Unit Testen en embedded software

Fout injectie en Software varianten

Gerard Fianen

INDES Integrated Development Solutions BV

Agenda

Ontwikkelingen in Unit Test & Code Coverage

- Software varianten test
- Fout Injectie

Ontwikkelingen in formele Statische Analyse

- Worst Case Stack usage analysis
- Worst Case Timing Analysis
- Proven correct C-Code

Begrippen

Unit test
is a level of software testing where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output.

Code Coverage
To determine what proportion of your project's code is actually being tested by unit tests or integration tests, you can use the code coverage. There are different kinds of coverage measurements :

- Statement Coverage (C0)
- Branch Coverage (C1)
- Decision Coverage (DC)
- Modified Condition / Decision Coverage (MC/DC)
- Multiple Condition Coverage (MCC)
- Entry Point Coverage (EPC)
- Function Coverage (FC)

Software varianten (Software Product Line)

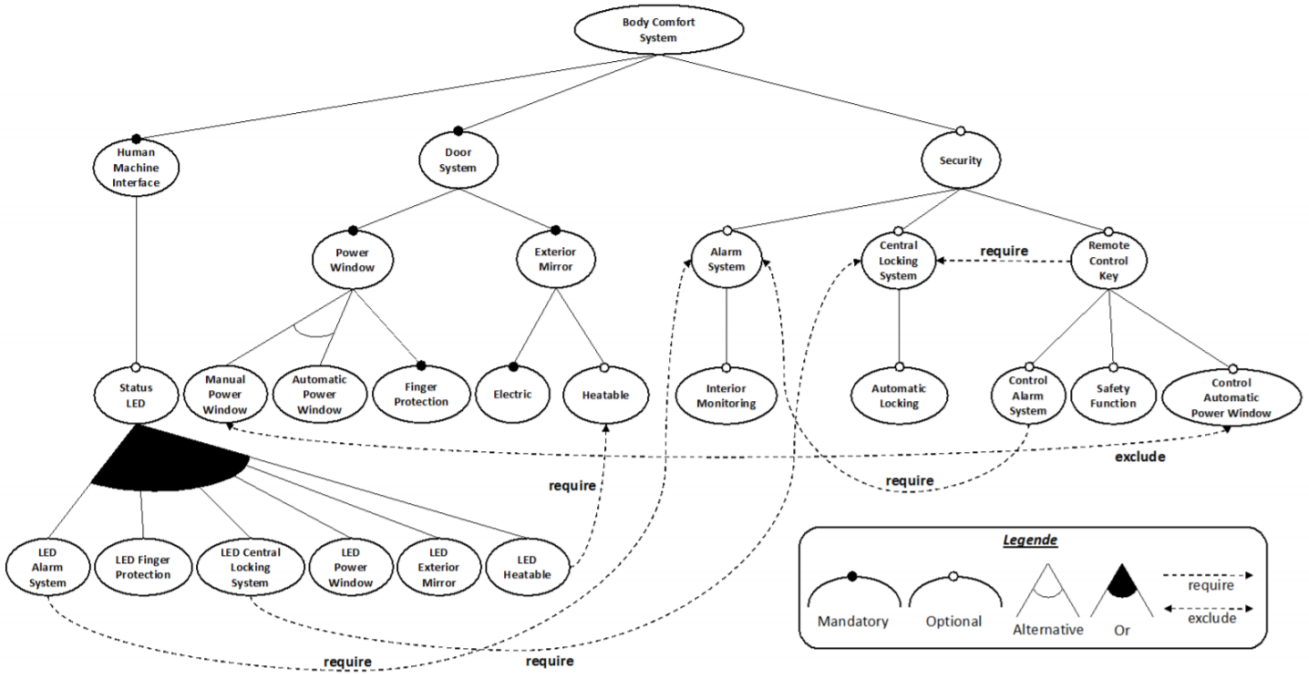
There are various possibilities to create software variants (e.g. C/C++ source code):

- Enabling/disabling of code parts by defines
- Generating code variants with tools (e.g. out of MATLAB)
- Copying, renaming, and changing the source file
- Executing identical sources on different hardware platforms

Case Study – Body Comfort System 2

28 Features, 11616 Product Variants, 1 Core Product, 40 Deltas

16 Products for Pair-Wise Feature Coverage



Bron : TU Braunschweig

Het doel: Test coverage

```
13 ret_t check_level(int level)
14 {
15 #ifdef VARIANT 1
16     level+= supplementary_level;
17 #endif
18
19     if (level < MINIMUM) {
20         return alarm;
21     }
22
23     return  ok;
24 }
25
```

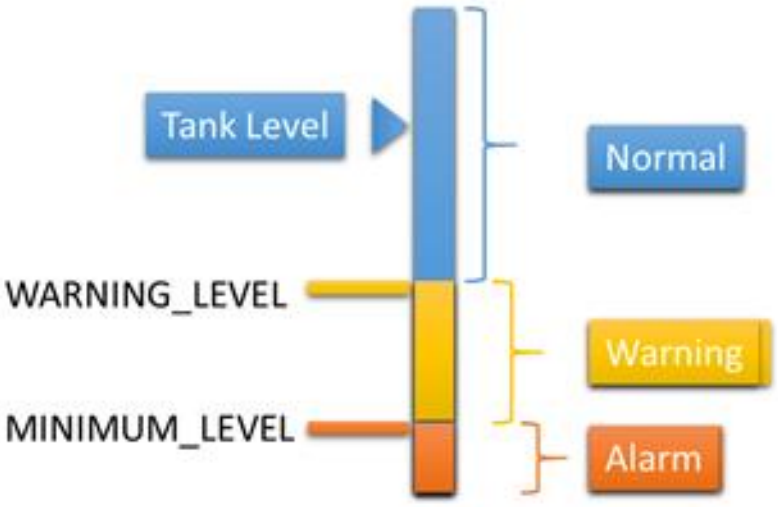
Example :

Variant specific code could be added up to the measured value

Programming error :
Missing of an addition operator in line 16

Error could remain undiscovered if
the commonly used code in line 19-23
remains untested in the variant.

Implementation Example



- Base
  - Passenger car
  - Truck
    - Truck with fixed supplementary tank
    - Truck with optional supplementary tank

fuel_sensor.c

```c
// Fuel level example
//
// Returns warning/alarm depending on the fuel level

#include "fuel_sensor.h"

tank_control_t tank;

#ifdef TRUCK
tank_control_t supplementary_tank;
#endif

//
//
fuel_level_t calculate_fuel_level() {
    short current_level = tank.level;

#ifdef TRUCK
    if (supplementary_tank.is_active) {
        current_level += supplementary_tank.level;
    }
#endif

    if (current_level < MINIMUM_LEVEL) {
        return alarm;
    }
    if (current_level < WARNING_LEVEL) {
        return warning;
    }
    return normal;
}
```

D&E event 2018

TESSY | Project: Variants

File Edit Source Window Help

C/C++ | Requirement Management | Overview | TIE - Test Interface Editor | CTE - Classification Tree Editor | TDE - Test Data Editor | SCE - Scenario Editor | CV - Coverage Viewer | IDA - Interface Data Assigner

Project Explorer

Tessy

fuel_sensor.c

```
1 // Fuel level example
2 //
3 // Returns warning/alarm depending on the fuel level
```

Outline

D:\TESSY_Demo\Projects\Variants
fuel_sensor.h
tank : tank_control_t
supplementary_tank : tank_contro
calculate_fuel_level() : fuel_level_t

TEE - Tessy Environment Editor [$(PROJECTROOT)\tessy\c

File Edit Options Help

Create configuration for „TRUCK“

- General
  - Compiler
  - Targets
  - Environments
- Configurations
  - File
    - GNU GCC
      - GNU GVD
      - Eclipse CDT
        - (Default)
    - GNU GCC (TRUCK)

Copy Environment	Einfg
Delete	Entf
Rename	F2
Restore Factory Value	Strg+R
Enable Environment	Strg+E
Expand Subtree	
Collapse Subtree	
Expand All	Strg+A

	Value
ASAP Project	
Batchtest Wait Time	0
Buffer Size	10240
Call Count Size	8
Compiler Call	gcc -c $(USEROPTIONS) -DTESSY -o $(OUTPUT) $(INPUT)
Compiler Commandline	
Compiler Concurrency	0
Compiler Defines	
Compiler Dialect	
Compiler Id	0x[illegible]
Compiler Includes	
Compiler Install Path	$(TE[illegible]SY_MSYS_MIN[illegible]W_ROOT)
Compiler Version	
Component Test	false
Constructor Exclude List	
Conversion DLL	none
Default Call Trace Evaluation	Ignore Call Trace
Defines Exclude List	*TESSY*, unix, _*, *i386*, *X86*, WIN32, WINNT
Display Name	$(Compiler) --- $(Target) - $(Environment)
Enable ASAP	false
Enable C0 Coverage	true
Enable C1 Coverage	true
Enable C99	true
Enable CLANG	true
Enable Create Default Constructors	true
Enable Create Function Stubs	false
Enable Create Method Stubs	true
Enable DC Coverage	true
Enable Define Variables	true
Enable EPC Coverage	true

Project Root: D:\TESSY_Demo\Projects\Variants

00:40

D&E event 2018

TESSY | Project: Variants

File Edit Window Help

Requirement Management | Overview | TIE - Test Interface Editor | CTE - Classification Tree Editor | TDE - Test Data Editor | SCE - Scenario Editor | CV -

Test Project

Filter test project

C1

- Base tests
  - Engine Control
    - Inject
    - Speed
  - Fuel System
    - Measure
      - calculate_fuel_level
- Truck tests
  - Engine Control
    - Inject
    - Speed
  - Fuel System

D&E event 2018

Test cases for the variant “passenger car“

Test Items

Name				Description	Specification
1 (1) Single tank	-	-	-		- tank equipment:single tank...
2 (1)	-	-	-		- tank equipment:single tank...
3 (1)	-	-	-		- tank equipment:single tank...
3.1	-	-	-		
4 (1)	-	-	-		- tank equipment:single tank...
4.1	-	-	-		
5 (1) Main + supplei	-	-	-		- tank equipment:main and supple...
6 (1)	-	-	-		- tank equipment:main and supple...
7 (1)	-	-	-		- tank equipment:main and supple...
8 (1)	-	-	-		- tank equipment:main and supple...

D&E event 2018

Rules for test case inheritance

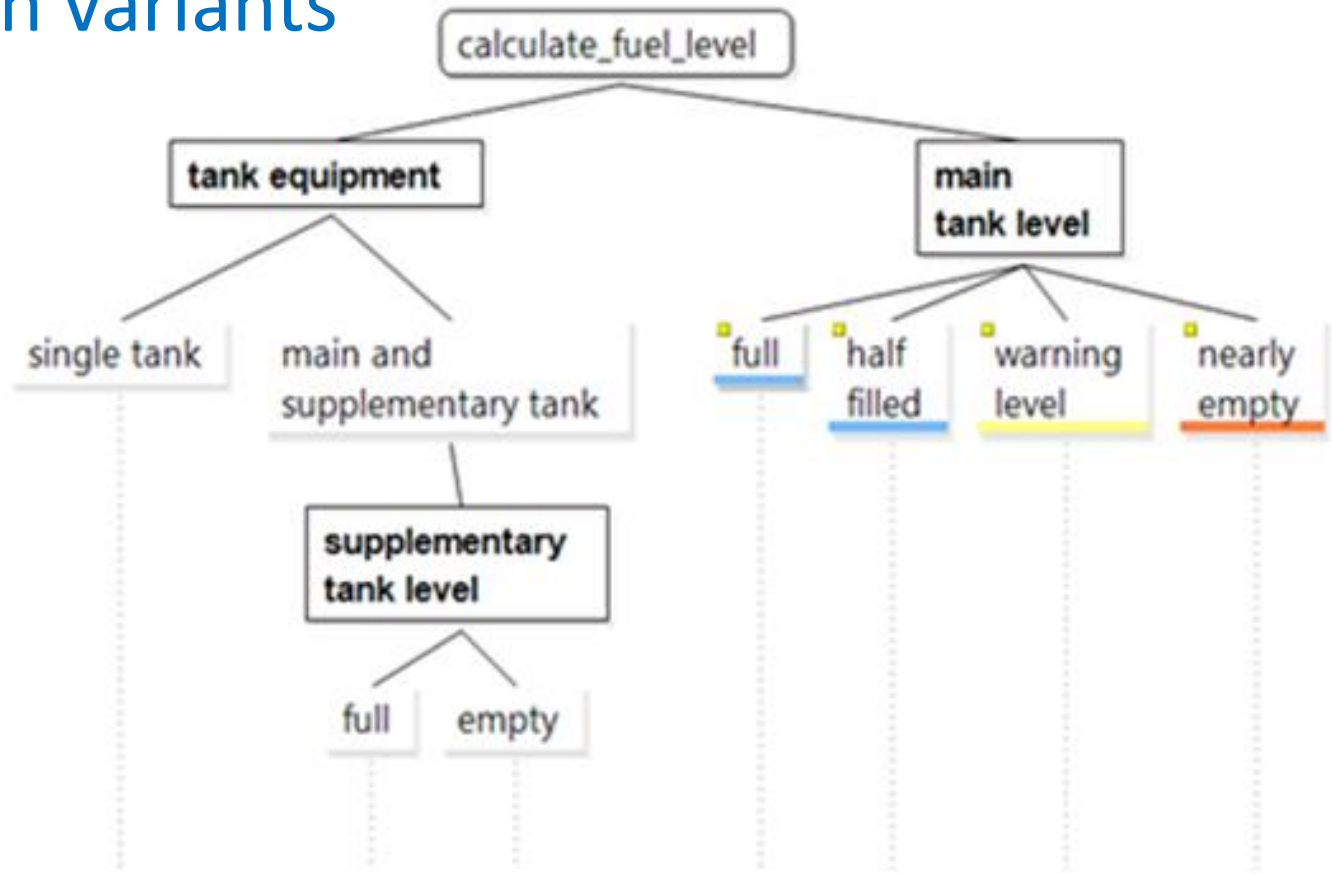
Inheritance operations :

- Change of inherited test data
- Deleting/hiding of inherited test cases
- Adding additional test cases

Variable values statuses can be result of :

- Value was inherited
- Value was inherited and overwritten
- Value was defined locally for this variant test

Testspecification variants



Test Table

1: Single tank	●	○	○	●	○	○	○
2:	●	○	○	○	○	○	○
3:	●	○	○	○	○	●	○
4:	●	○	○	○	○	○	●
5: Main + supplementary tank	○	●	○	○	○	●	○
6:	○	○	●	○	○	●	○
7:	○	●	○	○	○	○	●
8:	○	○	●	○	○	○	●

D&E event 2018

Color coding of inherited and overwritten values

Base Tests

Test Data of 'calculate_fuel_level'

type filter text

	1.1	2.1
Inputs		
Globals		
tank_control_t tank		
short level	MAXIMUM_LEVEL	40
Outputs		
Return		
fuel_level_t	normal	normal

Tests for Variant „Truck“

Test Data of 'calculate_fuel_level'

type filter text

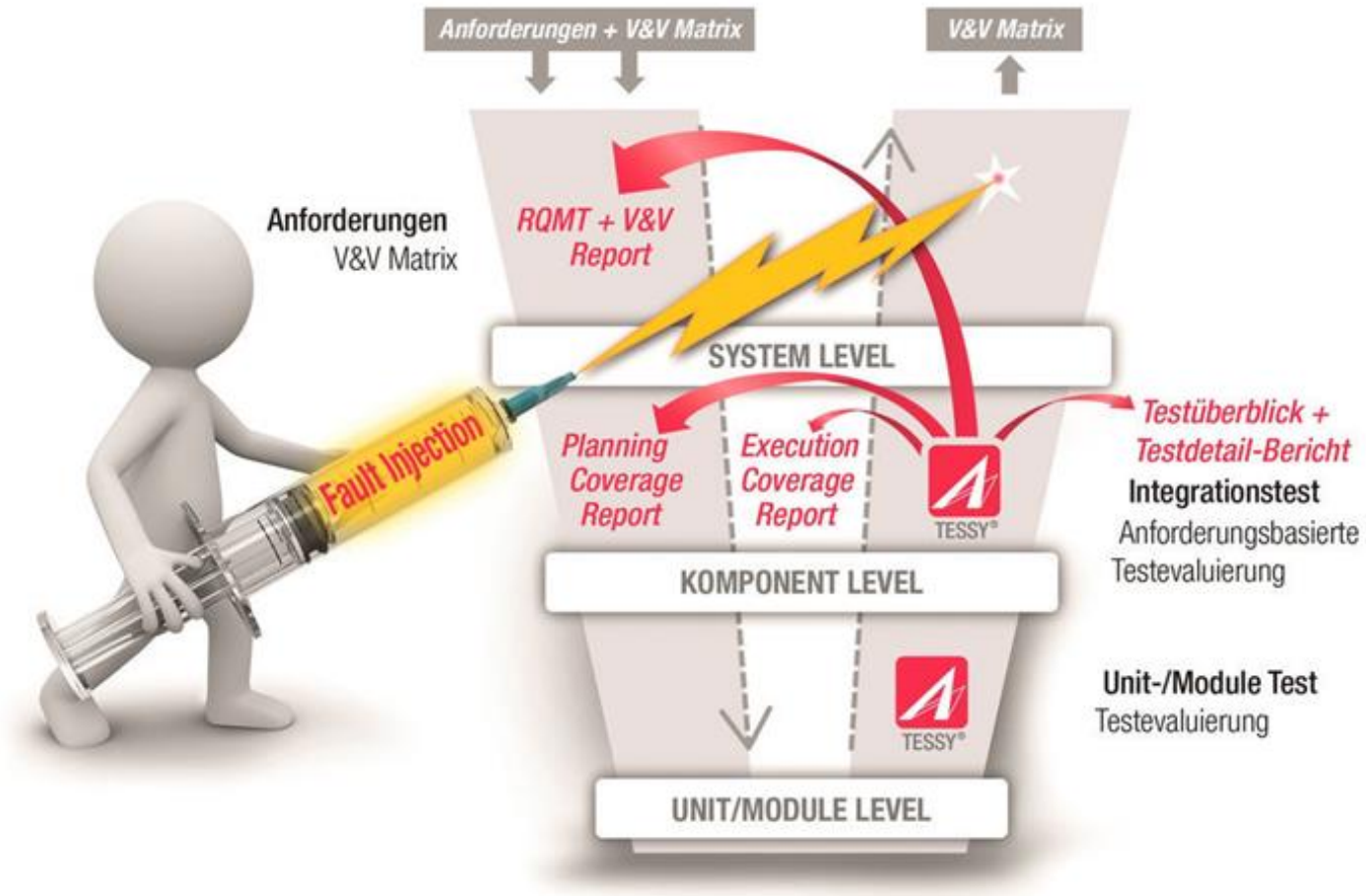
	1.1	2.1
Inputs		
Globals		
tank_control_t supple		
short level	"none"	"none"
short is_active	0	0
tank_control_t tank		
short level	MAXIMUM_LEVEL	500
Outputs		
Return		
fuel_level	normal	normal

Inherited values are highlighted in blue

Overwritten values are highlighted in purple

D&E event 2018

Fault injection

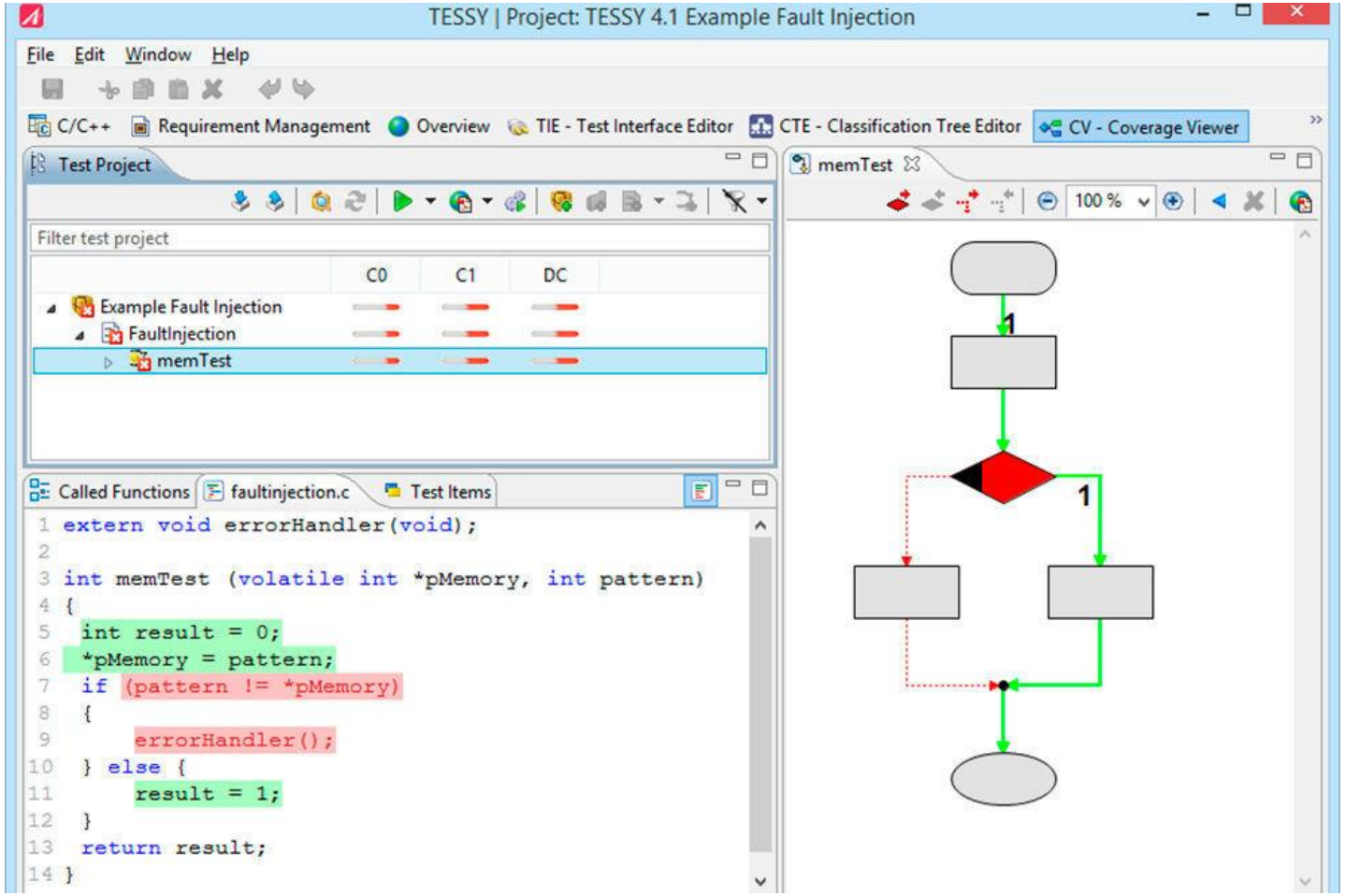


Fault injection

```
1  extern void errorHandler(void);
2
3  int memTest (volatile int *pMemory, int pattern)
4  {
5      int result = 0;
6      *pMemory = pattern;
7      if (pattern != *pMemory)
8      {
9          errorHandler();
10     } else {
11         result = 1;
12     }
13     return result;
14 }
```

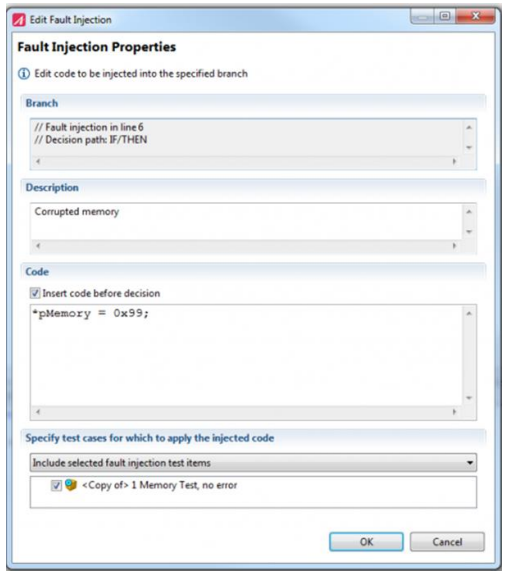
In regel 6 wordt in het geheugen geschreven.
In regel 7 wordt getest of inderdaad de goede waarde in het geheugen staat.

Fault injection

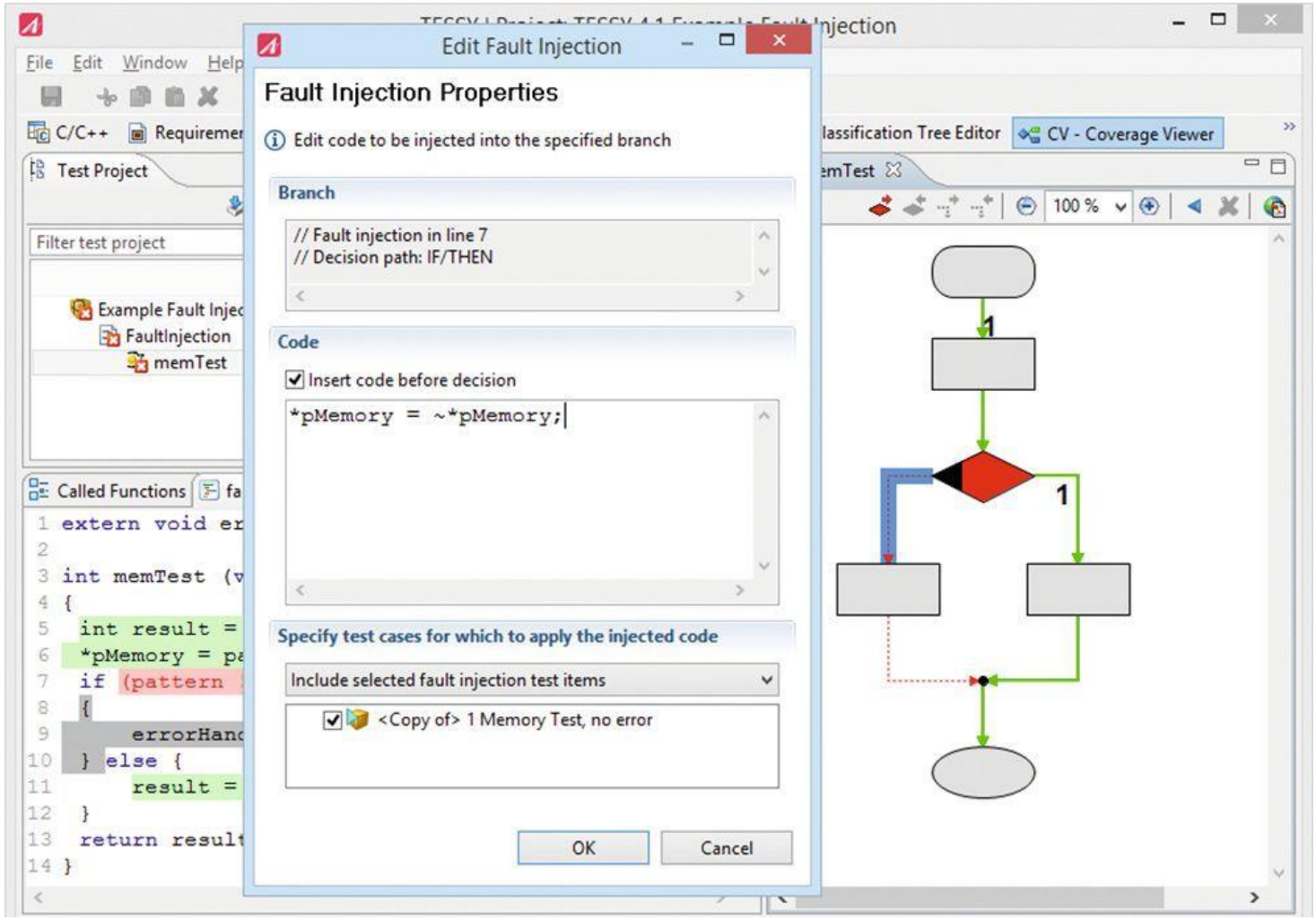


Geen 100% Code-Coverage

Fault injection



Fault injections are created based on unreached branches of the function flow graph.



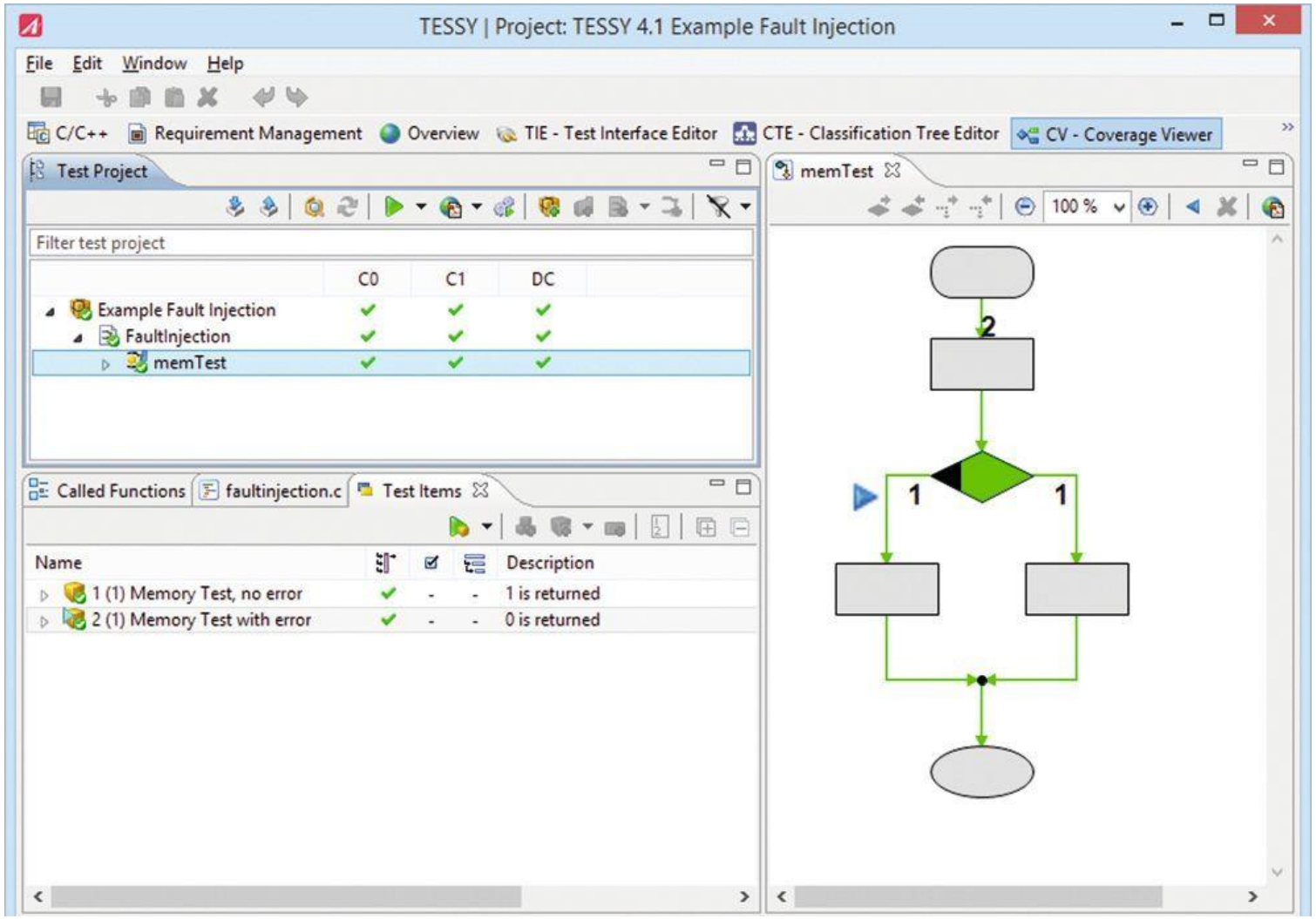
Fault injection

```
6       *pMemory = pattern;
7       #ifdef FAULTINJECTION
8           if (tstFaultInjection != 0)
9               {
10                  *pMemory = ~*pMemory;
11              }
12      #endif
13      if (pattern != *pMemory)
```

Fault injection

100% Coverage !

In regression testing, fault injections are automatically placed in the correct location in the source code even after code changes.



Ontwikkelingen in (formele) Statische Analyse

Formele verificatie door toepassing van Abstact Interpretation

Scope:

- Binary code: worst-case stack usage , worst-case execution time
- Source code: violations of coding rules, run-time errors, data races

“Sound” tools

- Verification is correct and exhaustive. Never yield false negatives.

Toelichting : Abstract interpretation (hidden slide)

abstract interpretation is a theory of sound approximation of the semantics of computer programs, based on monotonic functions over ordered sets, especially lattices. It can be viewed as a partial execution of a computer program which gains information about its semantics (e.g., control-flow, data-flow) without performing all the calculations.

Its main concrete application is formal static analysis, the automatic extraction of information about the possible executions of computer programs; such analyses have two main usages:

- inside compilers, to analyse programs to decide whether certain optimizations or transformations are applicable;

- for debugging or even the certification of programs against classes of bugs.

Sound tools guarantee that the verification they perform is correct and exhaustive. They can never yield false negatives, but by undecidability may produce false alarms (or false positive) signaling a potential error with no instance during any execution (because the static analysis is not precise enough to eliminate the potential error).

More: https://en.wikipedia.org/wiki/Abstract_interpretation

Worst-Case Stack Height Analysis

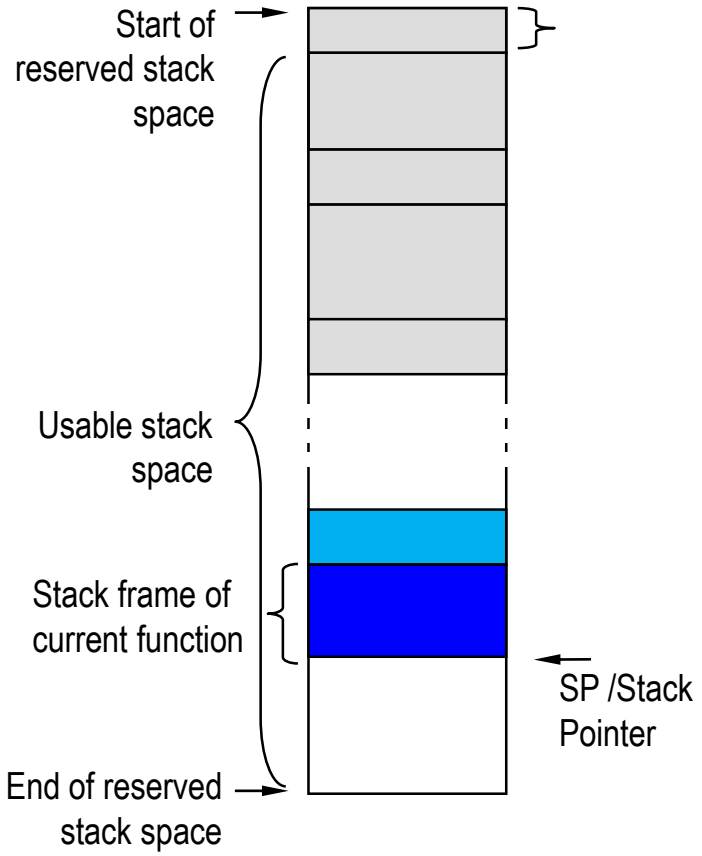
Stack space has to be reserved at configuration time => maximal stack usage has to be known in advance.

A traditional approach: pollution checks

- Fill the stack area with a pattern (0xAAAA)
- Let the system run for a long time
- Monitor the maximum stack usage so far

Error-prone and expensive!

- Typical stack usage of a task can be very different from maximum stack usage.
- Dynamic testing typically cannot guarantee that the worst case stack usage has been observed.

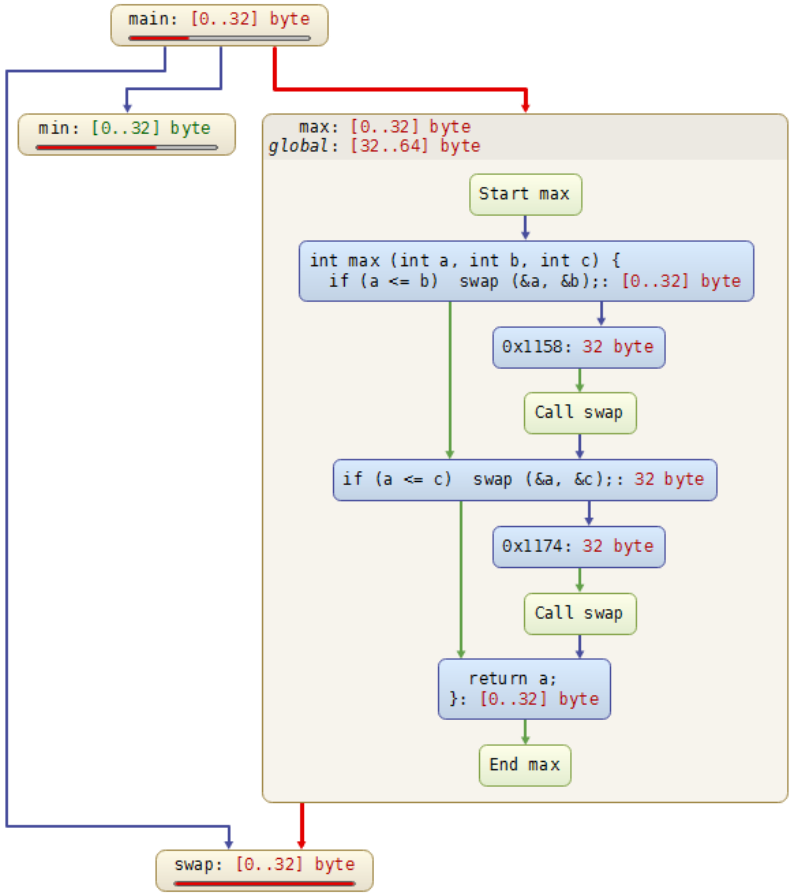


StackAnalyzer: Static Stack Usage Analysis

StackAnalyzer is an Abstract Interpretation based static analyzer which calculates safe and precise upper bounds of the maximal stack usage of the tasks in the system.

It can prove the absence of stack overflows:

- on binary code
- without code modification
- taking into account loops and recursions
- taking into account inline assembly and library function calls



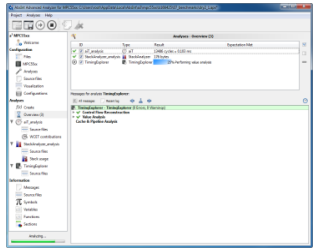
Computing the Worst-Case Stack Height

Executable (elf,coff,…)

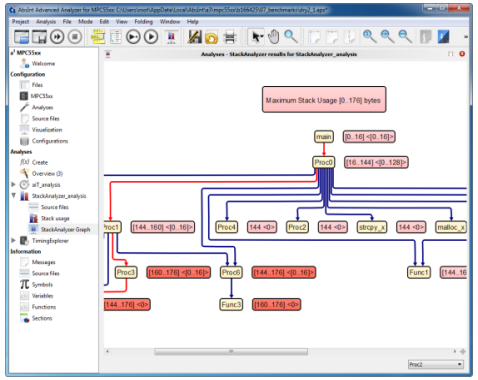
Function pointers, recursion depths, …

```
instruction "_main" + 1 computed
calls "_fooA", "_fooB", "_fooC";
routine "_fib" incarnates max 5;
```

StackAnalyzer



- Stack Usage
- Visualization
- Documentation



StackAnalyzer computes safe upper bounds of the stack usage of the tasks in a program for all inputs

Static program analysis based on Abstract Interpretation

Worst-Case Timing Analysis

Application Code

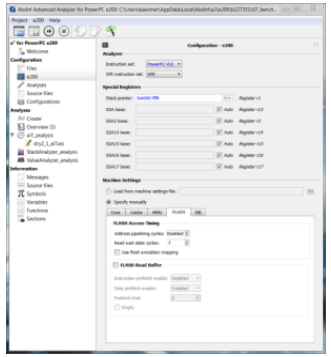
```
void Task (void) {
  variable++;
  function();
  next++:
  if (next)
    do this;
  terminate()
}
```

Compiler
Linker

Executable
(*.elf /*.out)

Specifications (*.ais)

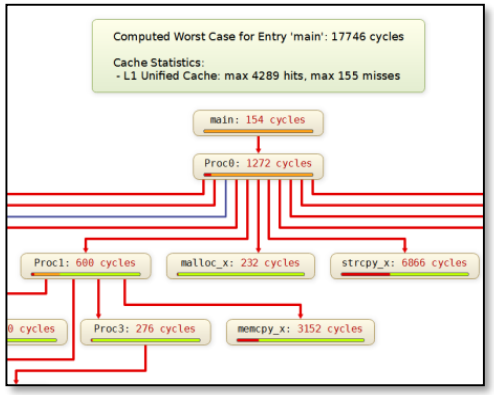
```
clock 10200 kHz ;
loop "_codebook" + 1 loop exactly 16 end;
recursion "_fac" max 6;
snippet "printf" is not analyzed and takes max 333 cycles;
flow "U_MOD" + 0xAC bytes / "U_MOD" + 0xC4 bytes is max 4;
area from 0x20 to 0x497 is readonly;
```

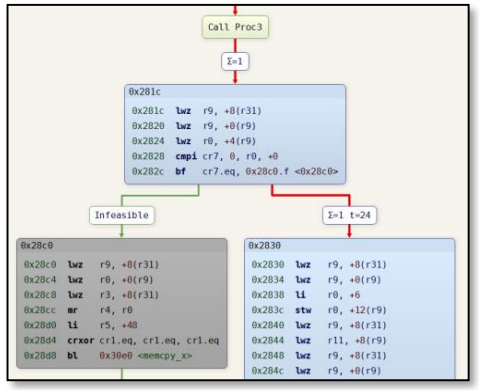


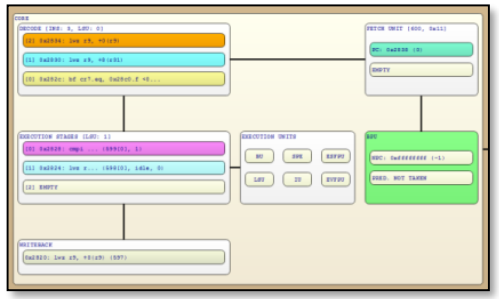
Entry Point

Program-Flow Traces

Worst Case Execution Time
+ Visualization, Reporting







Worst Case Execution Time (WCET)
estimate based on local tracing information
+ Trace Coverage report
+ Time Variance report over all traces
+ Visualization, Reporting

D&E event 2018

Toelichting : Worst-Case Timing Analysis

- Global static program analysis by Abstract Interpretation (sound): microarchitecture analysis (caches, pipelines, …) + value analysis
- Integer linear programming for path analysis
- Safe and precise bounds on the worst-case execution time

Meer over de tools in deze presentative :

Tessy Unit test & code Coverage :
www.razorcat.com
https://www.razorcat.com/en/product-tessy.html

https://www.absint.com/products.htm

www.indes.com info@indes.com Tel : 0345 – 545.535

INDES –
Integrated Development Solutions BV

Cross Compilers, Debuggers, IDE
RTOS, Middelware, Protocol stacks, GUI, Database
Debug & Trace probes, Emulators
Real-Time Trace, RTOS-Event Trace
Static Analysis, Timing Analysis, Stack Analysis
Unit Test, Code Coverage
System-level Test

PoE conformance test, Ethernet-PHY test

Bezoek ons op stand 27

www.indes.com info@indes.com Tel : 0345 – 545.535