Considering Execution Environment Resilience: A White-Box Approach

Stefan Klikovits[1,2], David PY Lawrence[1,3],
Manuel Gonzalez-Berges[2], Didier Buchs[1]

[1]Université de Genève, Carouge, Switzerland
[2]CERN, Geneva, Switzerland
[3]Honeywell International Sarl., Rolle, Switzerland

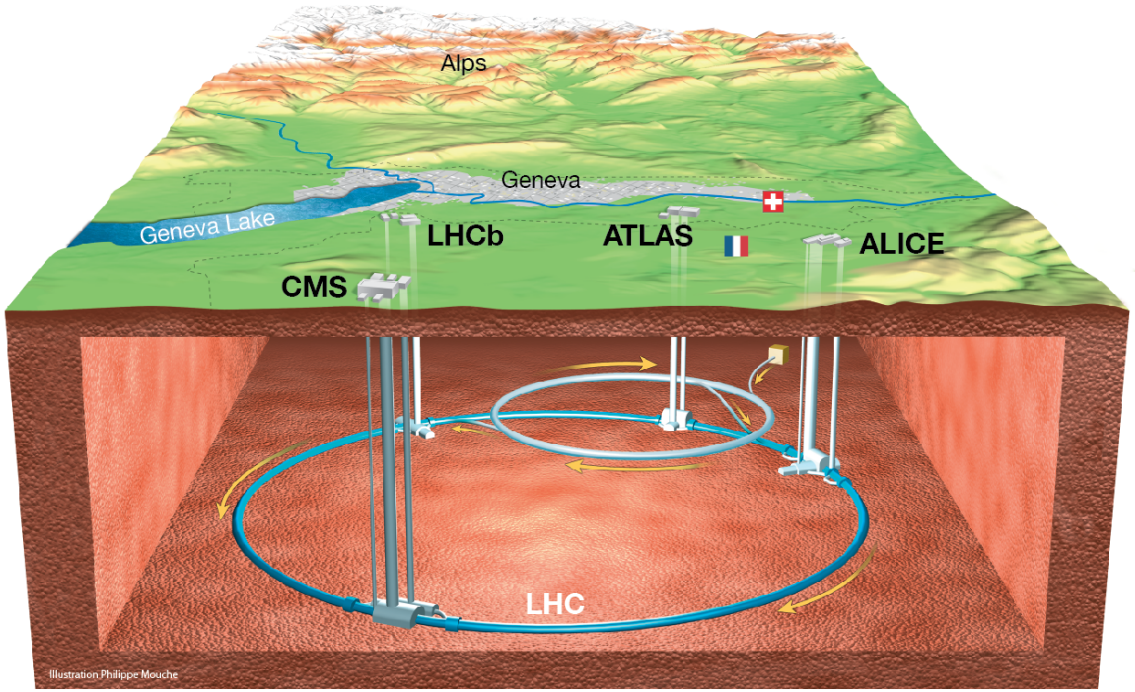
Tuesday 11th August, 2015

What is this all about?

How to

- generate test cases w. little user interaction
- on a large scale
- unit/component level

Welcome to CERN



Credit: *CERN (www.cern.ch)*

Welcome to CERN

- LHC, experiments, infrastructure (e.g. power grid)
- large-scale, widespread, complex systems
- many types of hard- and software
- $>$ 100 subsystems,
  10,000s of devices,
  100,000s of measurement points
- thousands of physicists/engineers/workers

high reliability and resilience expectations

How do we supervise it?

SIMATIC
WinCC Open Architecture

- two frameworks (UNICOS, JCOP) built on top
- *Control* (CTRL): proprietary scripting language

So where is the problem?

- until recently no automated unit test support
- frequent changes in execution environment
- (mostly) manual verification
- big expenses (time) on QA side

Testing

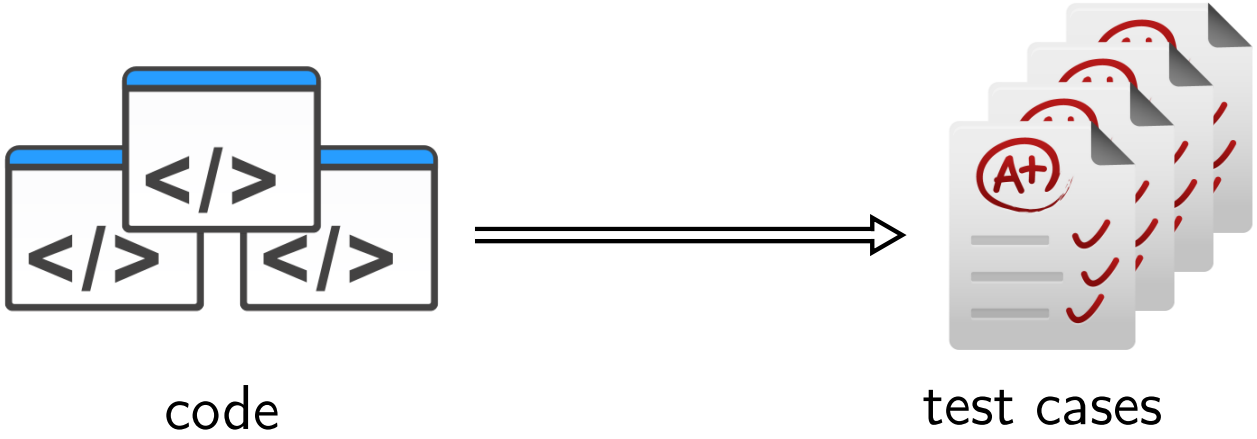
```
1  f(x){
2    if GLOBAL_VAR:
3      return dbGet(x)
4    else:
5      return -1
6  }
```

f(x)

```
1  test_f(){
2    dbSet("test",5) // prepare
3    GLOBAL_VAR = True
4    x = f("test") // act
5    assert(x == 5) // assert
6  }
```

Test case for f(x)

What do we want?



Iterative TEst Case System

- regression testing
- consider dependencies
- automatic test case generation (ATCG)
- build on existing research & tools
- generate unit & component tests

Automated Test Case Generation

- source code based
- black-box (function signature) vs. white-box (function body)

Semi-purification

- replace dependencies with parameters

```
1  f(x){
2    if GLOBAL_VAR:
3      return dpGet(x)
4    else:
5      return -1
6  }
```

A non-pure function

```
1  f_sp(x,a,b){
2    if a:
3      return b
4    else:
5      return -1
6  }
```

Semi-purified `f(x)`

Semi-purification

- replace dependencies with parameters

```
1  f(x){
2    if GLOBAL_VAR:
3      return dpGet(x)
4    else:
5      return -1
6  }
```

A non-pure function

```
1  f_sp(x,a,b){
2    if a:
3      return b
4    else:
5      return -1
6  }
```

Semi-purified `f(x)`

```
1  test_f_sp(){
2    x = f("test",True,5) // act
3    assert(x == 5) // assert
4  }
```

Test case

Semi-purification (cont.)

- replace dependencies with parameters

```
1  functionA(x){
2    a = functionB(x)
3    return a
4  }
5
6  functionB(x){
7    b = GLOBAL_VAR
8    b++
9    return b
10 }
```

Function with SRC

```
1  functionA_sp(x,y){
2    a = functionB(x,y)
3    return a
4  }
5
6  functionB_sp(x,y){
7    b = y
8    b++
9    return b
10 }
```

Semi-purified w. SRC

Semi-purification (cont.)

- replace dependencies with parameters

```
1  functionA(x){
2    a = functionB(x)
3    return a
4  }
5
6  functionB(x){
7    b = GLOBAL_VAR
8    b++
9    return b
10 }
```

Function with SRC

```
1  functionA_sp(x,y){
2    a = functionB(x,y)
3    return a
4  }
5
6  functionB_sp(x,y){
7    b = y
8    b++
9    return b
10 }
```

Semi-purified w. SRC

Semi-purification: Concept

- code contains dependencies
  - global variables, data base values, subroutine calls, other resources
- manual way: test doubles (mocks, stubs, fakes, . . .) [ME06]
- remove dependencies
  - based on localization [SW03, SK13]
  - input parameters instead of dependencies
  - use any ATCG (black- and white-box)

"Shortcut"

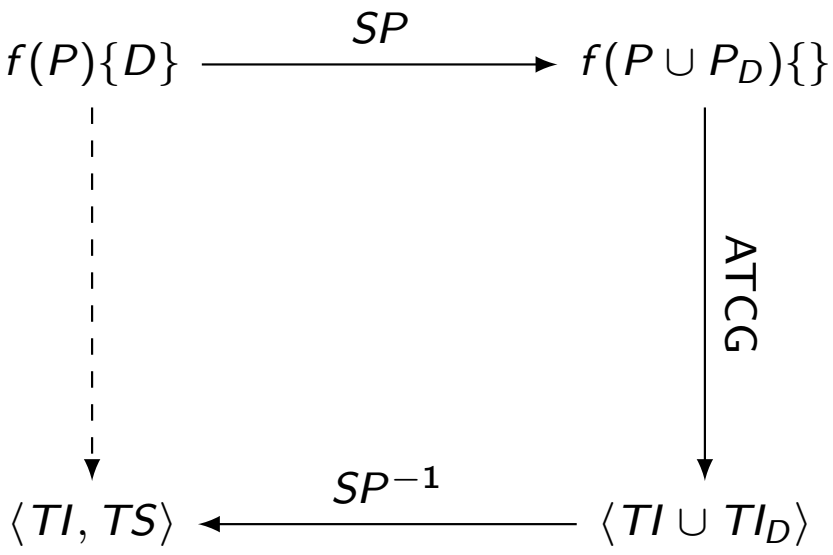


Figure: Test case generation schema

- P . . . Parameters
- D . . . Dependencies
- TI . . . Test Input
- TS . . . Test Setup Routine

Identified Bottlenecks

- Loops
- Shared Subroutines
- Concurrency

SP: Bottlenecks (cont.)

- Loops

```
1  sleepUntilReady(){ // a = bool
2
3    while dpGet(notReadyDP):
4      sleep(5) // sleep for 5 seconds
5
6  }
```

A semi-purified loop

SP: Bottlenecks (cont.)

- Loops

```
sleepUntilReady(a){ // a = bool

  while a:// replaces dpGet(notReadyDP)
    sleep(5) // sleep for 5 seconds

}
```

A semi-purified loop

Test Cases:

- a: False ⇒ loop not executed
- a: True ⇒ endless loop

SP: Bottlenecks (cont.)

- Loops

```
1  sleepUntilReady(a){ // a = [bool]
2    i = 0
3    while a[i]: // replaces dpGet(notReadyDP)
4      sleep(5) // sleep for 5 seconds
5      i++
6  }
```

A semi-purified loop

Test Cases:

- a: [False]⇒ loop not executed
- a: [True, True, . . . , False] ⇒ loop execution

Questions:

- how long should the list be?
- how to modify correctly?
- Test modified code or w. threads?

SP: Bottlenecks (cont.)

- Shared subroutine dependencies

```
1  var SPEED_VAR = 1
2  adjustSpeed(){
3    x = getTheSpeed()
4    if x < 10 :
5      doubleTheSpeed()
6  }
```

CUT

```
1  getTheSpeed(){
2    return SPEED_VAR
3  }
```

Subroutine 1

```
1  doubleTheSpeed(){
2    speed = SPEED_VAR
3    SPEED_VAR
```

Subroutine 2

SP: Bottlenecks (cont.)

- Shared subroutine dependencies

```
var SPEED_VAR = 1
adjustSpeed(){
  x = getTheSpeed()
  if x < 10 :
    doubleTheSpeed()
}
```

CUT

```
getTheSpeed(){
  return SPEED_VAR
}
```

Subroutine 1

```
doubleTheSpeed(){
  speed = SPEED_VAR
  SPEED_VAR
```

Subroutine 2

SP: Bottlenecks (cont.)

- Shared subroutine dependencies

```
1
2  adjustSpeed(a){
3    x = getTheSpeed(a)
4    if x < 10 :
5      doubleTheSpeed()
6  }
```

CUT

```
1  getTheSpeed(a){
2    return a // SPEED_VAR
3  }
```

Subroutine 1

```
1  doubleTheSpeed(){
2    speed = SPEED_VAR
3    SPEED_VAR
```

Subroutine 2

SP: Bottlenecks (cont.)

- Shared subroutine dependencies

```
1
2  adjustSpeed(a, b){
3    x = getTheSpeed(a)
4    if x < 10 :
5      doubleTheSpeed(b)
6  }
```

CUT

```
1  getTheSpeed(a){
2    return a // SPEED_VAR
3  }
```

Subroutine 1

```
1  doubleTheSpeed(b){
2    speed = b // SPEED_VAR
3    b = speed*2
```

Subroutine 2

SP: Bottlenecks (cont.)

- Shared subroutine dependencies

```
1
2  adjustSpeed(a, b){
3    x = getTheSpeed(a)
4    if x < 10 :
5      doubleTheSpeed(b)
6  }
```

CUT

```
1  getTheSpeed(a){
2    return a // SPEED_VAR
3  }
```

Subroutine 1

```
1  doubleTheSpeed(b){
2    speed = b // SPEED_VAR
3    b = speed*2
```

Subroutine 2

SP: Bottlenecks (cont.)

- Shared subroutine dependencies

```
1
2  adjustSpeed(a, b){
3    x = getTheSpeed(a)
4    if x < 10 :
5      doubleTheSpeed(b)
6  }
```

CUT

```
1  getTheSpeed(a){
2    return a // SPEED_VAR
3  }
```

Subroutine 1

```
1  doubleTheSpeed(b){
2    speed = b // SPEED_VAR
3    b = speed*2
```

Subroutine 2

SP: Bottlenecks (cont.)

- Shared subroutine dependencies

```
1
2  adjustSpeed(a, b){
3    x = getTheSpeed(a)
4    if x < 10 :
5      doubleTheSpeed(b)
6  }
```

CUT

```
1  getTheSpeed(a){
2    return a // SPEED_VAR
3  }
```

Subroutine 1

```
1  doubleTheSpeed(b){
2    speed = b // SPEED_VAR
3    b = speed*2
```

Subroutine 2

- BUT: a and b replace the same dependency

SP: Bottlenecks (cont.)

- Concurrency

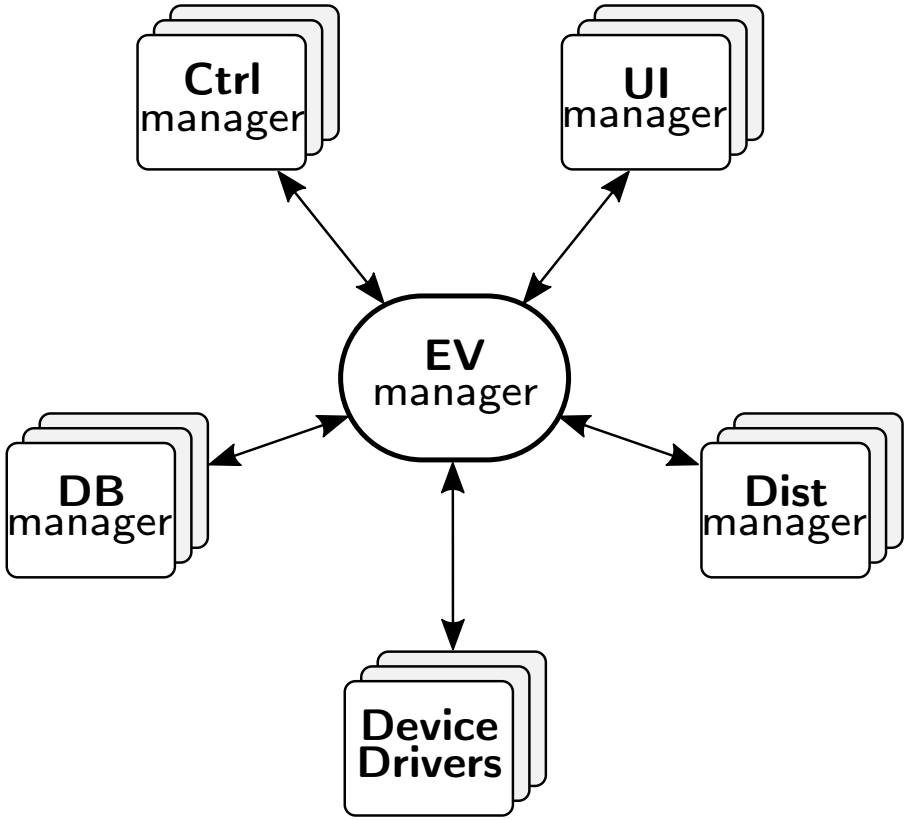


Figure: WinCC OA's manager concept

SP: Bottlenecks (cont.)

- access to ALL data points
- 100+ sub-systems
- discover dirty read/write scenarios (`adjustSpeed()`)

ITEC implementation

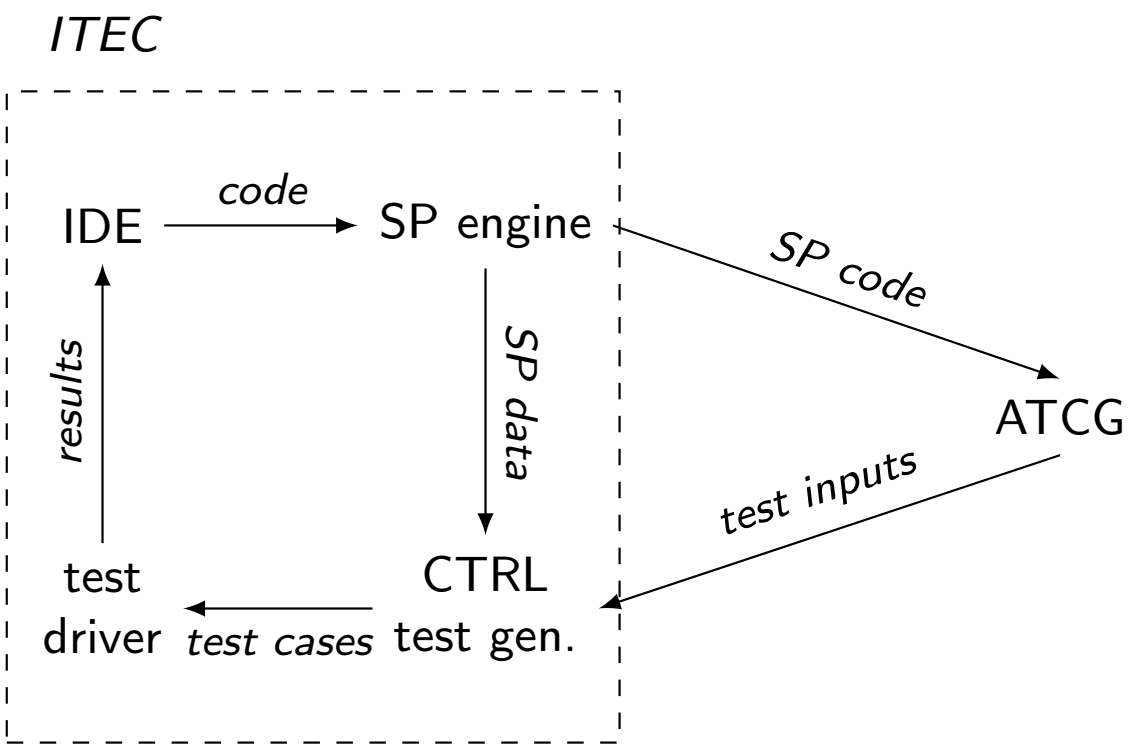


Figure: ITEC workflow

ITEC implementation

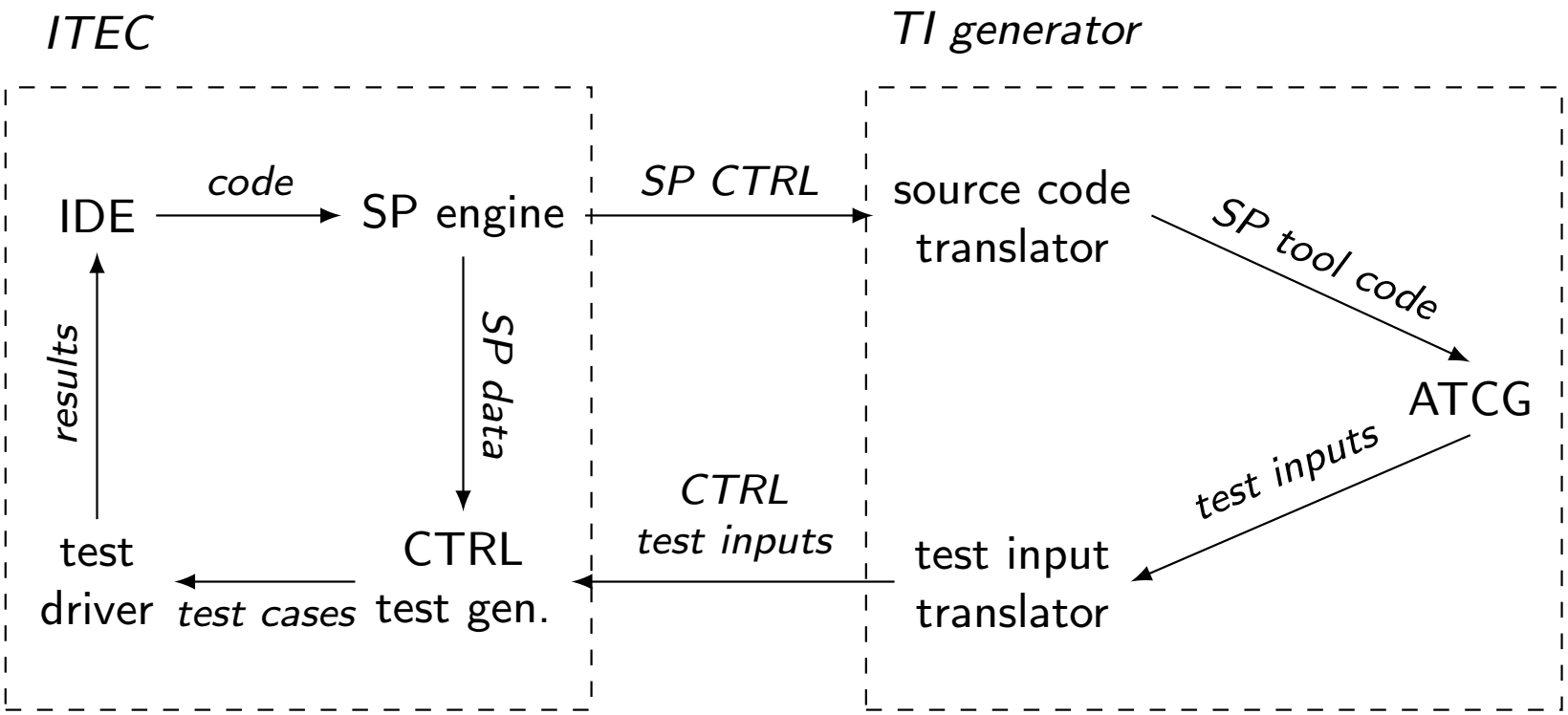


Figure: ITEC workflow

Conclusion

Semi-purification

- remove dependencies
- facilitate unit tests generation
- use black-box techniques on all code

Future works

What's next?

- connect to test framework & roll-out
- intrinsic domain information
- compare different ATCG
- research into bottlenecks

References

[ME06] Meszaros, G.:
XUnit Test Patterns: Refactoring Test Code, Chapter 23. Test Double Patterns, pages 521–590.
Prentice Hall PTR, Upper Saddle River, NJ, USA, (2006)

[SC03] Sward, R. E., Chamillard, A. T.:
Re-engineering Global Variables in Ada.
In: Proc. 2004 ACM SIGAda international conference on Ada, pp. 29–34, ACM, New York, (2003).

[SK13] Sankaranarayanan, H., Kulkarni, P.:
Source-to-Source Refactoring and Elimination of Global Variables in C Programs.
In: Journal of Software Engineering and Applications, Vol. 6 No. 5, pp. 264–273, (2013).

Considering Execution Environment Resilience: A White-Box Approach

Stefan Klikovits[1,2], David PY Lawrence[1,3],
Manuel Gonzalez-Berges[2], Didier Buchs[1]

[1]Université de Genève, Carouge, Switzerland
[2]CERN, Geneva, Switzerland
[3]Honeywell International Sarl., Rolle, Switzerland

Tuesday 11th August, 2015