Unit- and Sequence Test Generation with HOL-TestGen

Tests et Methodes Formelles

Prof. Burkhart Wolff
Univ - Paris-Sud / LRI

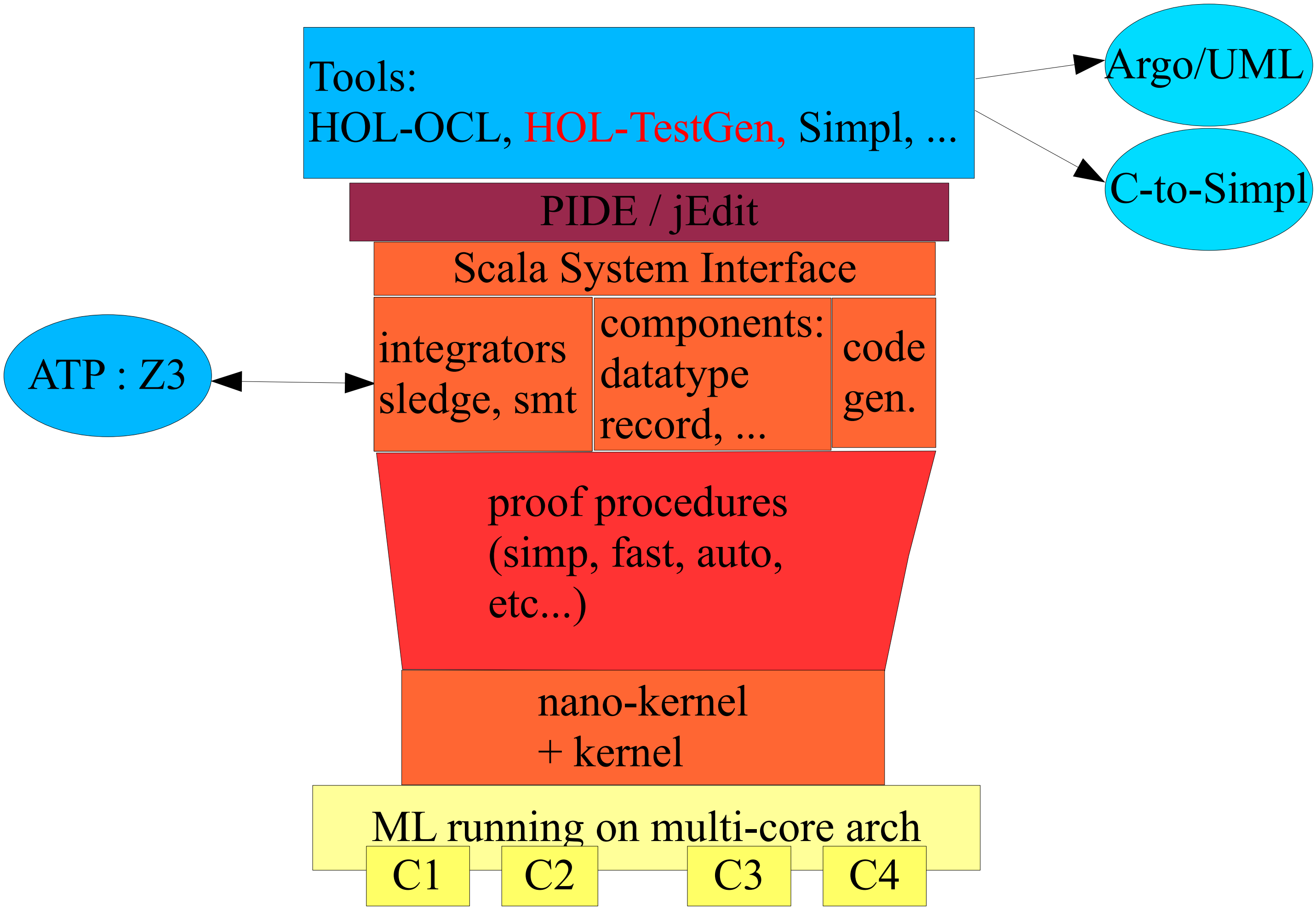
Overview

- HOL-TestGen and its Business-Case
- The Standard Workflow for Unit Testing
- Demo
- The Workflow for Sequence Tests

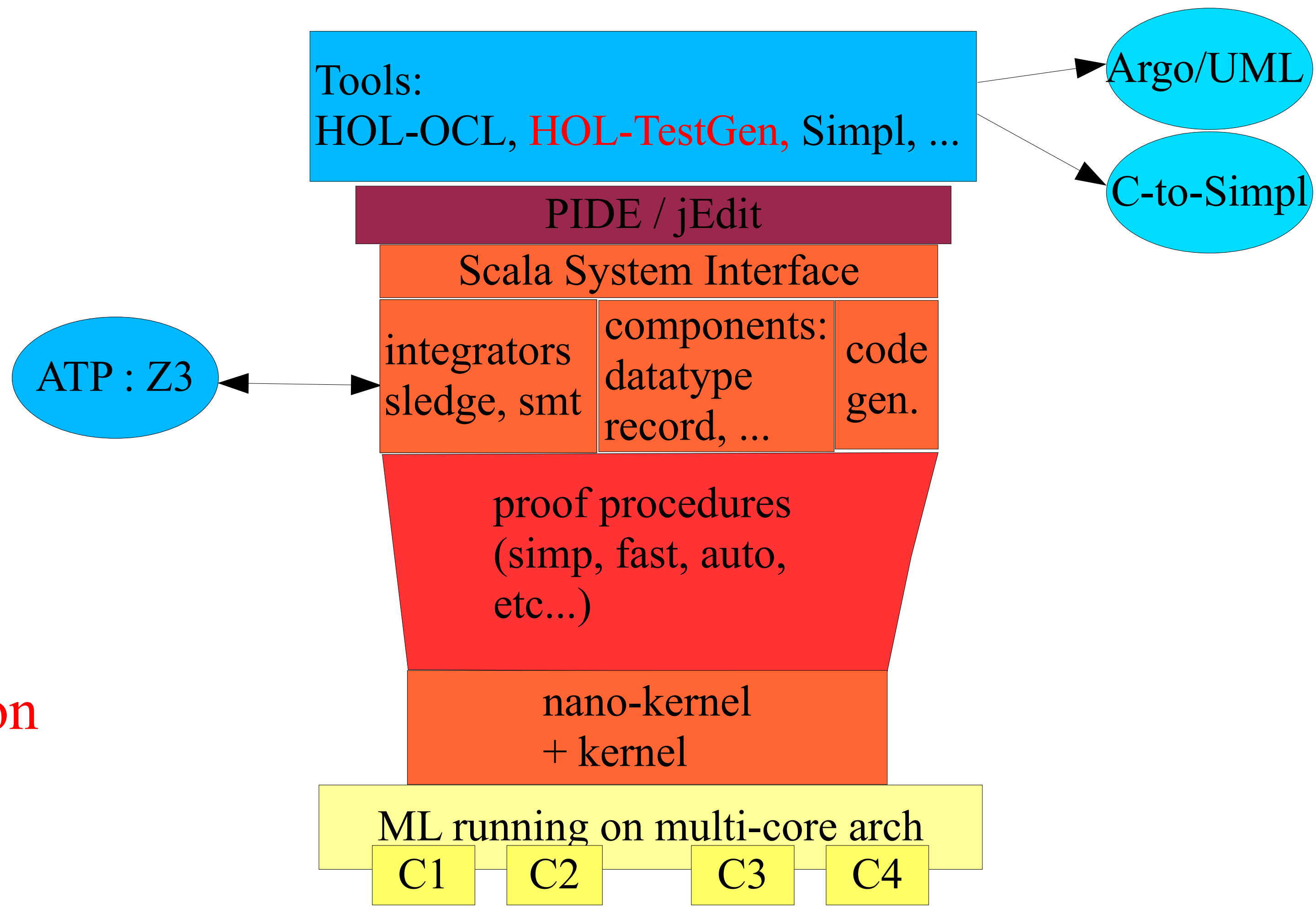
HOL-TestGen and its Business-Case

- HOL-TestGen is somewhat unusual test-Tool:
  - implemented as "PlugIn" in a major Interactive Theorem Proving Environment : Isabelle/HOL
  - conceived as formal testcase-generation method based on symbolic execution of a model (in HOL)
  - Favors Expressivity and emphasizes Test-Plans as formal entities; emphasis on Interactivity

HOL-TestGen as Plugin in the Isabelle Architecture



HOL-TestGen as Plugin in the Isabelle Architecture



<u>Advantage</u>:

- Reuse of powerful components in unique, interactive integrated environment

- seamless integration of test and proof activities

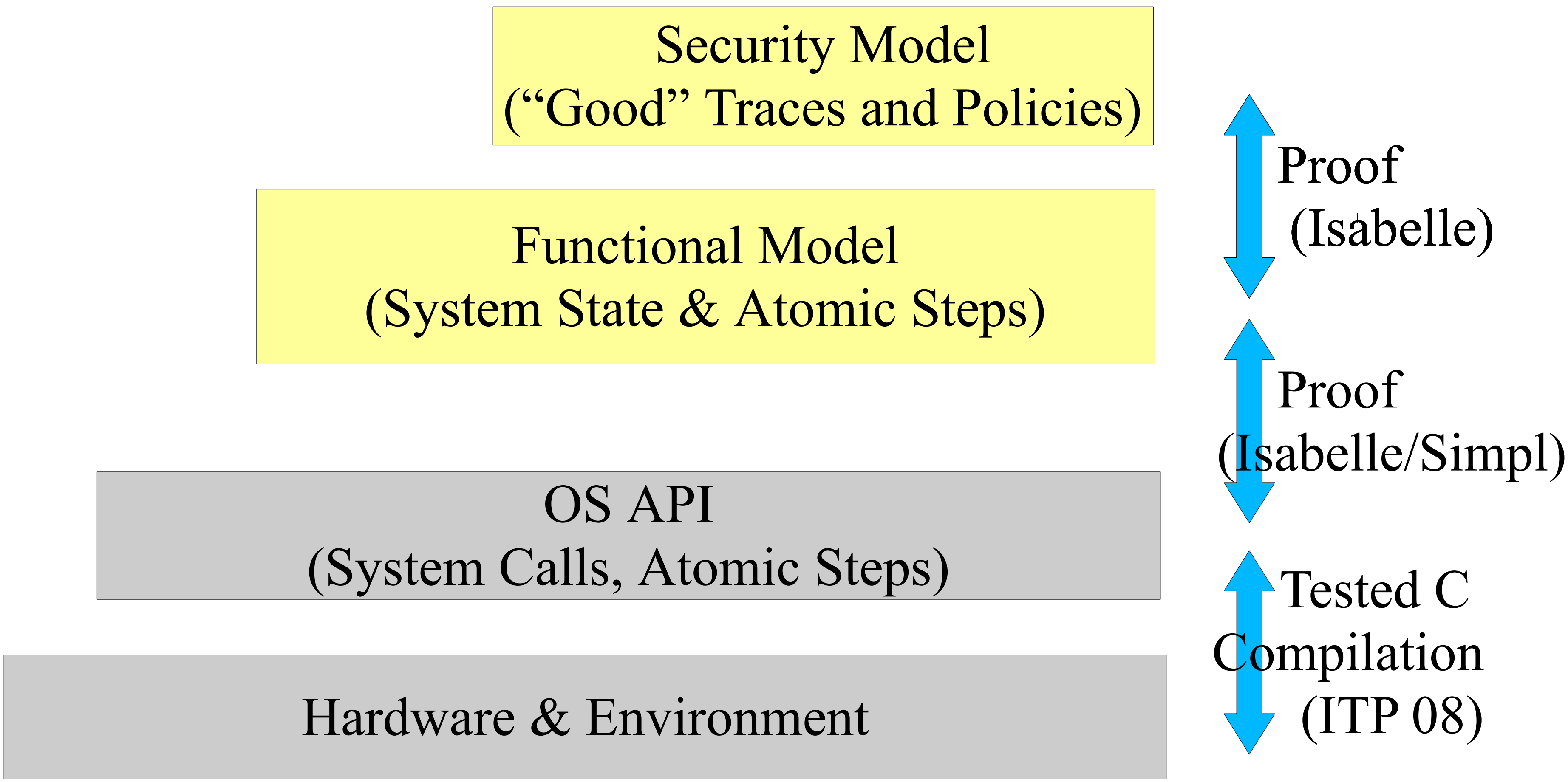
HOL-TestGen's Business Case

- If you have already a system model in Isabelle or Coq, you might want to link it to a real implementation.

- This has particular relevance in a Certification project (for example: Common Criteria EAL 5 - 7

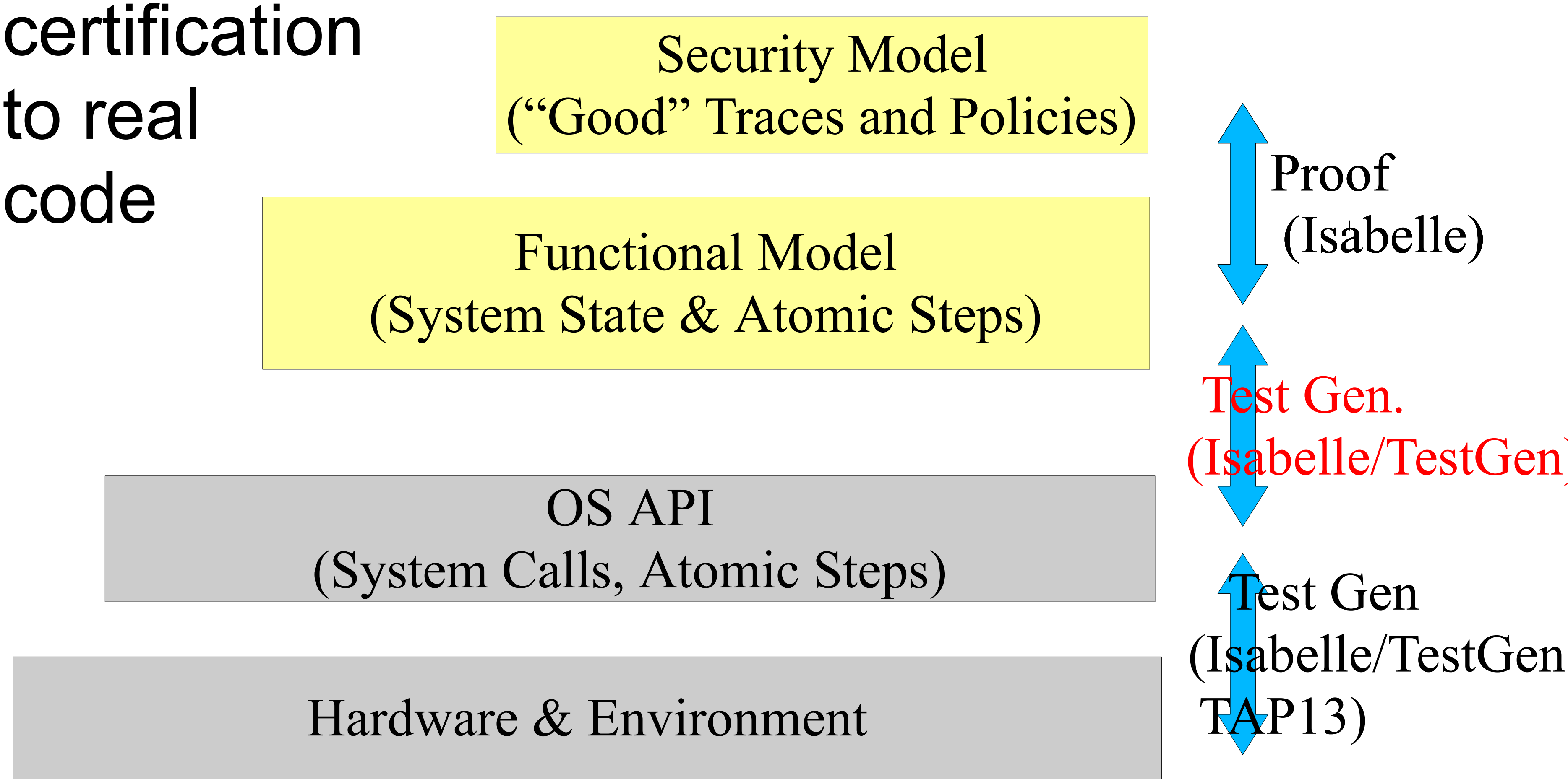
HOL-TestGen's Business Case

- NICTA seL4 Verified Project



EUROMILS PikeOS Project

- CC EAL 5+; Linking the FM model of the certification to real code



Look and Feel : The PIDE Interface for Theories

header

command

```
theory AVL_def
imports Testing Main
begin
```

```
datatype 'a tree = ET |  MKT 'a "'a tree" "'a tree"
```

```
fun height :: "'a tree ⇒ nat"
where
  "height ET = 0"
| "height (MKT n l r) = 1 + max (height l) (height r)"
```

```
fun is_in  :: " 'a ⇒ 'a tree ⇒ bool"
where
  "is_in k ET = False"
| "is_in k (MKT n l r) = (k=n ∨ is_in k l ∨ is_in k r)"
```

Look and Feel : The PIDE Interface for Theories

header →

```
theory AVL_def
imports Testing Main
begin
```

command →

```
datatype 'a tree = ET |  MKT 'a "'a tree" "'a tree"
```

→

```
fun height :: "'a tree ⇒ nat"
where
  "height ET = 0"
| "height (MKT n l r) = 1 + max (height l) (height r)"
```

→

```
fun is_in  :: "'a ⇒ 'a tree ⇒ bool"
where
  "is_in k ET = False"
| "is_in k (MKT n l r) = (k=n ∨ is_in k l ∨ is_in k r)"
```

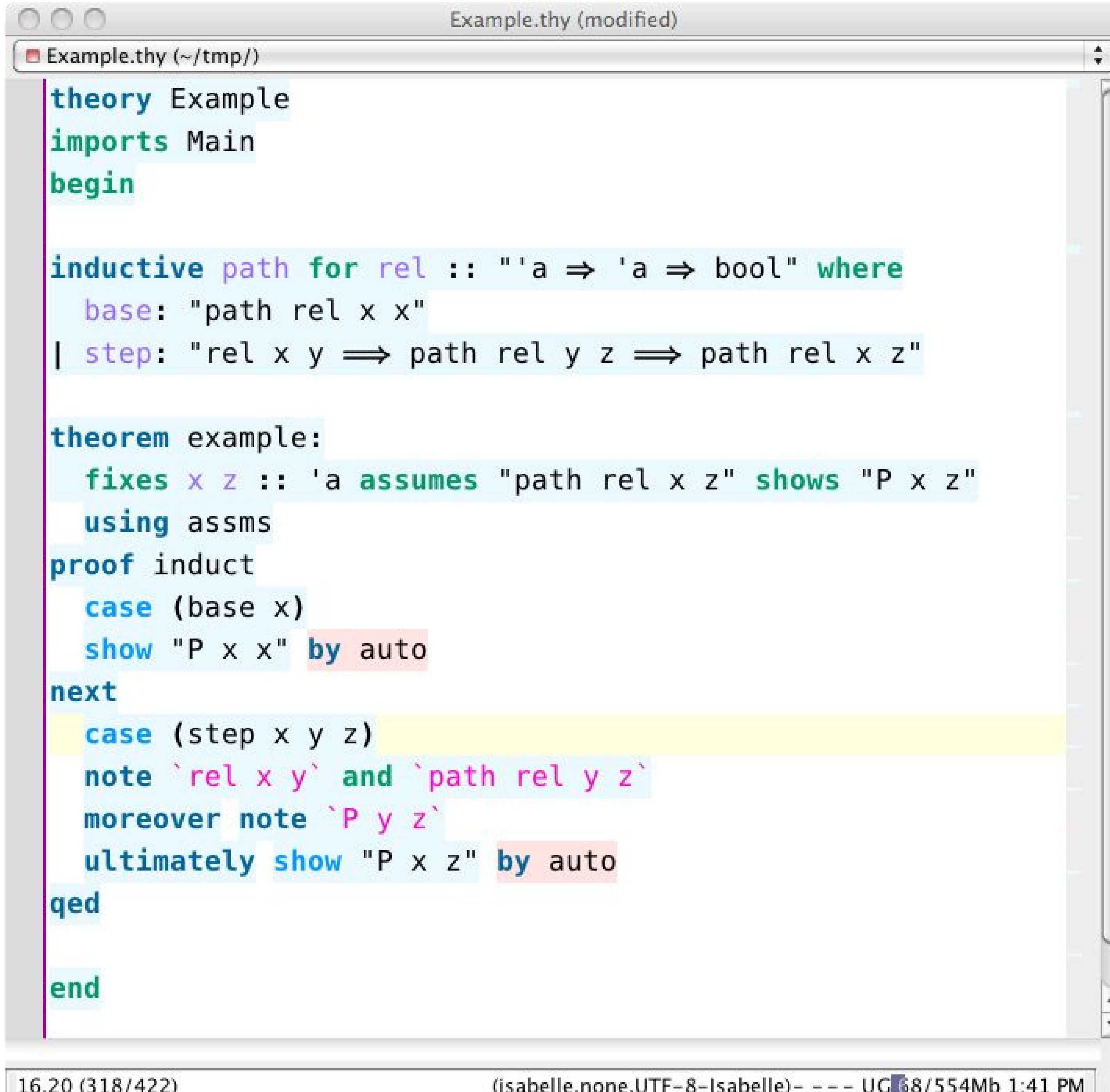
Models look pretty much like SML or OCaml programs ...

Parallel,
Asynchronous
execution and
validation
in the

jEdit - GUI

IDE look and feel;
Very attractive
work environment.

(Better than Eclipse ;-)



HOL-TestGen Workflow

- Modelisation
  - writing background theory of problem domain

Black-Box Testing: "The Standard Workflow"

- Writing a test-theory (the "model")

Black-Box Testing: “The Standard Workflow”

- Writing a test-theory (the “model”)

  Example: Sorting in HOL

```
primrec  is_sorted ::”int list ⇒ bool”
where    “is_sorted [] = True”
       | “is_sorted (x#xs) =
              case xs of
                 []  ⇒ True
               | (y#ys) ⇒ (x≤y) ∧ is_sorted ys”
```

Black-Box Testing: "The Standard Workflow"

- Writing a test-theory (the "model")

  Example: Sorting in HOL

```
primrec  is_sorted ::"int list ⇒ bool"
where    "is_sorted [] = True"
        | "is_sorted (x#xs) =
                case xs of
                   []  ⇒ True
                 | (y#ys) ⇒ (x≤y) ∧ is_sorted ys"
```

Black-Box Testing: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS

Black-Box Testing: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS

testspec “ is_sorted($PUT\ x$)
$\wedge$ asc(x, $PUT\ x$)”

Black-Box Testing: "The Standard Workflow"

- Writing a test-theory
- Writing a test-specification TS

pattern:

testspec "pre $x \longrightarrow$ post x $(PUT\, x)$"

Black-Box Testing: "The Standard Workflow"

- Writing a test-theory
- Writing a test-specification TS

example:

test_spec "is_sorted x → is_sorted ($prog\ a\ x$)"

or

test_spec "is_sorted ($PUT\ l$)"

Black-Box Testing: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem
  (“Testcase Generation”)

Black-Box Testing: “The Standard Workflow“

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem

("Testcase Generation")

apply(gen_test_cases 3 1 “*PUT*”)

Black-Box Testing: "The Standard Workflow"

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem ("Testcase Generation")

$$TC_1 \Rightarrow \ldots \Rightarrow TC_n \Rightarrow THYP(H_1) \Rightarrow \ldots \Rightarrow THYP(H_m) \Rightarrow TS$$

  - where testcases TC_i have the form

$$Constraint_1(x) \Rightarrow \ldots \Rightarrow Constraint_k(x) \Rightarrow P(\mathit{prog}\ x)$$

  - and where $THYP(H_i)$ are test-hypothesis

Black-Box Testing: "The Standard Workflow"

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem

Example:

is_sorted (PUT l)

1: is_sorted(PUT [])
2: is_sorted(PUT [?X])
3: THYP(∃ x. is_sorted(PUT [x]) → ∀ x. is_sorted(PUT [x]))
4: is_sorted(PUT [?X, ?Y])

Black-Box Testing: "The Standard Workflow"

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem

...

5: THYP(∃ x y. is_sorted(PUT[x,y]) →
∀ x y. is_sorted(PUT[x,y]))

6: is_sorted(PUT [?X, ?Y, ?X])

7: THYP(∃ x y z. is_sorted(PUT [x,y,z]) →
∀ x y z. is_sorted(PUT [x,y,z]))

8: THYP(3 < |l| → is_sorted(PUT l))

Black-Box Testing: "The Standard Workflow"

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem
- Generation of test-data

Black-Box Testing: "The Standard Workflow"

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem
- Generation of test-data

gen_test_data "..."

Black-Box Testing: "The Standard Workflow"

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem
- Generation of test-data

```
is_sorted(PUT 1 [])
is_sorted(PUT 1 [0])
is_sorted(PUT 1 [2])
is_sorted(PUT 1 [1,2])
```

Black-Box Testing: "The Standard Workflow"

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem
- Generation of test-data
- Generating a test-harness

Black-Box Testing: “The Standard Workflow”

- Writing a test-theory
- Writing a test-specification TS
- Conversion into test-theorem
- Generation of test-data
- Generating a test-harness
- Run of testharness and generation of test-document

Midi Example: Red Black Trees

Red-Black-Trees: Test Specification

```
testspec :
(redinv t  ∧
 blackinv t)

 →

      (redinv (delete x t)  ∧
       blackinv (delete x t))
```

where `delete` is the program under test.

HOL-TestGen Workflow

Demo

Black-Box Sequence Testing:

- HOL is a state-less language;

  how to model and test stateful systems ?

- How to test systems where you have only control over the initial state ?

- How to test concurrent programs implementing a model ?

How to model and test stateful systems ?

- Use Monads !!!
  - The transition in an automaton $(\sigma,(\iota, o),\sigma)$set can isomorphically represented by:

$$\iota \Rightarrow \sigma \Rightarrow (o,\sigma) \text{ set}$$

or for a deterministic transition function:

$$\iota \Rightarrow \sigma \Rightarrow (o,\sigma) \text{ option}$$

… which category theorists or functional programmers would recognize as a Monad function space

How to model and test stateful systems ?

- Use Monads !!!
  - The transition in an automaton $(\sigma,(\iota, o),\sigma)$set can isomorphically represented by:

$$\iota \Rightarrow (o \times \sigma)\ \mathrm{Mon}_{SBE}$$

or for a deterministic transition function:

$$\iota \Rightarrow (o \times \sigma)\ \mathrm{Mon}_{SE}$$

… which category theorists or functional programmers would recognize as a Monad function space

How to model and test stateful systems ?

- Monads must have two combination operations bind and unit enjoying three algebraic laws.
  - For the concrete case of Mon_{SE}:

```
definition bind_SE :: "('o,'σ)MON_SE ⇒('o ⇒('o','σ)MON_SE) ⇒('o','σ)MON_SE"
where      "bind_SE f g = (λσ. case f σ of None ⇒None
                                  | Some (out, σ') ⇒g out σ')"

definition unit_SE :: "'o ⇒('o, 'σ)MON_SE" ("(return _)" 8)
where      "unit_SE e = (λσ. Some(e,σ))"
```

– and write o←m; m' o for $bind_{SE}$ m (λo. m' o)
and return for $unit_{SE}$

How to model and test stateful systems ?

- Valid Test Sequences:

$$\sigma \models o_1 \leftarrow m_1\, \iota_1; \ldots ; o_n \leftarrow m_n\, \iota_n; \mathrm{return}(P\, o_1 \cdots o_n)$$

- … can be generated to code
- … can be symbolically executed …

$$\frac{}{(\sigma \models \mathrm{return}\, P) = P}$$

$$\frac{C_m\, \iota\, \sigma \qquad m\, \iota\, \sigma = None}{(\sigma \models ((s \leftarrow m\, \iota; m'\ s))) = False}$$

$$\frac{C_m\, \iota\, \sigma \qquad m\, \iota\, \sigma = Some(b, \sigma')}{(\sigma \models s \leftarrow m\, \iota; m'\ s) = (\sigma' \leftarrow (m'\ b))}$$

How to model and test stateful systems ?

- Test Refinements for a step-function SPEC and a step function SUT:

$$\sigma \models o_1 \leftarrow \mathrm{SPEC}_1\ \iota_1; \ldots ; o_n \leftarrow \mathrm{SPEC}_n\ \iota_n; \mathrm{return}(res = [o_1 \cdots o_n])$$

$$\longrightarrow$$

$$\sigma \models o_1 \leftarrow \mathrm{SUT}_1\ \iota_1; \ldots ; o_n \leftarrow \mathrm{SUT}_n\ \iota_n; \mathrm{return}(res = [o_1 \cdots o_n])$$

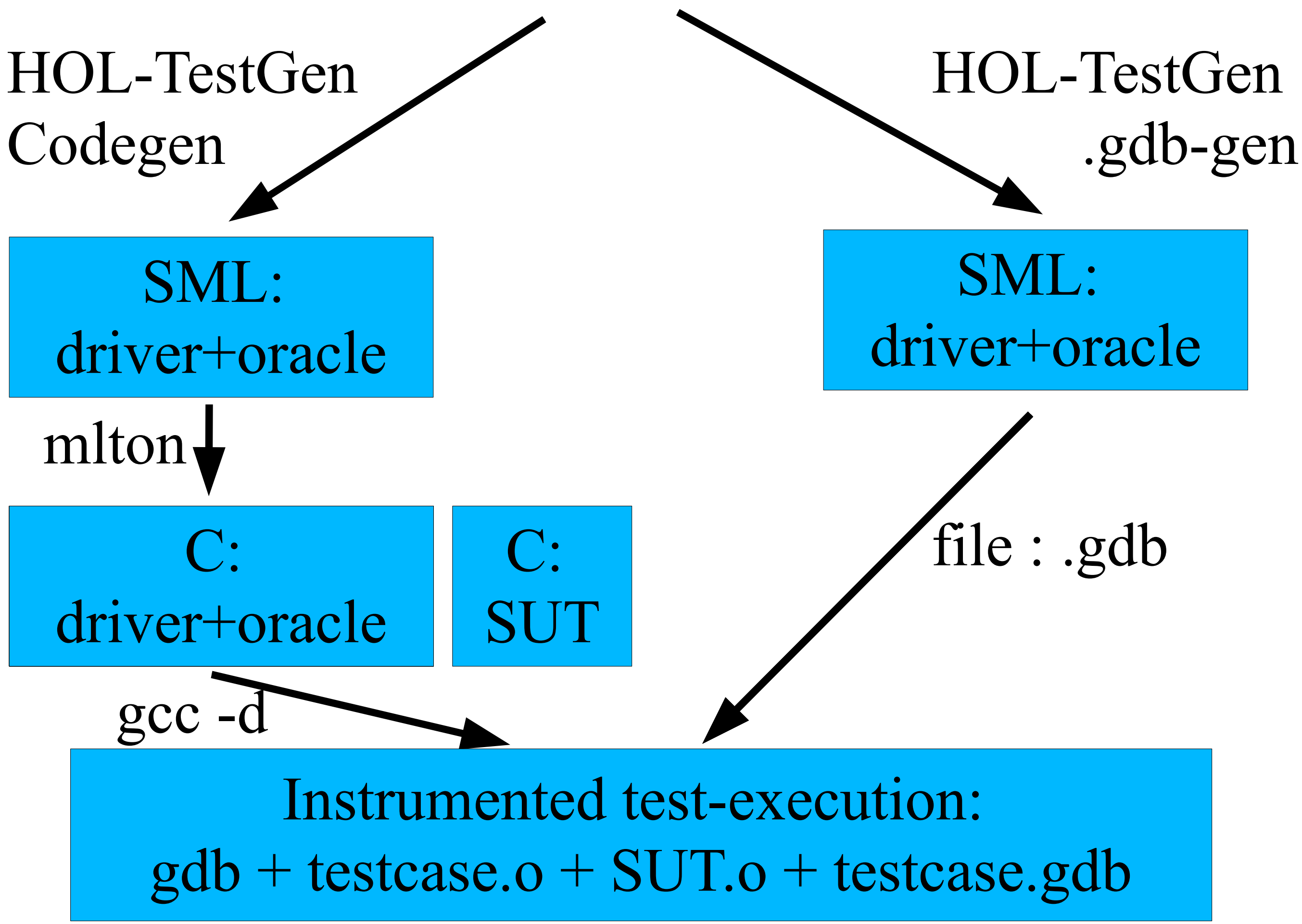
- The premis is reduced by symbolic execution to constraints over *res*; a constraint solver (Z3) produces an instance for *res*. The conclusion is compiled to a test-driver/test-oracle linked to *SUT.*

How to test concurrent programs implementing a model ?

- Assumption: Code compiled for LINUX and instrumented for debugging (gcc -d)
- Assumption: No dynamic thread creation (realistic for PikeOS); identifiable atomic actions in the code;
- Assumption: Mapping from abstract atomic actions in the model to code-positions known.
- Abstract execution sequences were generated to .gdb scripts forcing explicit thread-switches of the SUT executed under gdb.

How to test concurrent programs implementing a model ?

$$\sigma \models o_1 \leftarrow \mathrm{SUT}_1\ \iota_1; \ldots ; o_n \leftarrow \mathrm{SUT}_n\ \iota_n; \mathrm{return}(res = [o_1 \cdots o_n])$$



Conclusion

- HOL-TestGen is an Advanced Model-based Testing Environment built on top of Isabelle/HOL
- Allows to establish a Link between a formal System Model in Isabelle/HOL and Real Code by (semi)-automated generation of tests.
- Smooth Integration of Test and Proof !