# Automated Structural Testing with PathCrawler

## Tutorial for QSIC 2012

**Nicky.WILLIAMS@cea.fr, Nikolai.KOSMATOV@cea.fr,**

**CEA, LIST, Software Safety Lab**

**Saclay (Paris), France**

**Xi’an, 27th August, 2012**

# Outline

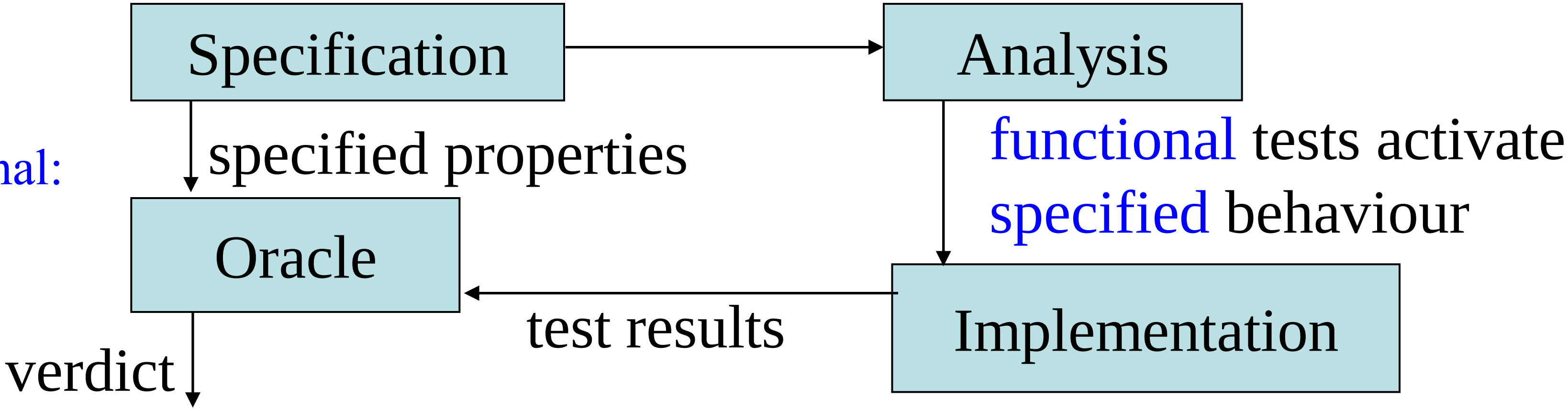
1. Structural testing: a brief introduction
2. PathCrawler tool
3. Test parameters
4. Oracle and program debugging
5. Structural test for other properties/purposes
6. Strengths and limits of structural testing
7. Bypassing the limits

# Outline

1. **Structural testing: a brief introduction**
2. PathCrawler tool
3. Test parameters
4. Oracle and program debugging
5. Structural test for other properties/purposes
6. Strengths and limits of structural testing
7. Bypassing the limits

# Structural vs. functional testing

Functional:



Specification

specified properties

Oracle

verdict

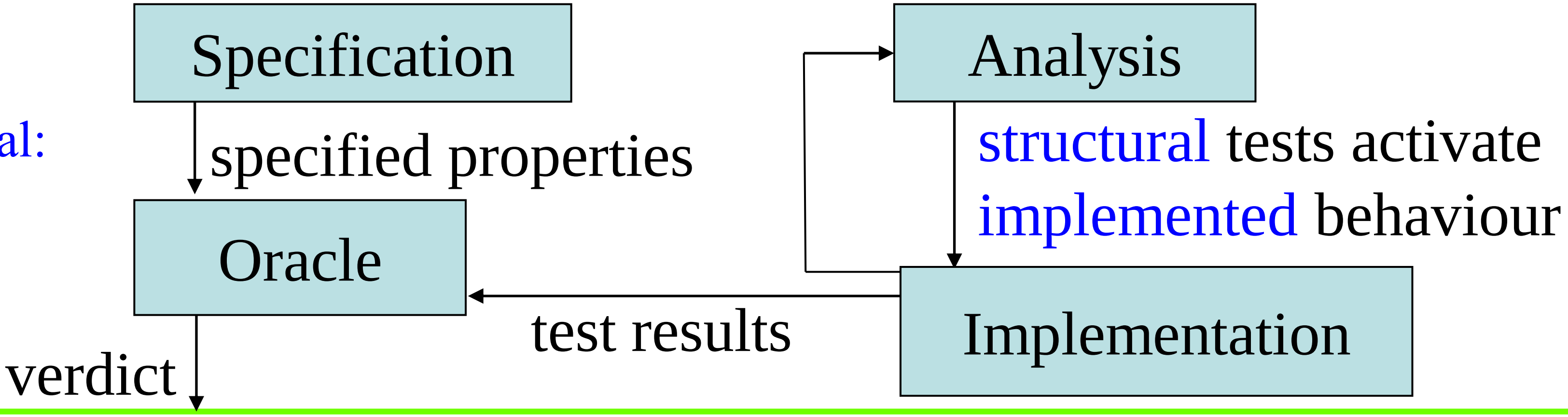
Analysis

functional tests activate specified behaviour

Implementation

test results

Structural:



Specification

specified properties

Oracle

verdict

Analysis

structural tests activate implemented behaviour

Implementation

test results

# Unit structural testing is useful

Manually created functional test cases do not cover all the code

- Certain « functional » test cases can be missed
- Certain parts of code can depend on implementation choices and cannot be properly covered by the specification

Evaluation of structural coverage

Adding test cases to complete structural coverage

# Unit structural testing can be mandatory

**Development, evaluation and certification standards**

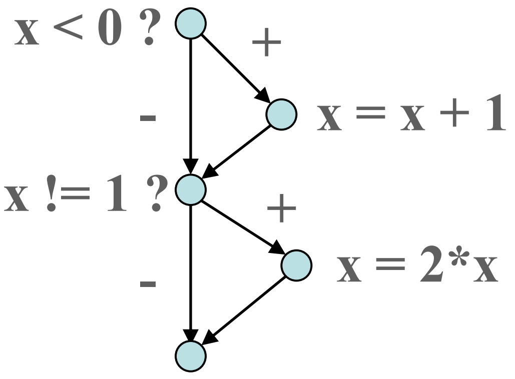
- **Common Criteria for IT Security Evaluation**
- **DO-178B (avionics)**
- **ECCS-E-ST-40C (space)**
- **IEC/EN 61508 (*Electronic Safety-related Systems*) & derived standards:**
  - ISO 26262 (automotive)
  - IEC/EN 50128 (rail)
  - IEC/EN 60601 (medical)
  - EC/EN 61513 (nuclear)
  - IEC/EN 60880 (nuclear safety-critical)
  - IEC/EN 61511 (process e.g. petrochemical, pharmaceutical)

# CFG and code coverage by example

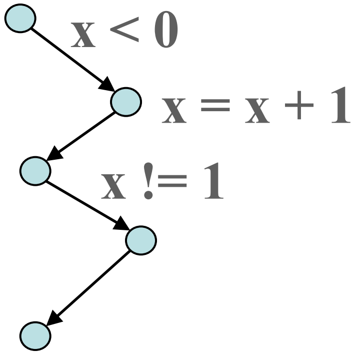
C code

```
int f(int x){
 if(x < 0)
  x = x + 1;
 if(x != 1)
  x = 2*x;
 return x; }
```

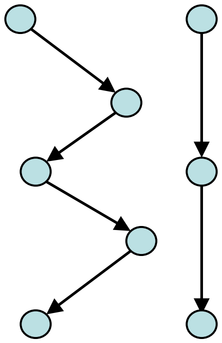
control-flow graph (CFG)



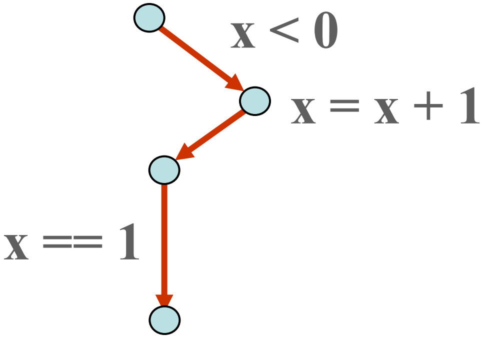
statement coverage



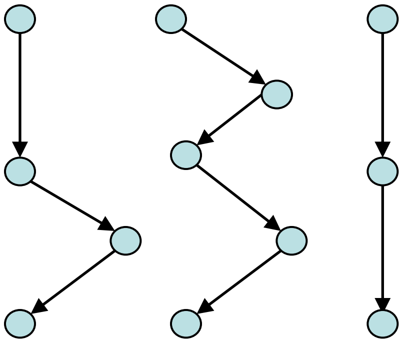
branch coverage



infeasible path



all-path coverage

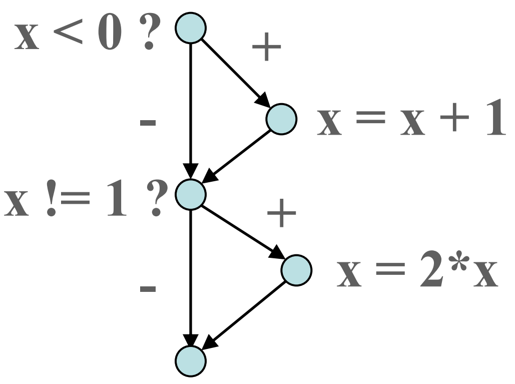


# Path predicate (path condition) by example

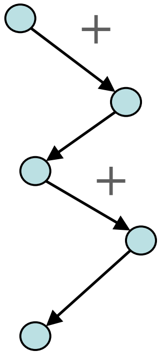
C code

```
int f(int x){
 if(x < 0)
  x = x + 1;
 if(x != 1)
  x = 2*x;
 return x; }
```

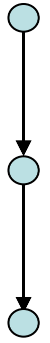
control-flow graph (CFG)



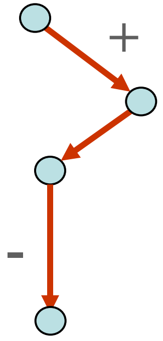
path predicate



$x_0 < 0 \ \wedge x_0 + 1 \neq 1$



$x_0 >= 0 \wedge x_0 = 1$



$x_0 < 0 \wedge x_0 + 1 = 1$

infeasible path

⇔

unsatisfiable path predicate

# Automated structural testing… Why?

Achieving desired test coverage manually is costly

Must be done again after any code modification

Infeasibility of a test objective can be difficult to show manually

Automated structural testing tools can be used

- to reach the uncovered objectives,
- to determine that some of them are unreachable,
- with a low cost overhead

# Outline

1. Structural testing: a brief introduction
2. PathCrawler tool
3. Test parameters
4. Oracle and program debugging
5. Strengths and limits of structural testing
6. Structural test for other properties/purposes
7. Bypassing the limits

# PathCrawler tool

- Concolic testing tool for C developed at CEA LIST
- Input: a complete compilable source code
- Automatically creates test cases to cover program paths (explored in depth-first search)
- Uses code instrumentation, concrete and symbolic execution, constraint solving
- Exact semantics: don’t rely on concrete values to approximate the path predicate
- Similar to PEX, DART/CUTE, KLEE, SAGE etc.

# PathCrawler explores the tree of feasible paths

*depth-first search with non-deterministic choice of suffix*

**test1: x = -5**

$$x_0 < 0 \xrightarrow[x_1 = x_0 + 1]{+2} x_1 \neq 1 \xrightarrow[x_2 = 2x_1]{+4}$$

```
int f(int x){
 if(x < 0)
  x = x + 1;
 if(x != 1)
  x = 2*x;
 return x; }
```

# PathCrawler explores the tree of feasible paths

*depth-first search with non-deterministic choice of suffix*

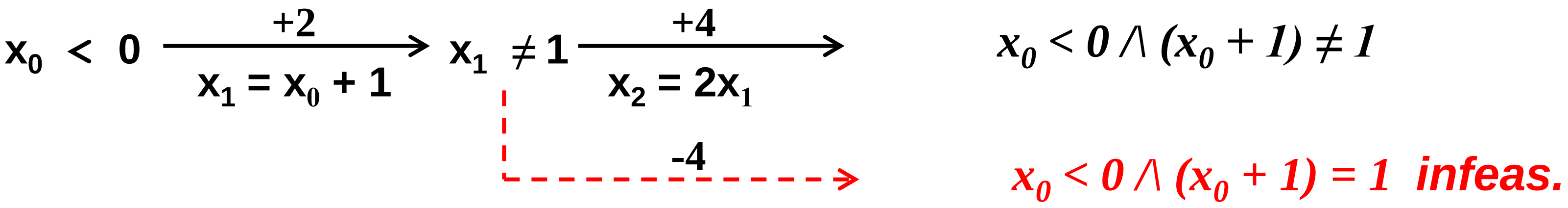
**test1:** $x = -5$ $\quad x_0 < 0 \xrightarrow[x_1 = x_0 + 1]{+2} x_1 \neq 1 \xrightarrow[x_2 = 2x_1]{+4} \quad x_0 < 0 \wedge (x_0 + 1) \neq 1$

```
int f(int x){
 if(x < 0)
  x = x + 1;
 if(x != 1)
  x = 2*x;
 return x; }
```

# PathCrawler explores the tree of feasible paths

*depth-first search with non-deterministic choice of suffix*

test1: $x = -5$



$x_0 < 0 \xrightarrow[x_1 = x_0 + 1]{+2} x_1 \neq 1 \xrightarrow[x_2 = 2x_1]{+4}$

$x_0 < 0 \wedge (x_0 + 1) \neq 1$

$\xrightarrow{-4}$

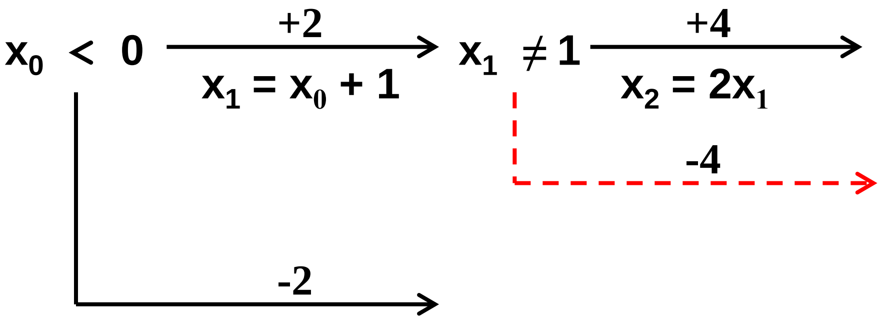
$x_0 < 0 \wedge (x_0 + 1) = 1$ ***infeas.***

```
1 int f(int x){
2  if(x < 0)
3   x = x + 1;
4  if(x != 1)
5   x = 2*x;
6  return x; }
```

# PathCrawler explores the tree of feasible paths

*depth-first search with non-deterministic choice of suffix*

**test1: x = -5**



$x_0 < 0 \wedge (x_0 + 1) \neq 1$

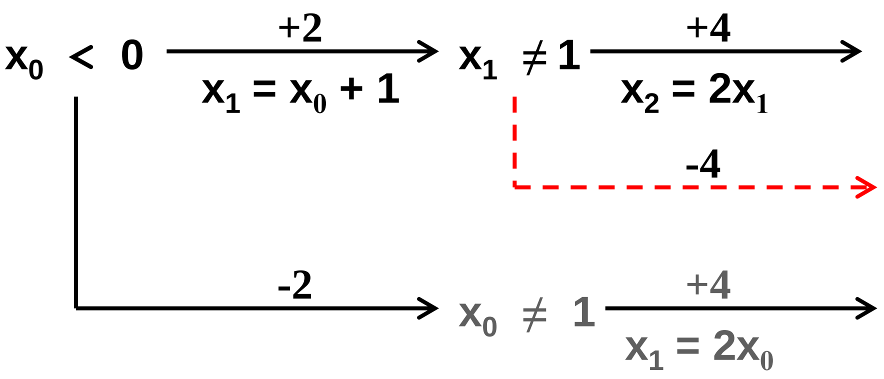
$x_0 < 0 \wedge (x_0 + 1) = 1$ ***infeas.***

$x_0 \geq 0$

```
int f(int x){
 if(x < 0)
  x = x + 1;
 if(x != 1)
  x = 2*x;
 return x; }
```

# PathCrawler explores the tree of feasible paths

*depth-first search with non-deterministic choice of suffix*



**test1: x = -5**

$x_0 < 0 \xrightarrow[x_1 = x_0 + 1]{+2} x_1 \neq 1 \xrightarrow[x_2 = 2x_1]{+4}$   $x_0 < 0 \wedge (x_0 + 1) \neq 1$

$x_1 \neq 1 \xrightarrow{-4}$   $x_0 < 0 \wedge (x_0 + 1) = 1$ ***infeas.***

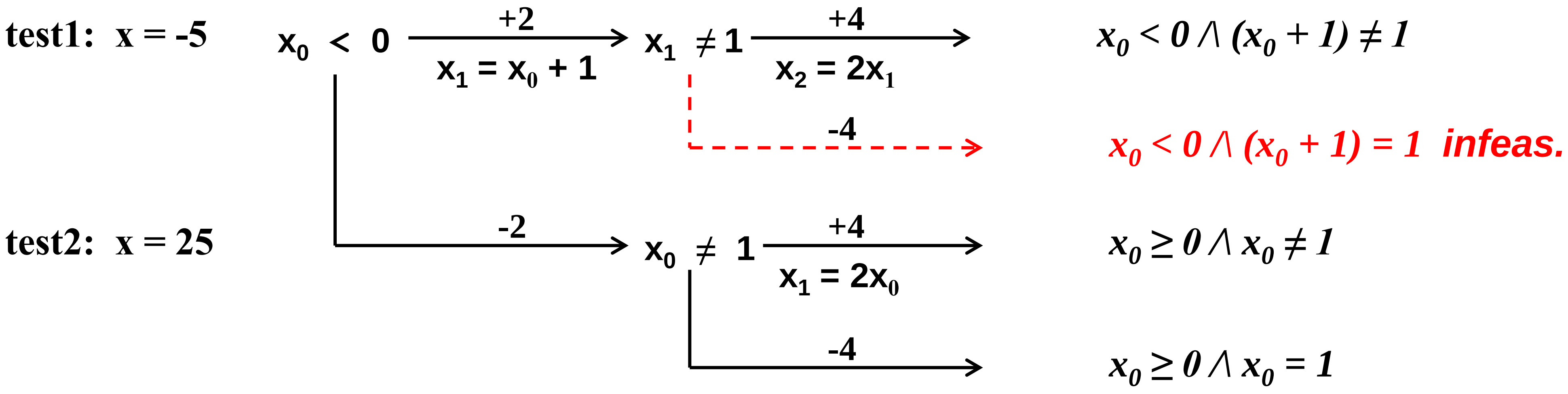
**test2: x = 25**

$x_0 < 0 \xrightarrow{-2} x_0 \neq 1 \xrightarrow[x_1 = 2x_0]{+4}$   $x_0 \geq 0 \wedge x_0 \neq 1$

```
1 int f(int x){
2  if(x < 0)
3   x = x + 1;
4  if(x != 1)
5   x = 2*x;
6  return x; }
```

# PathCrawler explores the tree of feasible paths

*depth-first search with non-deterministic choice of suffix*



**test1:** x = -5

**test2:** x = 25

$x_0 < 0$ $\xrightarrow[x_1 = x_0 + 1]{+2}$ $x_1 \neq 1$ $\xrightarrow[x_2 = 2x_1]{+4}$ $x_0 < 0 \wedge (x_0 + 1) \neq 1$

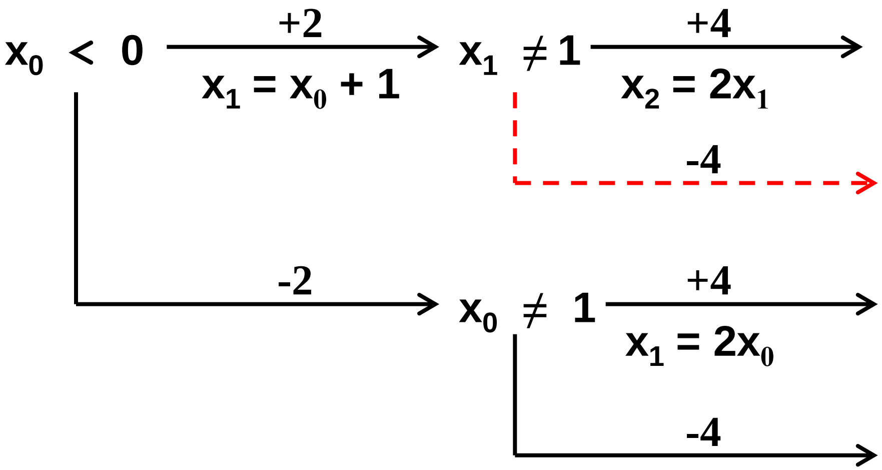
$x_1 \neq 1$ $\xrightarrow{-4}$ $x_0 < 0 \wedge (x_0 + 1) = 1$ ***infeas.***

$x_0 < 0$ $\xrightarrow{-2}$ $x_0 \neq 1$ $\xrightarrow[x_1 = 2x_0]{+4}$ $x_0 \geq 0 \wedge x_0 \neq 1$

$x_0 \neq 1$ $\xrightarrow{-4}$ $x_0 \geq 0 \wedge x_0 = 1$

# PathCrawler explores the tree of feasible paths

*depth-first search with non-deterministic choice of suffix*



test1: $x = -5$ — $x_0 < 0 \wedge (x_0 + 1) \neq 1$

$x_0 < 0 \wedge (x_0 + 1) = 1$ ***infeas.***

test2: $x = 25$ — $x_0 \geq 0 \wedge x_0 \neq 1$

test3: $x = 1$ — $x_0 \geq 0 \wedge x_0 = 1$

# pathcrawler-online.com

## Freely available test-case generation web service

- **Instead of open-source or demonstration version**
- **No porting, no installation, universal user interface**
- **Well adapted to**
  - Teaching
  - Use by project partners
  - Evaluation, understanding of Precondition and Oracle
- **Limited version (contact us for unlimited access)**

## During the tutorial

- **Browser: no cache recommended**
- **Do not start several test generation sessions in parallel**

# Example 1. Robust implementation of Tritype

Simple program `Tritype`

- inputs: three floating-point numbers `i`, `j`, `k`
- returns the type of the triangle with sides `i`, `j`, `k`:
  `3` (not a triangle), `2` (equilateral), `1` (isosceles), `0` (other)

**Robust : validity of inputs is tested ("not a triangle")**

**⇒ Any test case can be interesting and useful**

**"Test with predefined params" on pathcrawler-online.com**

**Observe the number of test cases. Check the results.**

# PathCrawler outputs

- A suite of test cases including
  - Input values (check these for Example 1)
  - Concrete outputs (check these for Example 1)
  - Symbolic outputs (better illustrated by Example 5)
  - Path predicate (better illustrated by Example 5)
  - Test driver
  - Oracle verdict (better illustrated by Example 10)
- Explored program paths with
  - their status (covered, infeasible, assume violated …)
  - path predicate (only for covered paths in online version)

# Outline

1. Structural testing: a brief introduction
2. PathCrawler tool
3. **Test parameters**
4. Oracle and program debugging
5. Strengths and limits of structural testing
6. Structural test for other properties/purposes
7. Bypassing the limits

# Example 2. Non robust implementation of Tritype

**No validity check lines 10-13, no “not a triangle” answer**

**⇒ Are the test cases still interesting?**

**“Test with predefined params” on pathcrawler-online.com**

**Observe the number of test cases. Check the results.**

**Where is the problem?**

**Do we really want such input values in this case?**

# Exercise 3. Customize test parameters for Tritype

**How to generate appropriate test cases only ?**

**⇒ define a precondition!**

**Exercise. Start from Example 2. “Customize test parameters”**

**- Restrict the domains of inputs `i`, `j`, `k` to non negative values:**

*[ 0 .. 1.7976931348623157e+308 ]*

**- Add 3 unquantified preconditions:**

$$i + j > k$$

$$j + k > i$$

$$i + k > j$$

**- Confirm parameters and check the results.**

# Example 4. C Precondition for Tritype

**Another way to define a precondition**

**⇒ in a C function**

`Tritype_precond` **returns `1` iff the precondition is verified**

**“Customize test parameters” on pathcrawler-online.com to check that Pathcrawler has activated the C precondition.**

**Confirm & observe the number of test cases & results.**

PathCrawler

# Test parameters

- Define admissible inputs (precondition)
  - Domains of input variables
  - Relations between variables…
- Wrong test parameters may
  - Indicate inexistent bugs (the bug is in the input)
  - Provoke runtime errors

## Example 5. Merge with default parameters

Merge of two sorted arrays `t1`, `t2` into a sorted array `t3`

- inputs: arrays `t1[3]`, `t2[3]`, `t3[6]` of fixed size

**“Test with predefined params” on pathcrawler-online.com**

**Check the concrete outputs.**

**What is wrong with the concrete outputs?**

**This example also illustrates well the information on array inputs, symbolic outputs and path predicate included in a test-case**

# Exercise 6. Quantified precondition for Merge

**If the input arrays `t1` and `t2` are not ordered, `Merge` does not work!**

**Exercise. Start from Example 5. “Customize test parameters”**

- **Add two quantified preconditions (*INDEX* is a reserved word):**
  *for all INDEX*
  *such that INDEX < 2*
  *we have t1[ INDEX ]<= t1[ INDEX+1 ]*
  *for all INDEX*
  *such that INDEX < 2*
  *we have t2[ INDEX ]<= t2[ INDEX+1 ]*
- **Confirm parameters and check the results.**

**Are the input arrays `t1` and `t2` sorted now? Is `t3` sorted?**

## Example 7. Merge with pointer inputs

**Merge of two sorted arrays `t1`, `t2` into a sorted array `t3`**

- **inputs: arrays `t1[ ]`, `t2[ ]`, `t3[ ]` of variable size, `l1` the size of `t1`, `l2` the size of `t2`, `l1+l2` the size of `t3`**
- **precondition** *t1, t2 ordered arrays* **predefined**
- **reduced domains of elements** [-100,100] **predefined**

**"Test with predefined params" on pathcrawler-online.com**

**Check the results.**

**Why are there errors?**

# Exercise 8. Input arrays (pointers) size

`t1`, `t2`, `t3` should contain resp. `l1`, `l2`, `l1+l2` allocated elements.
Wrong input array size => Runtime errors while executing tests!

**Exercise. Start from Example 7. "Customize test parameters"**

- **Specify domains for dim(`t1`), dim(`t2`) , dim(`t3`)**

  *0 <= dim(t3) <= 6*
  *0 <= dim(t2) <= 3*
  *0 <= dim(t1) <= 3*

- **Add three unquantified preconditions:**

  *dim(t1) == l1*
  *dim(t2) == l1*
  *dim(t3) == l1 + l2*

- **Confirm parameters and check the results.**

**Are there errors? Why? How many test cases are generated?**

# Partial test coverage : k-path criterion

- In presence of loops, all-path criterion may generate too many test cases
- The user may want to limit their number

- **k-path coverage restricts the all-path criterion to paths with at most k consecutive iterations of each loop (k=0,1,2…)**

# Exercise 9. Merge with partial test coverage: k-path

**To reduce the number of test cases, modify test criterion.**

**Exercise. Continue Exercise 8 with the same test parameters you defined. “Customize test parameters”**

- **Set “Path selection strategy” to 2 (for k-path with k=2)**
- **Confirm parameters and check the results.**

**How many test cases are generated now?**

# Outline

1. Structural testing: a brief introduction
2. PathCrawler tool
3. Test parameters
4. **Oracle and program debugging**
5. Strengths and limits of structural testing
6. Structural test for other properties/purposes
7. Bypassing the limits

# Oracle

Role of an oracle:

- examines the inputs and outputs of each test
- decides whether the implementation has given the expected results
- provides a verdict (success, failure)

An oracle can be provided by

- another, or previous implementation
- checking the results without implementing the algorithm

# Exercise 10a. Oracle and debugging

Start from Example 10a, “Customize test parameters” to see an example of an oracle

**Is this oracle complete ?**

# Exercise 10b. Oracle and debugging

Start from Example 10b, “Customize test parameters” to see another example of an oracle

**Is this oracle complete ?**

# Exercise 10c. Oracle and debugging

Start from Example 10a, “Customize test parameters” to see the predefined oracle

**Exercise. Confirm parameters and check the results.**

**Can you find an error in the implementation?**

Hint: The paths of failed test cases have a common part…

# Outline

1. Structural testing: a brief introduction
2. PathCrawler tool
3. Test parameters
4. Oracle and program debugging
5. **Structural test for other properties/purposes**
6. Strengths and limits of structural testing
7. Bypassing the limits

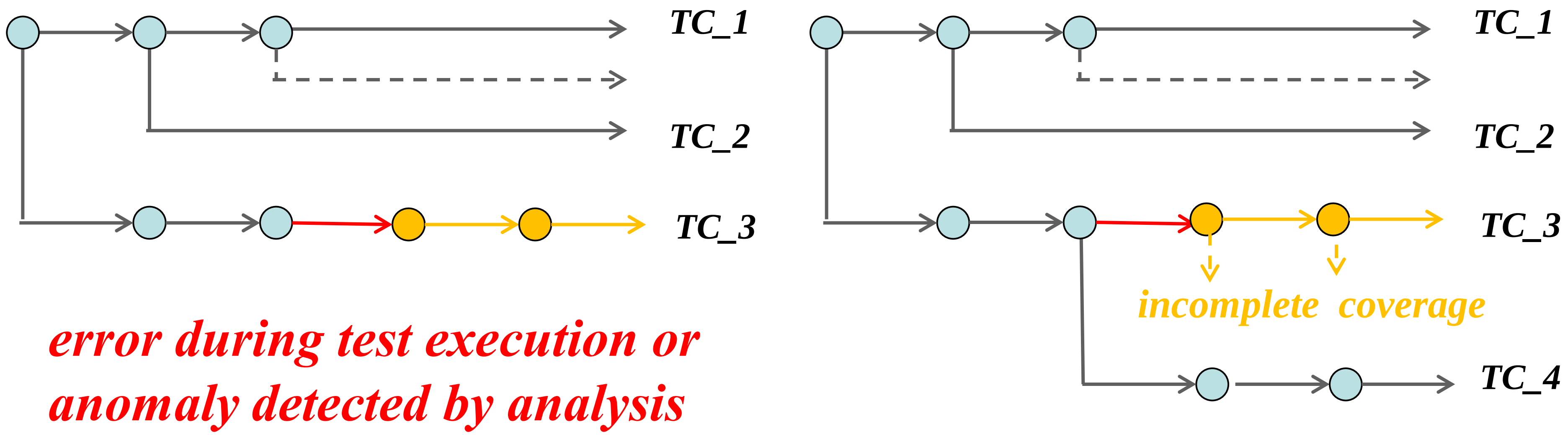
# Structural test for other properties or purposes

PathCrawler explores the implementation and can also be used to check:

- for **runtime errors** during program execution (seen in Ex.7)
- for **anomalies** detected during analysis of the covered paths:
    - uninitialised variables
    - buffer overflow
    - integer overflow
    - …
- whether the implementation performs **unnecessary computation**
- the effective **execution time** of each path (at least for one set of inputs), by running the generated tests on a platform which can measure execution time
- for unreachable or **"dead" code**: check infeasible partial paths.

  If all paths leading to the code are infeasible then the code is unreachable (for the given precondition): is this intentional ?

# Runtime error or anomaly: search space is pruned



# Example Uninit. Uninitialised variable

In this example, the local variables are not always initialised before their value is read. This is a typical “anomaly”: probably a bug but does not cause a run-time error.

**“Test with predefined parameters” and check the results.**

**Are there any errors or warnings? Why?**
**Are all feasible paths covered?**

# Example UC. Unnecessary computation

`Bsearch` is an implementation of dichotomic search for value `x` in sorted array `A`.

**"Customize test parameters" to see the predefined oracle and parameters. Confirm them and check the results.**

**Examine the predicates and input values of the cases where x is present. Is this an efficient implementation?**

# Outline

1. Structural testing: a brief introduction
2. PathCrawler tool
3. Test parameters
4. Oracle and program debugging
5. Structural test for other properties/purposes
6. **Strengths and limits of structural testing**
7. Bypassing the limits

# Dichotomic search: structural vs. other strategies

**Example:** dichotomic search for a value int `x` in a sorted array int `A[10]`.

**Random testing:** Unlikely to construct cases in which x equals one of the elements of A and to detect false negatives (x not detected when present)

**Functional testing:** Constructs

- many cases in which x is present (probably from 1 to 10?) and
- fewer cases in which x is absent (1 or 2 ?)

**Structural testing:** Constructs a case

- for each position in A for which x can be detected and
- for each relation to elements of A for which absence of x is detected.

Structural test. constructs more presence cases than random, more absence cases than functional, rarely constructs cases where x is present by chance.

# Example Chance. Failures by chance?

`Bsearch` is another implementation of dichotomic search for value `x` in sorted array `A`. It contains a bug which can result in false positives (x present but not detected).

**The parameters are the same as in the previous example. Confirm them and check the results.**

**Is the presence or absence of x in A always determined by the path predicate?**

Hint: look at failing cases or those where x is present.

# Example 11. Limitations of structural testing

`Bsearch` is another erroneous implementation of dichotomic search for value `x` in sorted array `A`.

**The parameters are the same as in the previous example. Confirm them and check the results.**

**Are there any failures?**

# Limitations of structural testing

Structural testing is

- **effective** when a bug is **always revealed** by a path,
- **less so** when ***only some of the values*** which activate the path cause the bug to be revealed

**PathCrawler chooses arbitrary values to test each path**

**They may not be the values which will reveal a bug**

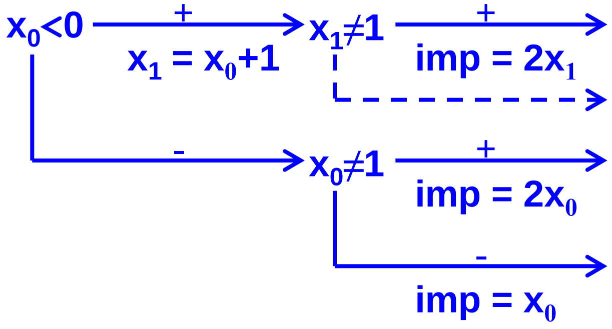
**We can make PathCrawler *go looking for bugs***

**by sub-dividing the paths**

# Outline

1. Structural testing: a brief introduction
2. PathCrawler tool
3. Test parameters
4. Oracle and program debugging
5. Structural test for other properties/purposes
6. Strengths and limits of structural testing
7. **Bypassing the limits**

# Cross-checking conformity with a specification



$x_0<0$

+

$x_1 = x_0+1$

$x_1 \neq 1$

+

imp = $2x_1$

-

$x_0 \neq 1$

+

imp = $2x_0$

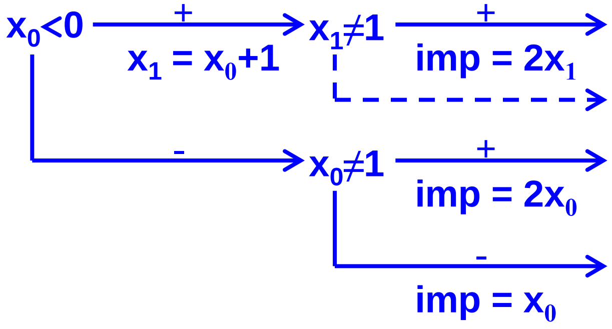
-

imp = $x_0$

implementation

```
int f(int x){
 if(x < 0)
  x = x + 1;
 if(x != 1)
  x = 2*x;
 return x; }
```

# Cross-checking conformity with a specification



$x_0<0$ + $x_1 = x_0+1$

$x_1 \neq 1$ + $imp = 2x_1$

-

$x_0 \neq 1$ + $imp = 2x_0$

- $imp = x_0$

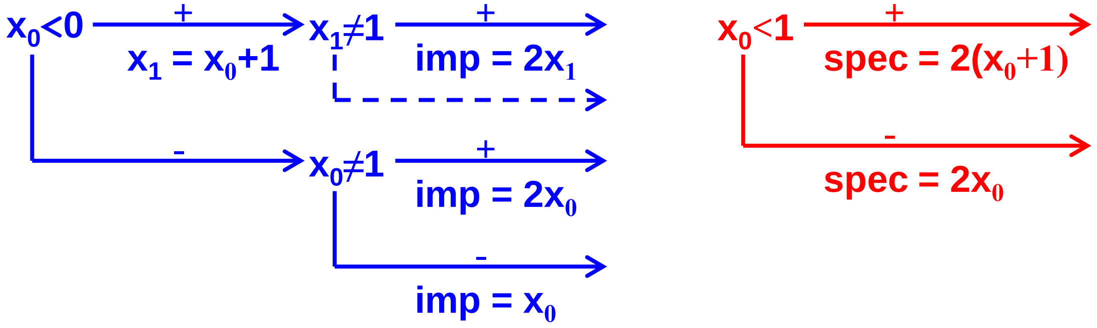
## implementation

```
int f(int x){
 if(x < 0)
  x = x + 1;
 if(x != 1)
  x = 2*x;
 return x; }
```

## specification

*If x is less than 1 then*
*the result should be 2(x + 1)*
*else the result should be 2x*

# Cross-checking conformity with a specification



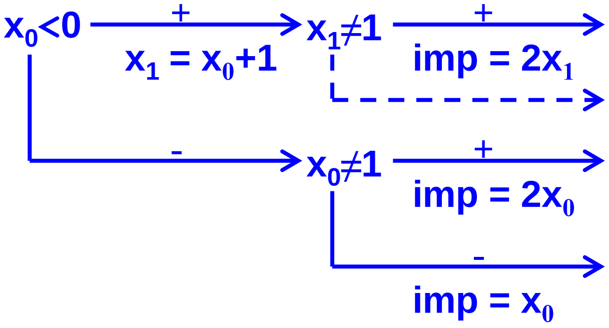
implementation

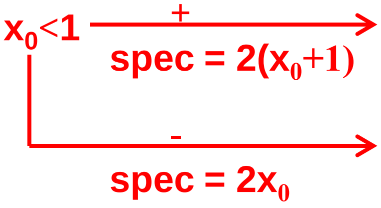
```
int f(int x){
 if(x < 0)
  x = x + 1;
 if(x != 1)
  x = 2*x;
 return x; }
```

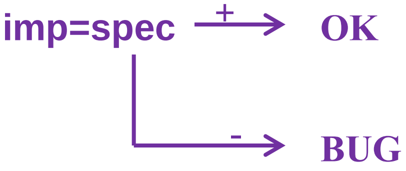
specification

```
int spec_f(int x){
 if(x < 1)
  x = 2*(x + 1);
 else
  x = 2*x;
 return x; }
```

# Cross-checking conformity with a specification







## implementation

```
int f(int x){
 if(x < 0)
  x = x + 1;
 if(x != 1)
  x = 2*x;
 return x; }
```

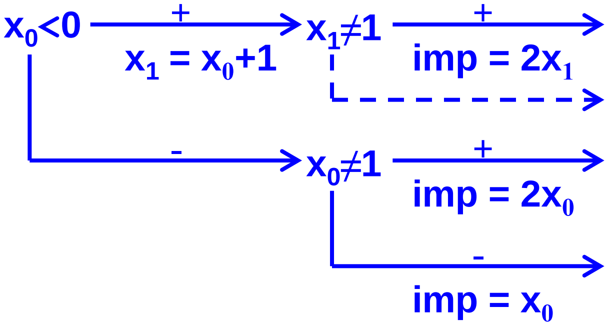
## specification

```
int spec_f(int x){
 if(x < 1)
  x = 2*(x + 1);
 else
  x = 2*x;
 return x; }
```

## comparison

```
int cross_f(int x){
 int imp = f(x);
 int spec=spec_f(x);
 if(imp!=spec)
   return 0;
 else return 1; }
```

# Cross-checking conformity with a specification



$x_0<0$

+

$x_1 = x_0+1$

$x_1 \neq 1$

+

$imp = 2x_1$

-

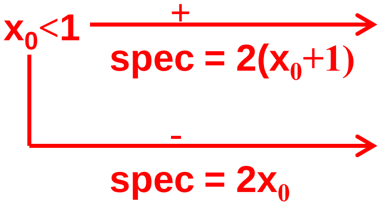
$x_0 \neq 1$

+

$imp = 2x_0$

-

$imp = x_0$



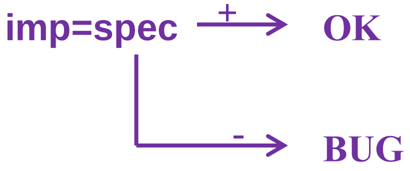
$x_0<1$

+

$spec = 2(x_0+1)$

-

$spec = 2x_0$



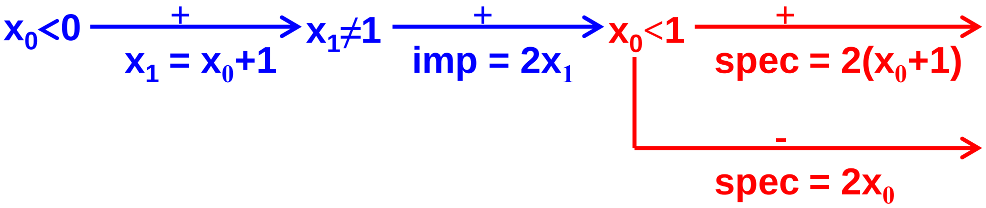
imp=spec

+

OK

-

BUG



$x_0<0$

+

$x_1 = x_0+1$

$x_1 \neq 1$

+

$imp = 2x_1$

$x_0<1$

+

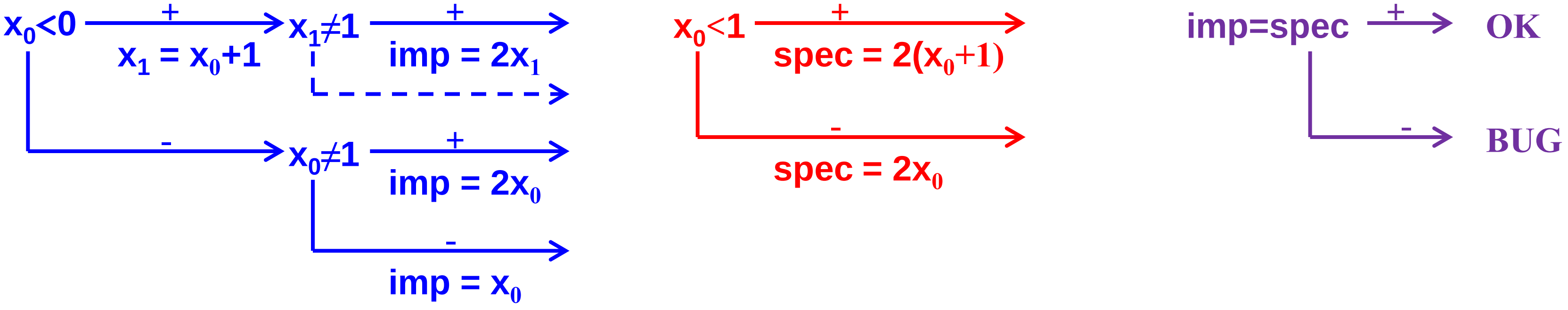
$spec = 2(x_0+1)$

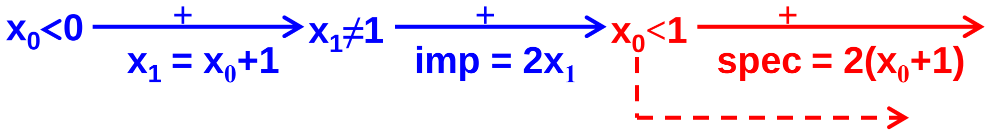
-

$spec = 2x_0$

```
int cross_f(int x){
 int imp = f(x);
 int spec=spec_f(x);
 if(imp!=spec)
   return 0;
 else return 1; }
```

# Cross-checking conformity with a specification





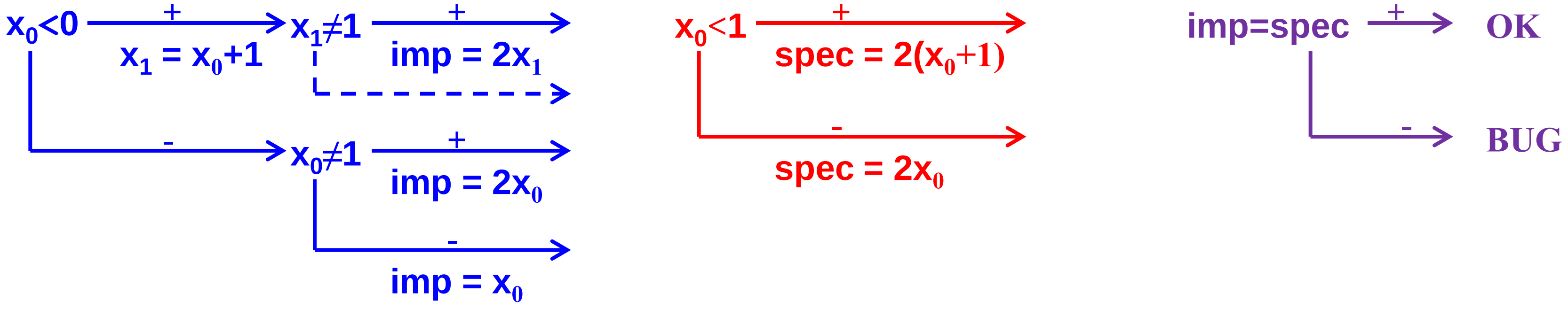
$$x_0 < 0 \wedge (x_0+ 1) \neq 1 \wedge x_0 < 1 \rightarrow x_0 < 0$$

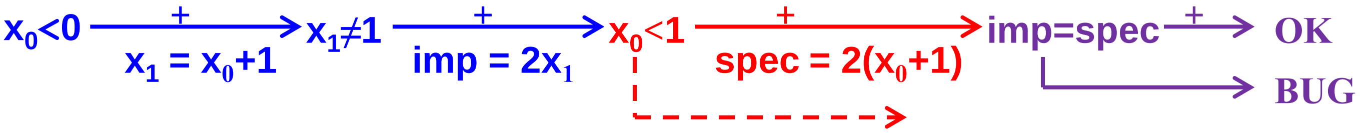
$$x_0 < 0 \wedge (x_0+ 1) \neq 1 \wedge x_0 \geq 1$$

```
int cross_f(int x){
 int imp = f(x);
 int spec=spec_f(x);
 if(imp!=spec)
   return 0;
 else return 1; }
```

PathCrawler

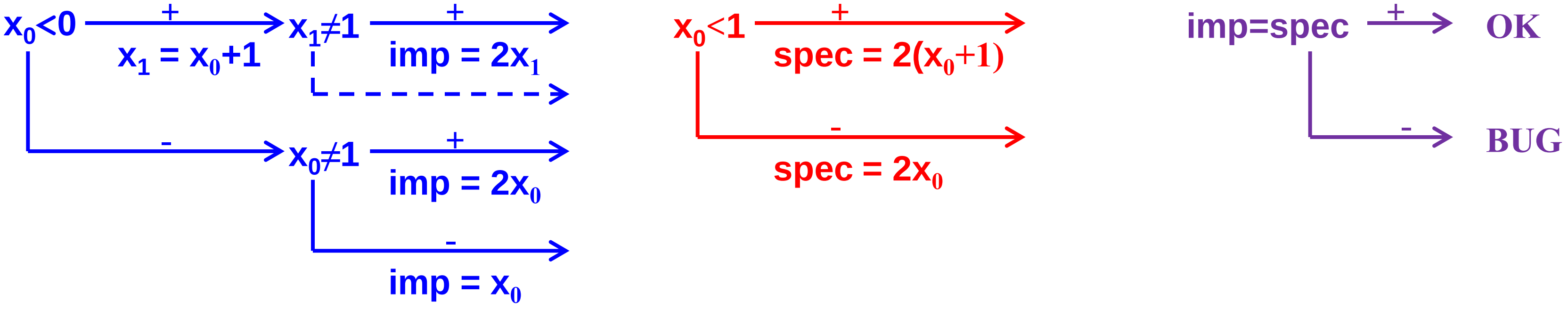
# Cross-checking conformity with a specification





```
int cross_f(int x){
 int imp = f(x);
 int spec=spec_f(x);
 if(imp!=spec)
   return 0;
 else return 1; }
```

# Cross-checking conformity with a specification



$x_0<0$ + $x_1 = x_0+1$ $x_1 \neq 1$ + $imp = 2x_1$

- $x_0 \neq 1$ + $imp = 2x_0$

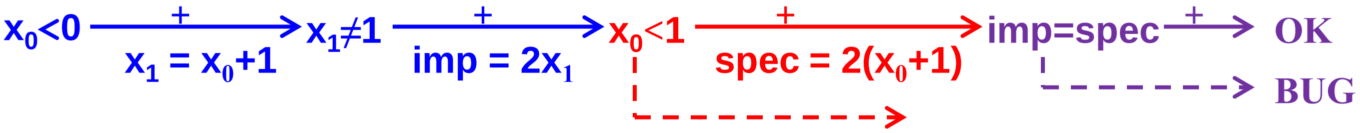
- $imp = x_0$

$x_0<1$ + $spec = 2(x_0+1)$

- $spec = 2x_0$

$imp=spec$ + OK

- BUG



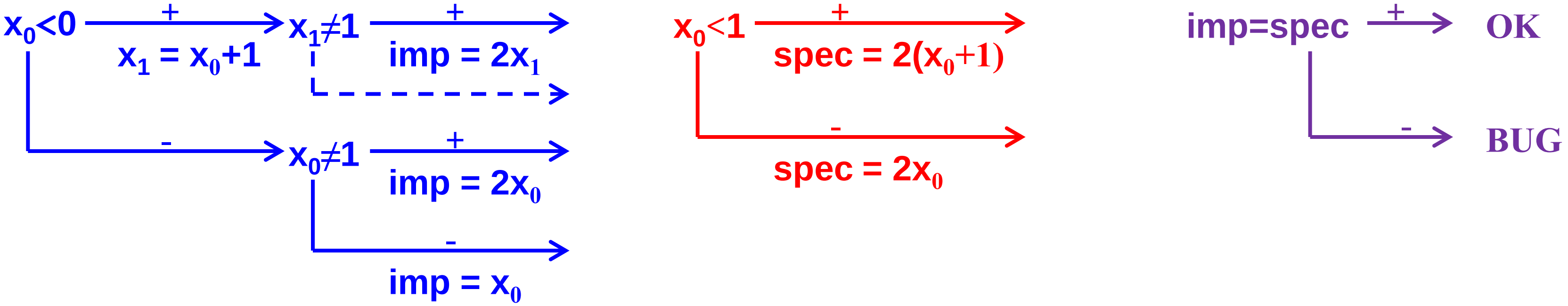
$x_0<0$ + $x_1 = x_0+1$ $x_1 \neq 1$ + $imp = 2x_1$ $x_0<1$ + $spec = 2(x_0+1)$ $imp=spec$ + OK

BUG

$x_0 < 0 \wedge 2(x_0+1) = 2(x_0+1)$

$x_0 < 0 \wedge 2(x_0+1) \neq 2(x_0+1)$

# Cross-checking conformity with a specification



$x_0<0$ $\xrightarrow{+}$ $x_1 = x_0+1$ $x_1 \neq 1$ $\xrightarrow{+}$ imp = $2x_1$

$x_0<0$ $\xrightarrow{-}$ $x_0 \neq 1$ $\xrightarrow{+}$ imp = $2x_0$

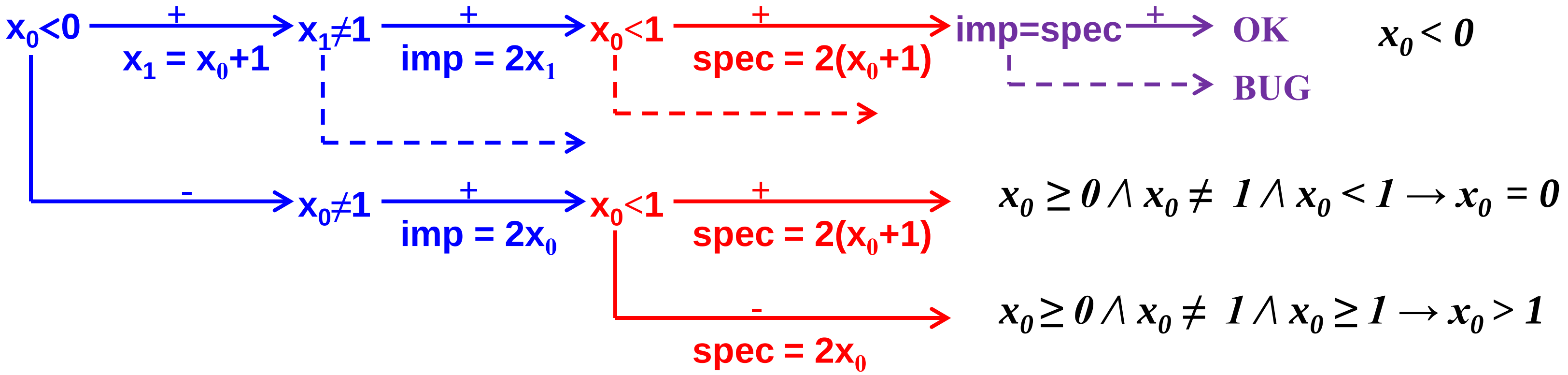
$x_0 \neq 1$ $\xrightarrow{-}$ imp = $x_0$

$x_0<1$ $\xrightarrow{+}$ spec = $2(x_0+1)$

$x_0<1$ $\xrightarrow{-}$ spec = $2x_0$

imp=spec $\xrightarrow{+}$ OK

imp=spec $\xrightarrow{-}$ BUG



$x_0<0$ $\xrightarrow{+}$ $x_1 = x_0+1$ $x_1 \neq 1$ $\xrightarrow{+}$ imp = $2x_1$ $x_0<1$ $\xrightarrow{+}$ spec = $2(x_0+1)$ imp=spec $\xrightarrow{+}$ OK

BUG

$x_0 < 0$

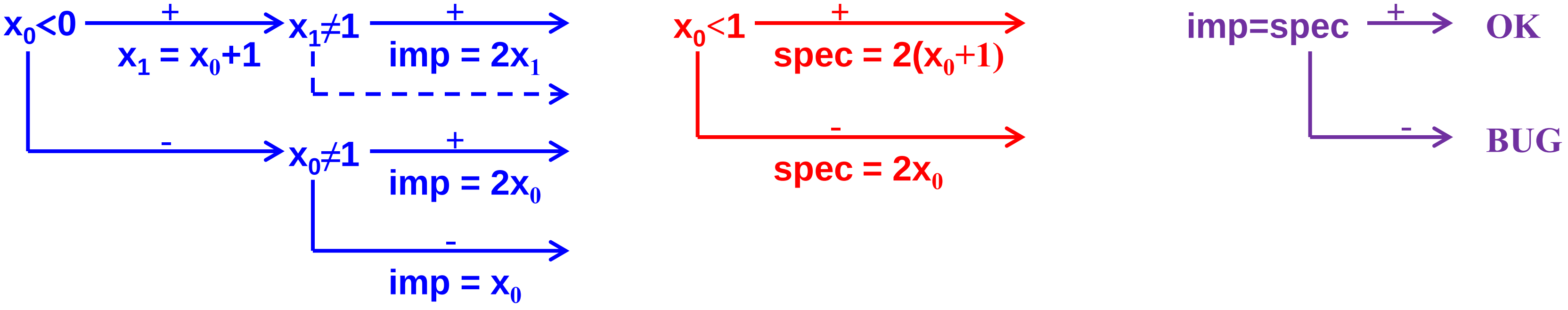
$x_0<0$ $\xrightarrow{-}$ $x_0 \neq 1$ $\xrightarrow{+}$ imp = $2x_0$ $x_0<1$ $\xrightarrow{+}$ spec = $2(x_0+1)$

$x_0 \geq 0 \wedge x_0 \neq 1 \wedge x_0 < 1 \rightarrow x_0 = 0$

$x_0<1$ $\xrightarrow{-}$ spec = $2x_0$

$x_0 \geq 0 \wedge x_0 \neq 1 \wedge x_0 \geq 1 \rightarrow x_0 > 1$

# Cross-checking conformity with a specification



$x_0<0$ $\xrightarrow{+}$ $x_1 = x_0+1$ $\rightarrow$ $x_1\neq 1$ $\xrightarrow{+}$ imp = $2x_1$

$x_0<0$ $\xrightarrow{-}$ $x_0\neq 1$ $\xrightarrow{+}$ imp = $2x_0$

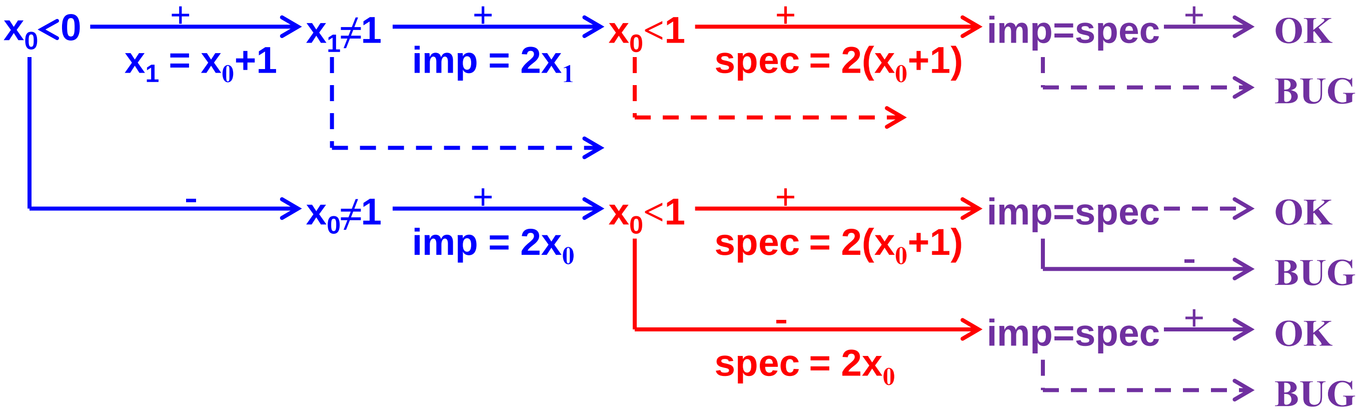
$x_0\neq 1$ $\xrightarrow{-}$ imp = $x_0$

$x_0<1$ $\xrightarrow{+}$ spec = $2(x_0+1)$

$x_0<1$ $\xrightarrow{-}$ spec = $2x_0$

imp=spec $\xrightarrow{+}$ OK

imp=spec $\xrightarrow{-}$ BUG



$x_0<0$ $\xrightarrow{+}$ $x_1 = x_0+1$ $\rightarrow$ $x_1\neq 1$ $\xrightarrow{+}$ imp = $2x_1$ $\rightarrow$ $x_0<1$ $\xrightarrow{+}$ spec = $2(x_0+1)$ $\rightarrow$ imp=spec $\xrightarrow{+}$ OK, BUG

$x_0<0$ $\xrightarrow{-}$ $x_0\neq 1$ $\xrightarrow{+}$ imp = $2x_0$ $\rightarrow$ $x_0<1$ $\xrightarrow{+}$ spec = $2(x_0+1)$ $\rightarrow$ imp=spec $\rightarrow$ OK, $\xrightarrow{-}$ BUG

$x_0<1$ $\xrightarrow{-}$ spec = $2x_0$ $\rightarrow$ imp=spec $\xrightarrow{+}$ OK, BUG

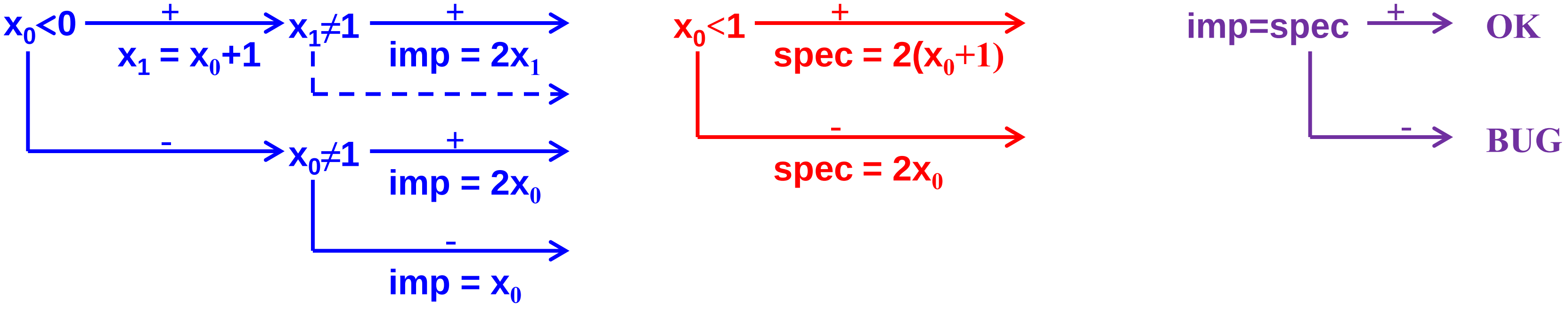
$x_0 = 0 \wedge 2x_0 = 2(x_0+1)$

$x_0 = 0 \wedge 2x_0 \neq 2(x_0+1)$

$x_0 > 1 \wedge 2x_0 = 2x_0$

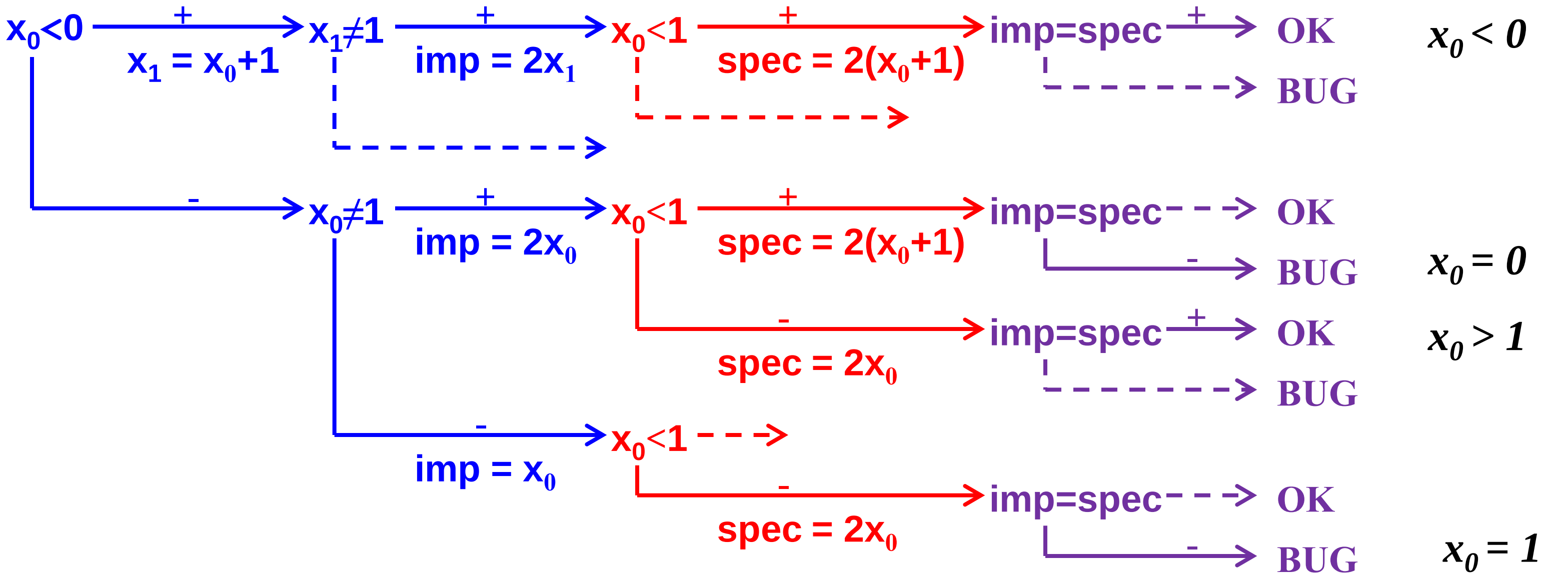
$x_0 > 1 \wedge 2x_0 \neq 2x_0$

# Cross-checking conformity with a specification


$x_0<0$ + $x_1 = x_0+1$ $x_1\neq1$ + imp = $2x_1$
- $x_0\neq1$ + imp = $2x_0$
- imp = $x_0$

$x_0<1$ + spec = $2(x_0+1)$
- spec = $2x_0$

imp=spec + OK
- BUG


$x_0<0$ + $x_1 = x_0+1$ $x_1\neq1$ + imp = $2x_1$ $x_0<1$ + spec = $2(x_0+1)$ imp=spec + OK BUG $x_0 < 0$
- $x_0\neq1$ + imp = $2x_0$ $x_0<1$ + spec = $2(x_0+1)$ imp=spec OK - BUG $x_0 = 0$
- spec = $2x_0$ imp=spec + OK BUG $x_0 > 1$
- imp = $x_0$ $x_0<1$
- spec = $2x_0$ imp=spec OK - BUG $x_0 = 1$

PathCrawler

# Example 12. Testing conformity with a specification

`Spec_Bsearch` is a specification for `Bsearch`, similar to the oracle. Test function `CompareBsearchSpec` that

- stores inputs, calls `Bsearch`,
- calls `Spec_Bsearch` to provide a verdict.

All-path testing will try cover all combinations of paths in `Bsearch` and `Spec_Bsearch`.

**"Customize test parameters" to see the predefined oracle and parameters. Confirm them and check the results.**

**Why are failures reported this time? Can you find the bug?**

# Assume/assert instead of precondition/oracle

**Preconditions filter** out cases with bad values of **inputs**

**Oracles check outputs**

We can also **check values at any point** in the source code:

***pathcrawler_assume(cond)***

to **filter** out cases where **cond is not satisfied**

***pathcrawler_assert(cond)***

to **check** if **cond is always satisfied,**

to force **search for a *counter-example*** by *creating a new branch* to explore

# Searching for run-time errors or anomalies

One way to **detect run-time errors and anomalies**
"**add a branch**" using `pathcrawler_assert`
to the source code at **each use** of **any partial operation**
(e.g. pointer de-referencing, division,…) and then
do structural testing to cover these branches

***May require a lot of tests!***

Better to restrict structural testing to
unconfirmed threats revealed by static analysis…

# The SANTE method: Static ANalysis and TEsting

SANTE calls

- static value analysis to prove some of threats safe and generate alarms for potential errors,

- structural testing only on reported alarms

[Chebaro et al., TAP 2010, TAP 2011, SAC 2012]

# Example 13. Confirming / invalidating threats in SANTE

Study Example 13.

“Test with predefined params” on pathcrawler-online.com

Check the results.

Is there any failure?

# Example 14. Confirming / invalidating threats in SANTE

Study Example 14.

“Test with predefined params” on pathcrawler-online.com

Check the results.

Is there any failure?

# Conclusion

Structural testing can be very useful to evaluate and complete test coverage

It also has many other uses

Test generation is automatic but the user must define the test parameters

This tutorial showed

- how to define a precondition, an oracle, an assertion
- test coverage criteria
- how to test conformity with a specification
- combined uses with static analysis

## Exercises

- Exercise 15. Are there errors? Complete test parameters. Check and explain the results.
- Ex. 16. Test with the predefined oracle. Find the bug.
- Ex. 17. Test with the predefined oracle. Find the bug.
- Ex. 18. Implement and test set operations `X∪Y, X∩Y, X-Y:`

```
int union(int X[], int Nx, int Y[], int Ny, int R[]);        // R=X ∪ Y
int intersection(int X[], int Nx, int Y[], int Ny, int R[]);// R=X ∩ Y
int complement(int X[], int Nx, int Y[], int Ny, int R[]);   // R=X – Y
```

All sets will be represented by sorted arrays of integers. Suppose X and Y are of size ≤5. Given sets X, Y with Nx, Ny elements resp. and an array R of sufficient size, each function writes the resulting set into R and returns the number of elements in R. Write oracles.

- Ex. 19. Use cross-checking for Ex18. Show its interest. (cf Ex.11,12)