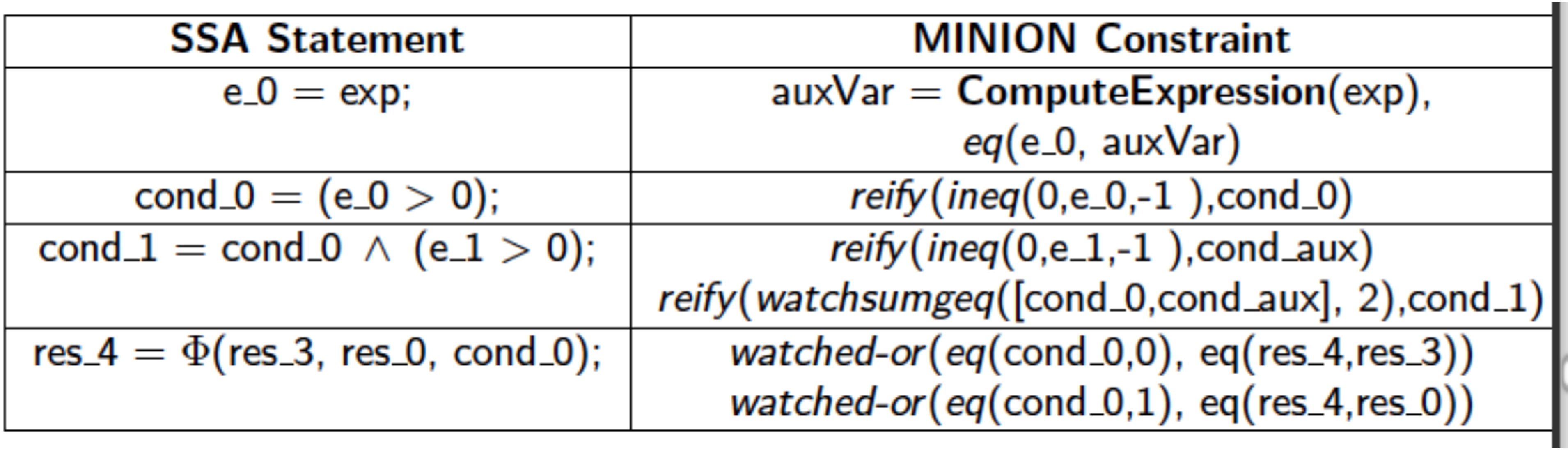
```
int power_SSA(int a, int exp) {
int e_0 = exp;
int res_0 = 1;
bool cond_0 = (e_0 > 0);
int res_1 = res_0 * a;
int e_1 = e_0 - 1;
bool cond_1 = cond_0 ∧(e_1 > 0);
int res_2 = res_1 * a;
int e_2 = e_1 - 1;
int res_3 = Φ(res_2, res_1, cond_1);
int e_3 = Φ(e_2, e_1, cond_1);
int res_4 = Φ(res_3, res_0,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

Summary SSA conversion

- Only assignment statements!
- Direct conversion to constraints possible
- The conditions used in the Φ function are equivalent to the path conditions
- No substantial increase of size

Conversion to CSPs

- Convertion only needed for assignments



SSA Statement	**MINION Constraint**
e_0 = exp;	auxVar = **ComputeExpression**(exp), *eq*(e_0, auxVar)
cond_0 = (e_0 > 0);	*reify*(*ineq*(0,e_0,-1),cond_0)
cond_1 = cond_0 $\wedge$ (e_1 > 0);	*reify*(*ineq*(0,e_1,-1),cond_aux) *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)) *watched-or*(*eq*(cond_0,1), eq(res_4,res_0))

ComputeExpression

- *Input:* An expression 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.

ComputeExpression (cont.)

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

Example

- **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

- Conversion in 3 steps:
  1. Convert program to loop-free variant (loop unrolling)
  2. Convert loop-free variant to SSA form
  3. Convert SSA form to constraint system (Minion)

Debugging / Testing

USING CONSTRAINTS FOR DEBUGGING

Input to the debugging problem

- A debugging problem comprises
  - A program, and
  - A failing test case

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

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

Introduce new variable *Ab*

- Use variable to state whether a statement is assumed to work correctly or not!

1. $Ab_1 \vee i = 2 * x;$
2. $Ab_2 \vee j = 2 * y;$
3. $Ab_3 \vee o1 = i + j;$
4. $Ab_4 \vee o2 = i * i;$

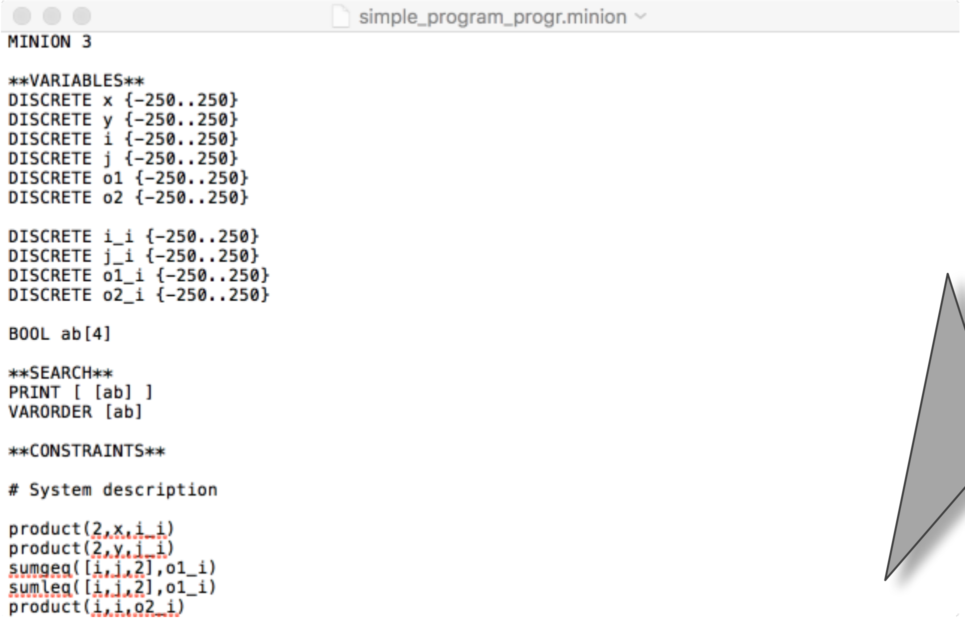
Debugging = CSP solving

1. $Ab_1 \vee$ `i_1 = 2 * x;`
2. $Ab_2 \vee$ `j_1 = 2 * y;`
3. $Ab_3 \vee$ `o1_1 = i_1 + j_1;`
4. $Ab_4 \vee$ `o2_1 = i_1 * i_1;`

x = 1, y = 2, o1_1 = 8, o2_1 = 4;

Convert to constraints

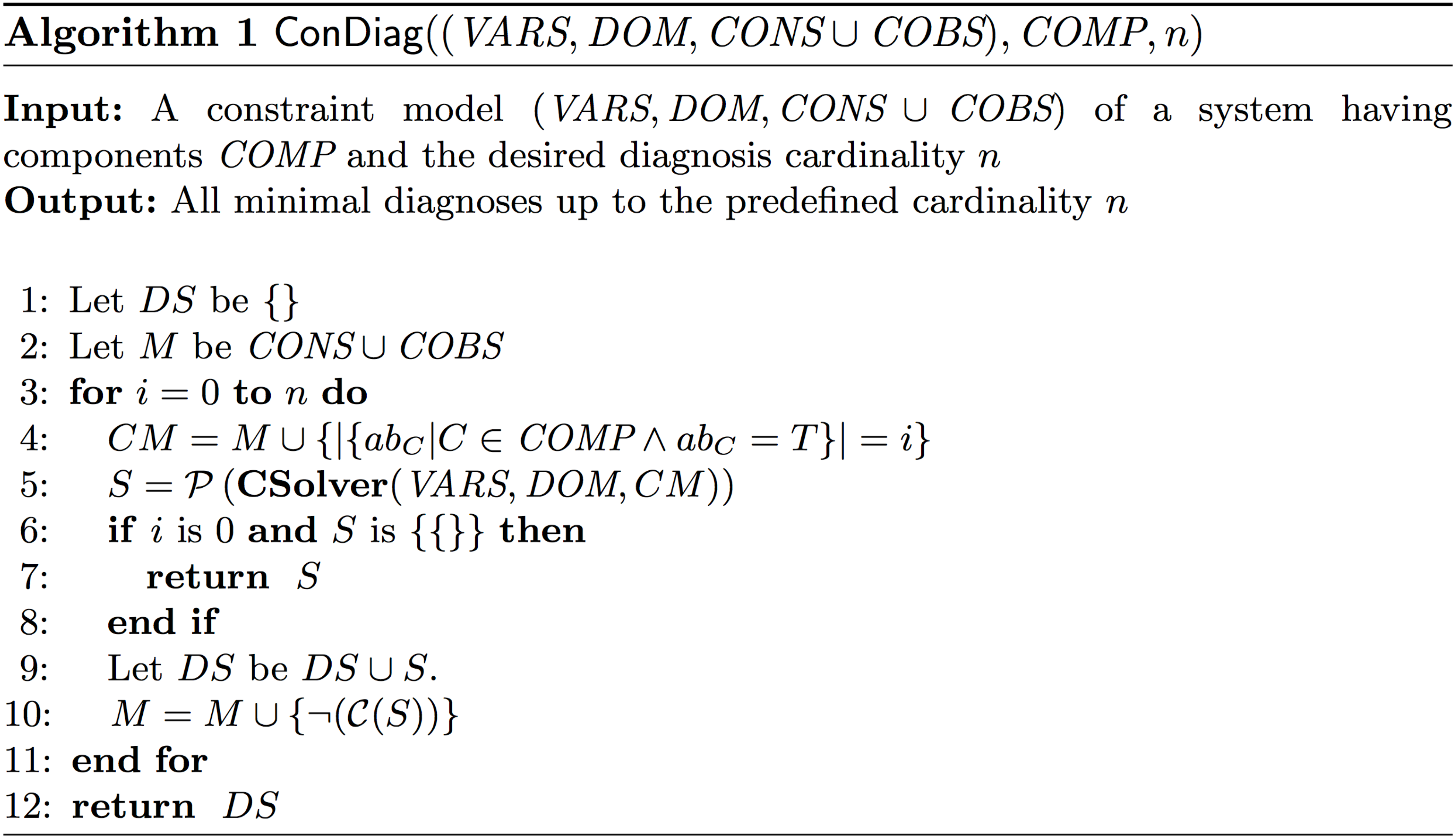
A solution to the resulting CSP is a solution to the debugging problem!



```
MINION 3

**VARIABLES**
DISCRETE x {-250..250}
DISCRETE y {-250..250}
DISCRETE i {-250..250}
DISCRETE j {-250..250}
DISCRETE o1 {-250..250}
DISCRETE o2 {-250..250}

DISCRETE i_i {-250..250}
DISCRETE j_i {-250..250}
DISCRETE o1_i {-250..250}
DISCRETE o2_i {-250..250}

BOOL ab[4]

**SEARCH**
PRINT [ [ab] ]
VARORDER [ab]

**CONSTRAINTS**

# System description

product(2,x,i_i)
product(2,y,j_i)
sumgeq([i,j,2],o1_i)
sumleq([i,j,2],o1_i)
product(i,i,o2_i)
```

Tool demo

Diagnosis algorithm



Algorithm 1 ConDiag($(VARS, DOM, CONS \cup COBS), COMP, n$)

Input: A constraint model $(VARS, DOM, CONS \cup COBS)$ of a system having components $COMP$ and the desired diagnosis cardinality n

Output: All minimal diagnoses up to the predefined cardinality n

```
Let $DS$ be $\{\}$
Let $M$ be $CONS \cup COBS$
for $i = 0$ to $n$ do
    $CM = M \cup \{|\{ab_C | C \in COMP \wedge ab_C = T\}| = i\}$
    $S = \mathcal{P}(\textbf{CSolver}(VARS, DOM, CM))$
    if $i$ is 0 and $S$ is $\{\{\}\}$ then
        return $S$
    end if
    Let $DS$ be $DS \cup S$.
    $M = M \cup \{\neg(\mathcal{C}(S))\}$
end for
return $DS$
```

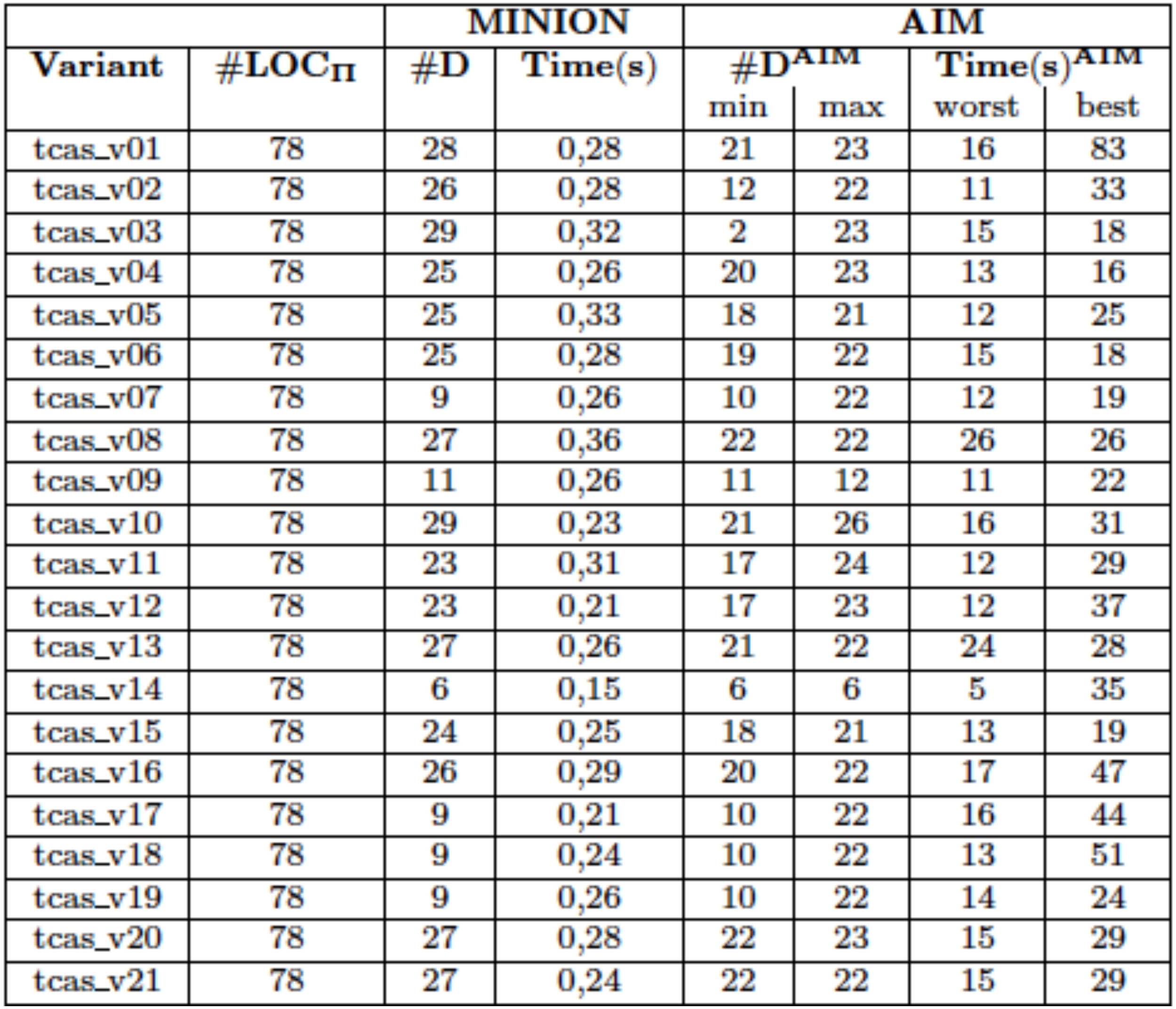
Nica, I., Wotawa, F.: Condiag - computing minimal diagnoses using a constraint solver. In: Proceedings of the International Workshop on Principles of Diagnosis (DX). pp. 185-191 (2012)

Some remarks

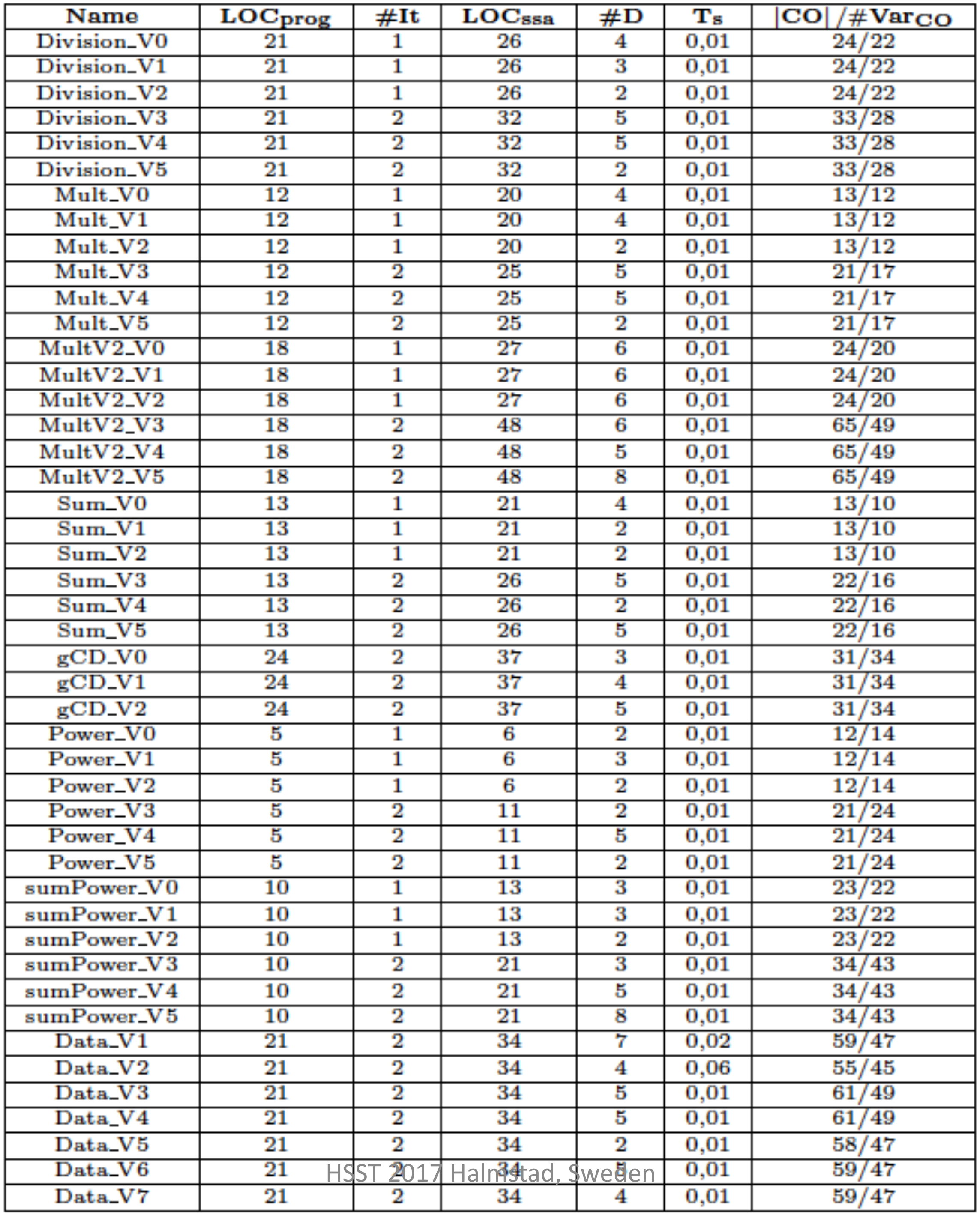
- Focus on small solutions (single faults)
  - Use constraint solver that searches for solutions where only on *Ab* variable is true!
- There must be a mapping back from the Ab variables to the statements of the original program

Results obtained

- Java implementation of the conversion process
- Use Minion V0.8 as constraint solver
- Intel Pentium Dual Core 2 GHz with 4 GB of RAM.
- AIM is a model-based debugging tool based on abstract interpretation (from Wolfgang Mayer, Markus Stumptner)



Variant	#LOC$_\Pi$	MINION #D	MINION Time(s)	AIM #D^{AIM} min	AIM #D^{AIM} max	AIM Time(s)^{AIM} worst	AIM Time(s)^{AIM} best
tcas_v01	78	28	0,28	21	23	16	83
tcas_v02	78	26	0,28	12	22	11	33
tcas_v03	78	29	0,32	2	23	15	18
tcas_v04	78	25	0,26	20	23	13	16
tcas_v05	78	25	0,33	18	21	12	25
tcas_v06	78	25	0,28	19	22	15	18
tcas_v07	78	9	0,26	10	22	12	19
tcas_v08	78	27	0,36	22	22	26	26
tcas_v09	78	11	0,26	11	12	11	22
tcas_v10	78	29	0,23	21	26	16	31
tcas_v11	78	23	0,31	17	24	12	29
tcas_v12	78	23	0,21	17	23	12	37
tcas_v13	78	27	0,26	21	22	24	28
tcas_v14	78	6	0,15	6	6	5	35
tcas_v15	78	24	0,25	18	21	13	19
tcas_v16	78	26	0,29	20	22	17	47
tcas_v17	78	9	0,21	10	22	16	44
tcas_v18	78	9	0,24	10	22	13	51
tcas_v19	78	9	0,26	10	22	14	24
tcas_v20	78	27	0,28	22	23	15	29
tcas_v21	78	27	0,24	22	22	15	29



Name	$\mathrm{LOC}_{\mathrm{prog}}$	#It	$\mathrm{LOC}_{\mathrm{ssa}}$	#D	T_{s}	$\lvert\mathrm{CO}\rvert / \#\mathrm{Var}_{\mathrm{CO}}$
Division_V0	21	1	26	4	0,01	24/22
Division_V1	21	1	26	3	0,01	24/22
Division_V2	21	1	26	2	0,01	24/22
Division_V3	21	2	32	5	0,01	33/28
Division_V4	21	2	32	5	0,01	33/28
Division_V5	21	2	32	2	0,01	33/28
Mult_V0	12	1	20	4	0,01	13/12
Mult_V1	12	1	20	4	0,01	13/12
Mult_V2	12	1	20	2	0,01	13/12
Mult_V3	12	2	25	5	0,01	21/17
Mult_V4	12	2	25	5	0,01	21/17
Mult_V5	12	2	25	2	0,01	21/17
MultV2_V0	18	1	27	6	0,01	24/20
MultV2_V1	18	1	27	6	0,01	24/20
MultV2_V2	18	1	27	6	0,01	24/20
MultV2_V3	18	2	48	6	0,01	65/49
MultV2_V4	18	2	48	5	0,01	65/49
MultV2_V5	18	2	48	8	0,01	65/49
Sum_V0	13	1	21	4	0,01	13/10
Sum_V1	13	1	21	2	0,01	13/10
Sum_V2	13	1	21	2	0,01	13/10
Sum_V3	13	2	26	5	0,01	22/16
Sum_V4	13	2	26	2	0,01	22/16
Sum_V5	13	2	26	5	0,01	22/16
gCD_V0	24	2	37	3	0,01	31/34
gCD_V1	24	2	37	4	0,01	31/34
gCD_V2	24	2	37	5	0,01	31/34
Power_V0	5	1	6	2	0,01	12/14
Power_V1	5	1	6	3	0,01	12/14
Power_V2	5	1	6	2	0,01	12/14
Power_V3	5	2	11	2	0,01	21/24
Power_V4	5	2	11	5	0,01	21/24
Power_V5	5	2	11	2	0,01	21/24
sumPower_V0	10	1	13	3	0,01	23/22
sumPower_V1	10	1	13	3	0,01	23/22
sumPower_V2	10	1	13	2	0,01	23/22
sumPower_V3	10	2	21	3	0,01	34/43
sumPower_V4	10	2	21	5	0,01	34/43
sumPower_V5	10	2	21	8	0,01	34/43
Data_V1	21	2	34	7	0,02	59/47
Data_V2	21	2	34	4	0,06	55/45
Data_V3	21	2	34	5	0,01	61/49
Data_V4	21	2	34	5	0,01	61/49
Data_V5	21	2	34	2	0,01	58/47
Data_V6	21	2	34	5	0,01	59/47
Data_V7	21	2	34	4	0,01	59/47

Remarks

- Debugging using constraints is feasible for smaller programs (e.g., at the method level)
- Pre and post conditions can be easily integrated as well as loop invariants
- The quality of the results (e.g., number of statements) depend on the underlying model

Remarks (cont.)

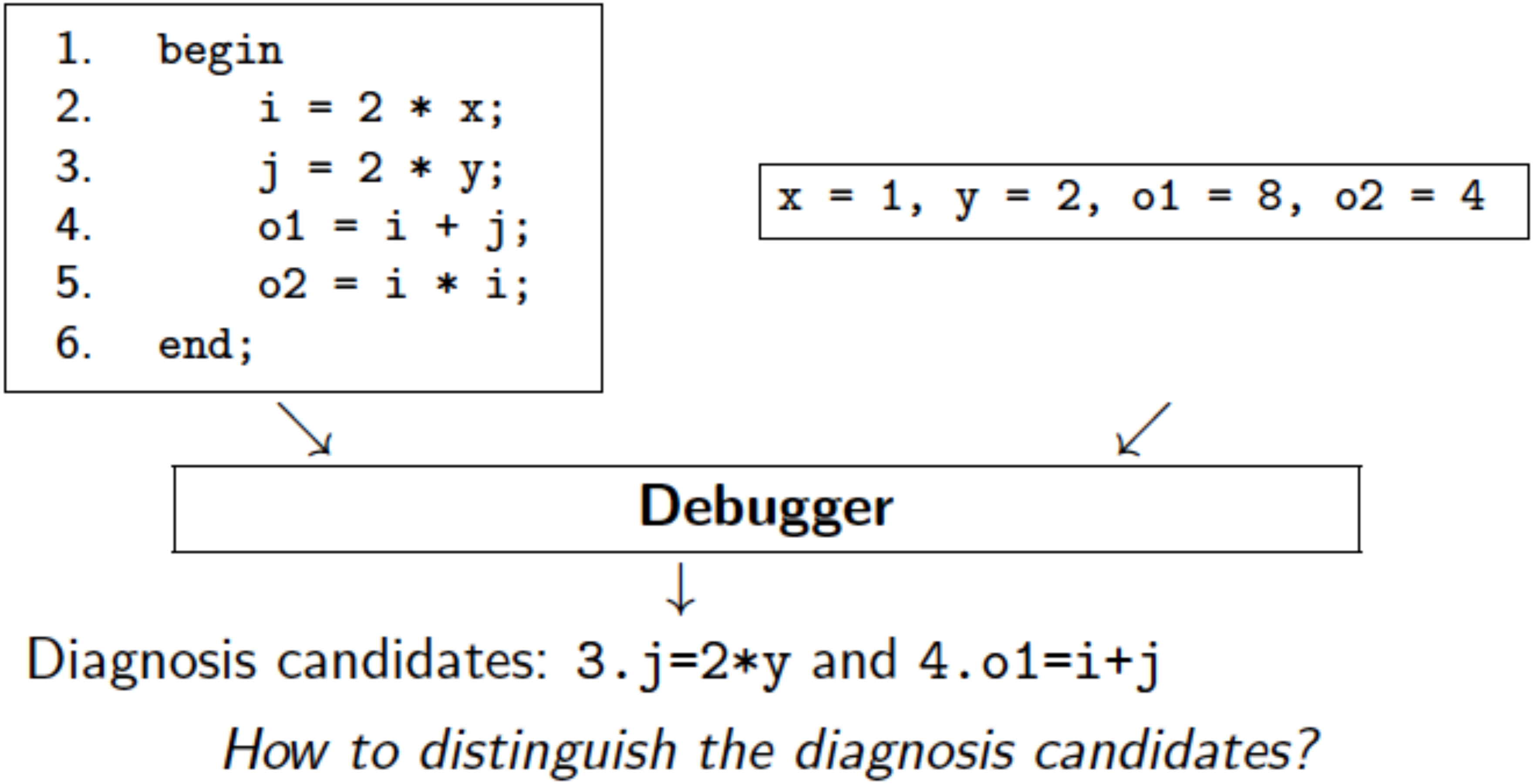
- In order to distinguish diagnosis candidates new knowledge is necessary:
  - Knowledge about intermediate values
  - Specification knowledge
  - **New test cases!**

DISTINGUISHING TEST CASES

Motivation

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

Motivation (cont.)



```
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?

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.

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

Example (cont.)

```
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.

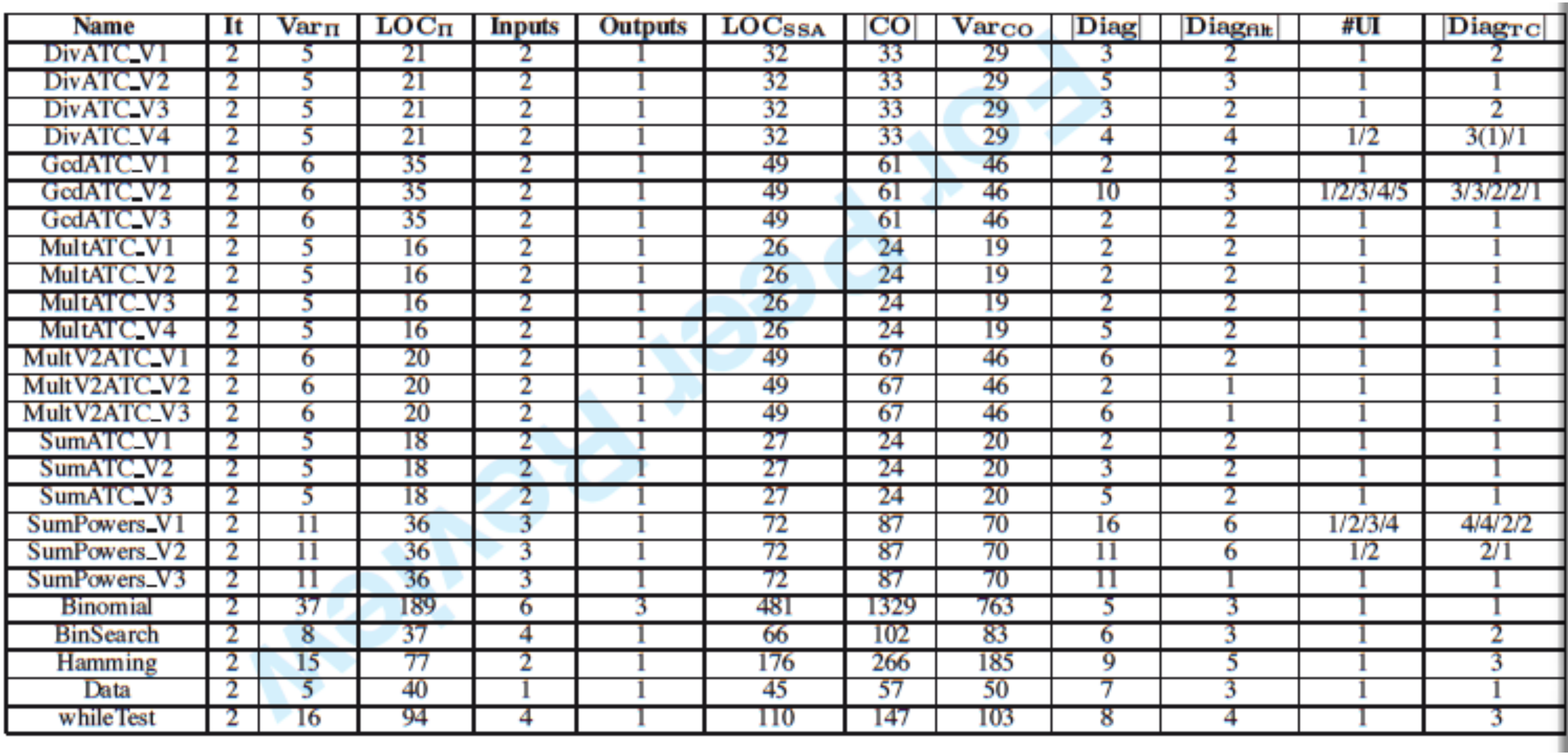
1. Convert programs into their constraint representation
2. Add constraints stating that the inputs have to be equivalent
3. Add constraints stating that at least one output has to be different
4. Use the constraint solver to compute the distinguishing test case

Inputs: Two programs Π_1 and Π_2 having the same input variables (IN) and output variables (OUT), and a maximum number of iterations $\#It$.

Outputs: A distinguishing test case.

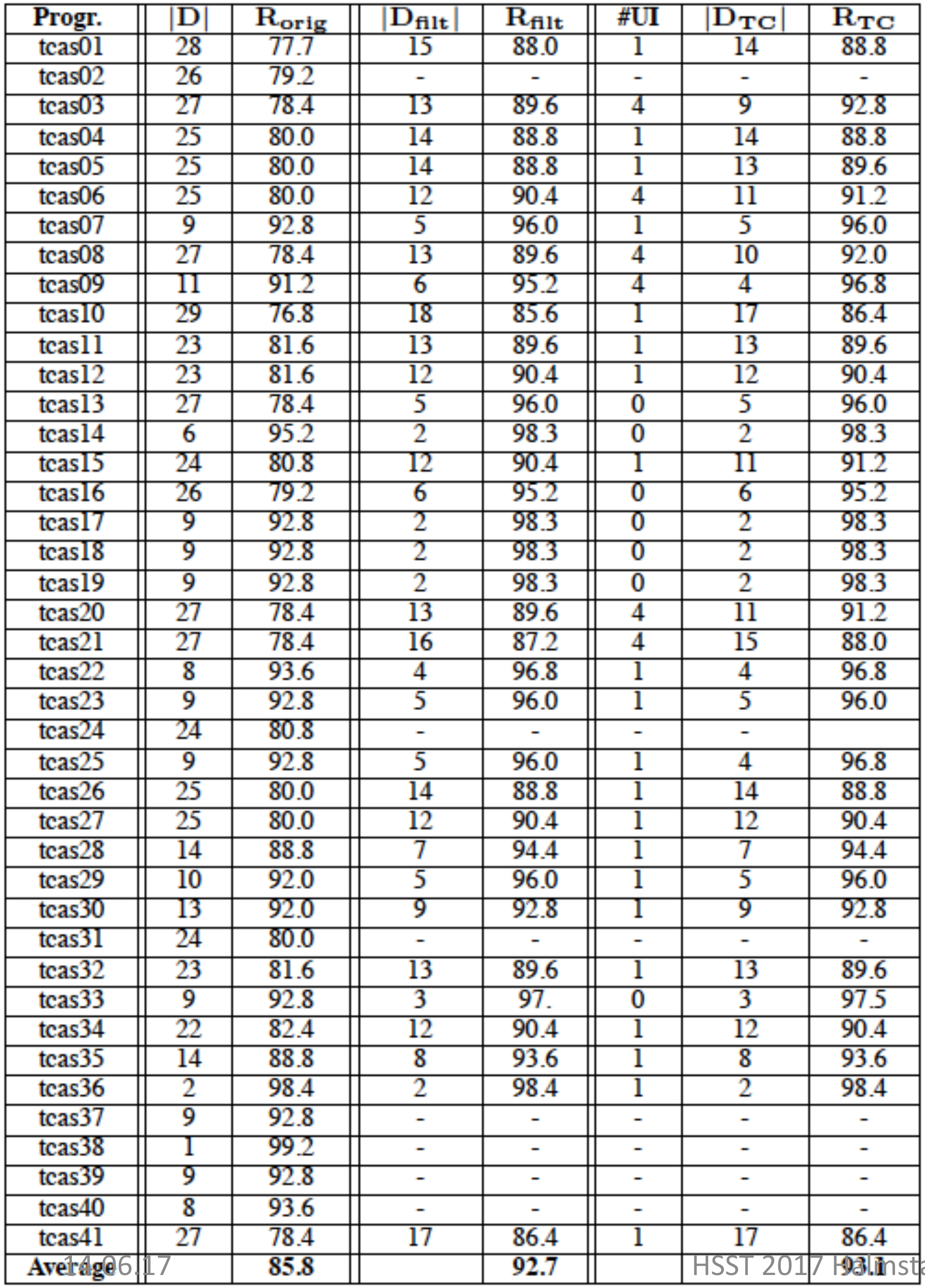
1. Call **convert**$(\Pi_1, \#It)$ and store the result in M_1.
2. Call **convert**$(\Pi_2, \#It)$ and store the result in M_2.
3. Rename all variables x used in constraints M_1 to x_P1.
4. Rename all variables x used in constraints M_2 to x_P2.
5. Let M be $M_1 \cup M_2$.
6. For all input variables $x \in IN$ do:
    1. Add the constraint $x\_\mathrm{P1} = x\_\mathrm{P2}$ to M.
7. For all output variables $x \in OUT$ do:
    1. Add the constraint $x\_\mathrm{P1} \neq x\_\mathrm{P2}$ to M.
8. Return the values of the input variables obtained when calling a constraint solver on M as result.

Experimental results

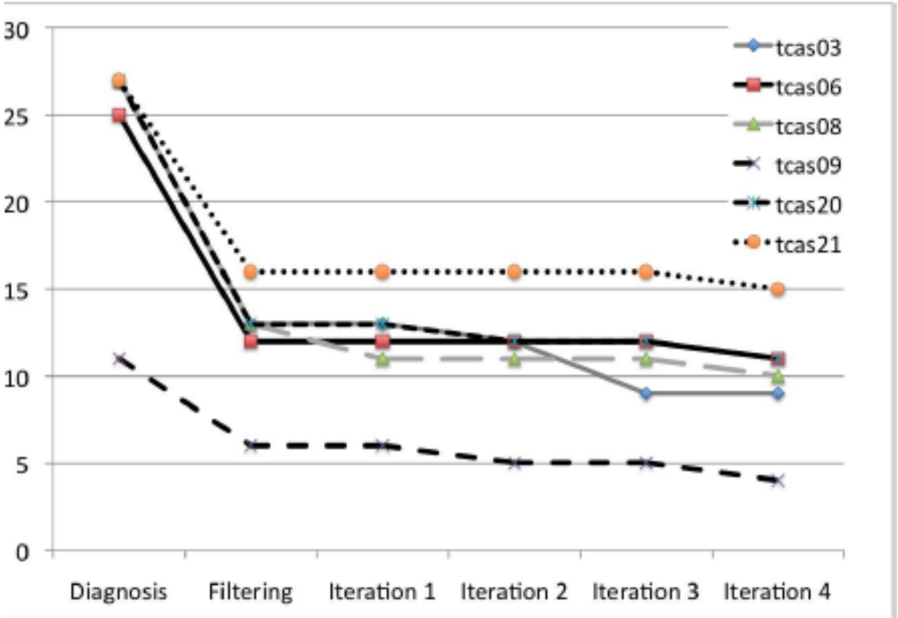


Name	It	Var_{Π}	LOC_{Π}	Inputs	Outputs	LOC_{SSA}	CO	Var_{CO}	Diag	$Diag_{Alt}$	#UI	$Diag_{TC}$
DivATC_V1	2	5	21	2	1	32	33	29	3	2	1	2
DivATC_V2	2	5	21	2	1	32	33	29	5	3	1	1
DivATC_V3	2	5	21	2	1	32	33	29	3	2	1	2
DivATC_V4	2	5	21	2	1	32	33	29	4	4	1/2	3(1)/1
GcdATC_V1	2	6	35	2	1	49	61	46	2	2	1	1
GcdATC_V2	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
Binomial	2	37	189	6	3	481	1329	763	5	3	1	1
BinSearch	2	8	37	4	1	66	102	83	6	3	1	2
Hamming	2	15	77	2	1	176	266	185	9	5	1	3
Data	2	5	40	1	1	45	57	50	7	3	1	1
whileTest	2	16	94	4	1	110	147	103	8	4	1	3

Experimental results



Progr.	$\lvert D \rvert$	R_{orig}	$\lvert D_{filt} \rvert$	R_{filt}	#UI	$\lvert D_{TC} \rvert$	R_{TC}
tcas01	28	77.7	15	88.0	1	14	88.8
tcas02	26	79.2	-	-	-	-	-
tcas03	27	78.4	13	89.6	4	9	92.8
tcas04	25	80.0	14	88.8	1	14	88.8
tcas05	25	80.0	14	88.8	1	13	89.6
tcas06	25	80.0	12	90.4	4	11	91.2
tcas07	9	92.8	5	96.0	1	5	96.0
tcas08	27	78.4	13	89.6	4	10	92.0
tcas09	11	91.2	6	95.2	4	4	96.8
tcas10	29	76.8	18	85.6	1	17	86.4
tcas11	23	81.6	13	89.6	1	13	89.6
tcas12	23	81.6	12	90.4	1	12	90.4
tcas13	27	78.4	5	96.0	0	5	96.0
tcas14	6	95.2	2	98.3	0	2	98.3
tcas15	24	80.8	12	90.4	1	11	91.2
tcas16	26	79.2	6	95.2	0	6	95.2
tcas17	9	92.8	2	98.3	0	2	98.3
tcas18	9	92.8	2	98.3	0	2	98.3
tcas19	9	92.8	2	98.3	0	2	98.3
tcas20	27	78.4	13	89.6	4	11	91.2
tcas21	27	78.4	16	87.2	4	15	88.0
tcas22	8	93.6	4	96.8	1	4	96.8
tcas23	9	92.8	5	96.0	1	5	96.0
tcas24	24	80.8	-	-	-	-	
tcas25	9	92.8	5	96.0	1	4	96.8
tcas26	25	80.0	14	88.8	1	14	88.8
tcas27	25	80.0	12	90.4	1	12	90.4
tcas28	14	88.8	7	94.4	1	7	94.4
tcas29	10	92.0	5	96.0	1	5	96.0
tcas30	13	92.0	9	92.8	1	9	92.8
tcas31	24	80.0	-	-	-	-	-
tcas32	23	81.6	13	89.6	1	13	89.6
tcas33	9	92.8	3	97.	0	3	97.5
tcas34	22	82.4	12	90.4	1	12	90.4
tcas35	14	88.8	8	93.6	1	8	93.6
tcas36	2	98.4	2	98.4	1	2	98.4
tcas37	9	92.8	-	-	-	-	-
tcas38	1	99.2	-	-	-	-	-
tcas39	9	92.8	-	-	-	-	-
tcas40	8	93.6	-	-	-	-	-
tcas41	27	78.4	17	86.4	1	17	86.4
Average		**85.8**		**92.7**			**93.1**



Remarks

- Computing distinguishing test cases from constraints is possible
- Impact for debugging
- Allows extending test suites

- **But**: Require mutants for each fault candidate computed using model-based debugging

SPECIFICATION KNOWLEDGE

Specification knoweldge

- Pre and post conditions
- Invariants
  - Loop invariants
  - Class invariants

- Can be used for improving debugging of loops, recursive functions, and function calls

Loop invariants & more

- Given the following program:

```
int power(int a, int exp)
PRE: { a ≥ 0, exp ≥ 0 }
int e = exp;
int res = 1;
while (e > 0) {
      INV: { res = $a^{exp-e}$ }
      res = res * a;
      e = e - 1; }
return res;
POST: { res = $a^{exp}$ }
```

Loop invariants & more (cont)

```
int power_SSA(int a, int exp) {
a >= 0 && exp >= 0;
1. int e_0 = exp;
2. int res_0 = 1;
3. bool cond_0 = (e_0 > 0);
4. int res_1 = res_0 * a;
5. int e_1 = e_0 - 1;
res_1 == a^(exp-e_1);
6. bool cond_1 = cond_0 ∧ (e_1 > 0);
7. int res_2 = res_1 * a;
8. int e_2 = e_1 - 1;
res_2 == a^(exp-e_2);
...
11. int res_4 = Φ(res_3, res_0, cond_0);
...
res_4 == a^exp;
```

Intermediate observation

- Pre and post conditions as well as invariants can be easily integrated in the SSA representation (and therefore also the constraint representation).

Handling large programs

- But how to debug larger programs using constraints?
  - Summarizing all constraints -> large constraint representation to be solved!
  - Use pre and post conditions instead of the constraints of methods -> modularization possible!

Modularized debugging

- **Idee**: replace every function call where the pre and post conditions are available with pre && post

```
foo () {
    int a = 2;
    int exp = 4;
    int result = power(a,exp);
}
```

```
foo () {
    int a = 2;
    int exp = 4;
    a >= 0 && exp >= 0 && result = a^exp;
}
```

Another observation

- When considering pre and post conditions the problem of debugging even for larger programs is feasible!

Summary specification knowledge

- Specification knowledge is important for debugging
  - Reduce the model size used for debugging
  - Gain information that helps to remove fault candidates
- Integration of specification knowledge into the constraint representation is straight forward

Summary

- Constraints for testing and debugging
- Able to remove up to 93 % of the source code for imperative programs on average using filtering and distinguishing test cases.
- Able to remove 99 % of statements for combinatorial circuits/programs and 97 % for sequential circuits/programs
- Better results than other approaches but computationally more demanding!

Conclusions

- Model-based debugging ensures optimal results
- For small programs (methods,..)
- Allows combining testing and fault localization under one general framework
- There is no silver bullet!

Open challenges

- Combining different debugging approaches
  - Spectrum-based
  - Mutation-based
  - Dependency-based
  - Model-based
  - …
- Improving performance
- Handling OO constructs still open research question

Some papers…

- Bernhard Peischl and Franz Wotawa. *Automated Source Level Error Localization in Hardware Designs*. *IEEE Design & Test of Computers, Jan-Feb, 2006*
- Wotawa, F.; Nica, M.; Aichernig, B.: *Generating Distinguishing Tests using the MINION Constraint Solver*. - in: Proc. of the 2nd Workshop on Constraints in Software Testing, Verification and Analysis (CSTVA'10), 2010.
- Franz Wotawa. *Fault localization based on dynamic slicing and hitting-set computation*. In Proc. 10th International Conference on Quality Software (QSIC), China, 2010.
- M. Nica, S. Nica, and F. Wotawa. *Does testing help to reduce the number of potentially faulty statement in debugging?* In Proc. Testing: Academic & Industrial Conference Practice and Research Techniques (TAIC-PART), 2010.

…and some more…

- Cristinel Mateis, Markus Stumptner, Dominik Wieland, and Franz Wotawa, Model-Based Debugging of Java Programs, Proc. Intl. Workshop on Automated and Algorithmic Debugging (AADEBUG), Munich, Germany, 2000.
- Wolfgang Mayer, Markus Stumptner, Dominik Wieland, and Franz Wotawa, Can AI help to improve debugging substantially? Debugging experiences with value-based models, Proc. European Conference on Artificial Intelligence (ECAI), Lyon, France, 2002.
- Wolfgang Mayer. *Static and Hybrid Analysis in Model-based Debugging*. PhD thesis, School of Computer and Information Science, University of South Australia, Adelaide, Australia, July 2007.
- Wolfgang Mayer and Markus Stumptner. Evaluating Models for Model-Based Debugging. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 128–137, L'Aquila, Italy, September 2008. IEEE Computer Society Press.

END OF PART 2

QUESTIONS?