Can Testing Be Liberated From The Automata Style ???

Towards a Monadic Approach of Symbolic Behavioral Test-Generation

Prof. Burkhart Wolff
Univ - Paris-Sud / LRI

Abstract

- Sequence Testing is an important sub-domain of formal model-based Testing. It addresses test scenarios where the tester controls the state of the System Under Test (SUT) only at the initialization time and then indirectly via a sequence of inputs. The latter may stimulate observable outputs on which the test-verdict must be based solely.

- A number of automata-based test-theories have been suggested that work fairly well for traces of impressing length — provided that the state space of the SUT is small. Whenever large state spaces have to be modeled — as is the case for operating systems, data-bases or web-services — both theory and implementations resists obstinately practical applicability: Theoretically, because symbolic representations of state spaces have to be treated; Practically, because these difficulties result in a small number of tools addressing sparse and fairly limited application domains.

- In this talk, I will present a novel approach to the problem based on Monads, their theory developed in Isabelle/HOL. Notions like Test-Sequence and Test-Refinement can be rephrased in terms of Monads, which opens the way both for efficient symbolic execution of system models as well as the efficient compilation to test-drivers. Theoretically, the monadic approach allows to
  1.) resists the tendency to surrender to finitism and constructivism at the first-best opportunity
  2.) provides a sensible shift from syntax to semantics: instead of a first-order, intentional view in nodes and events in automata, the heart of the calculus is on computations and their compositions

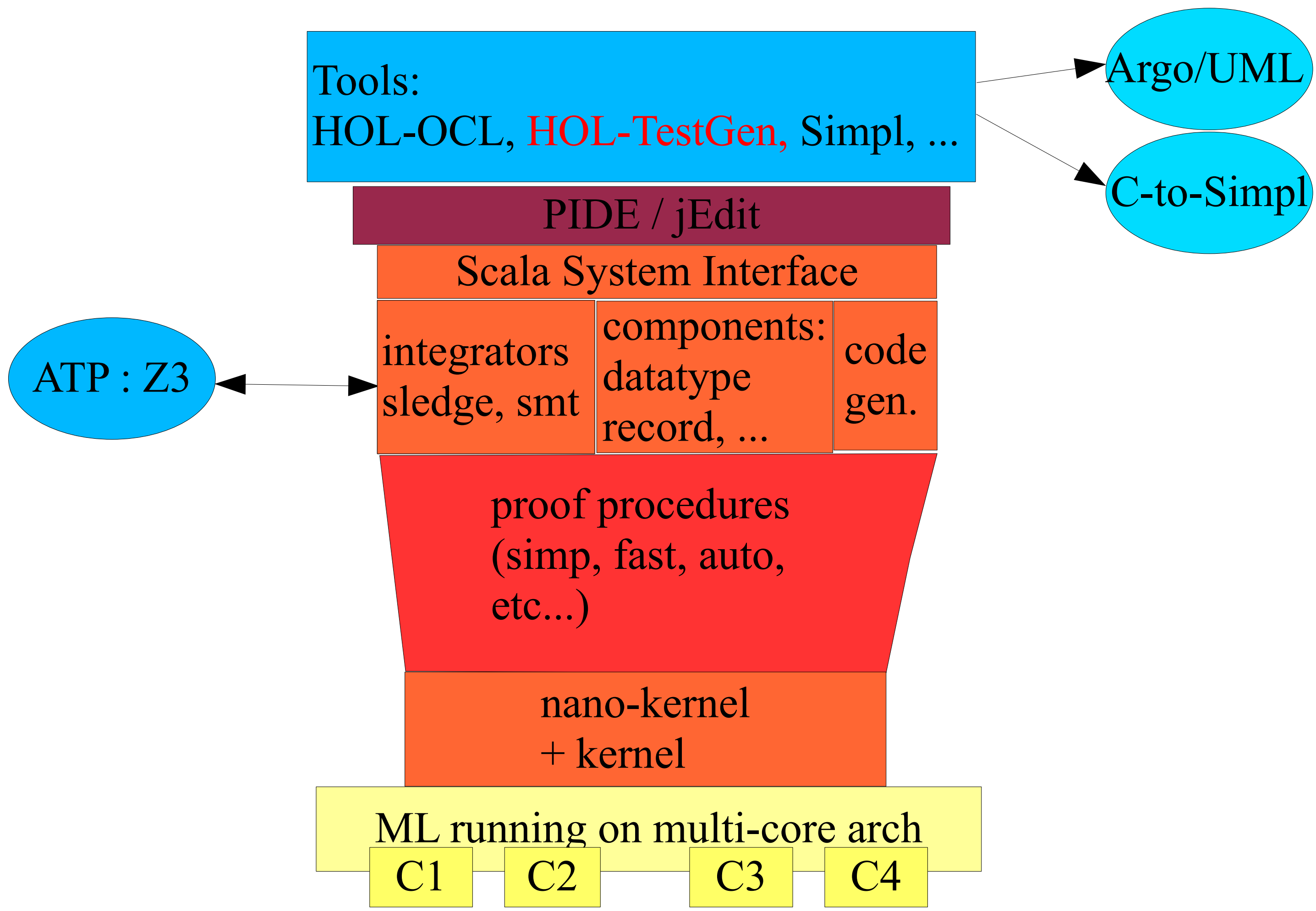
Overview

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

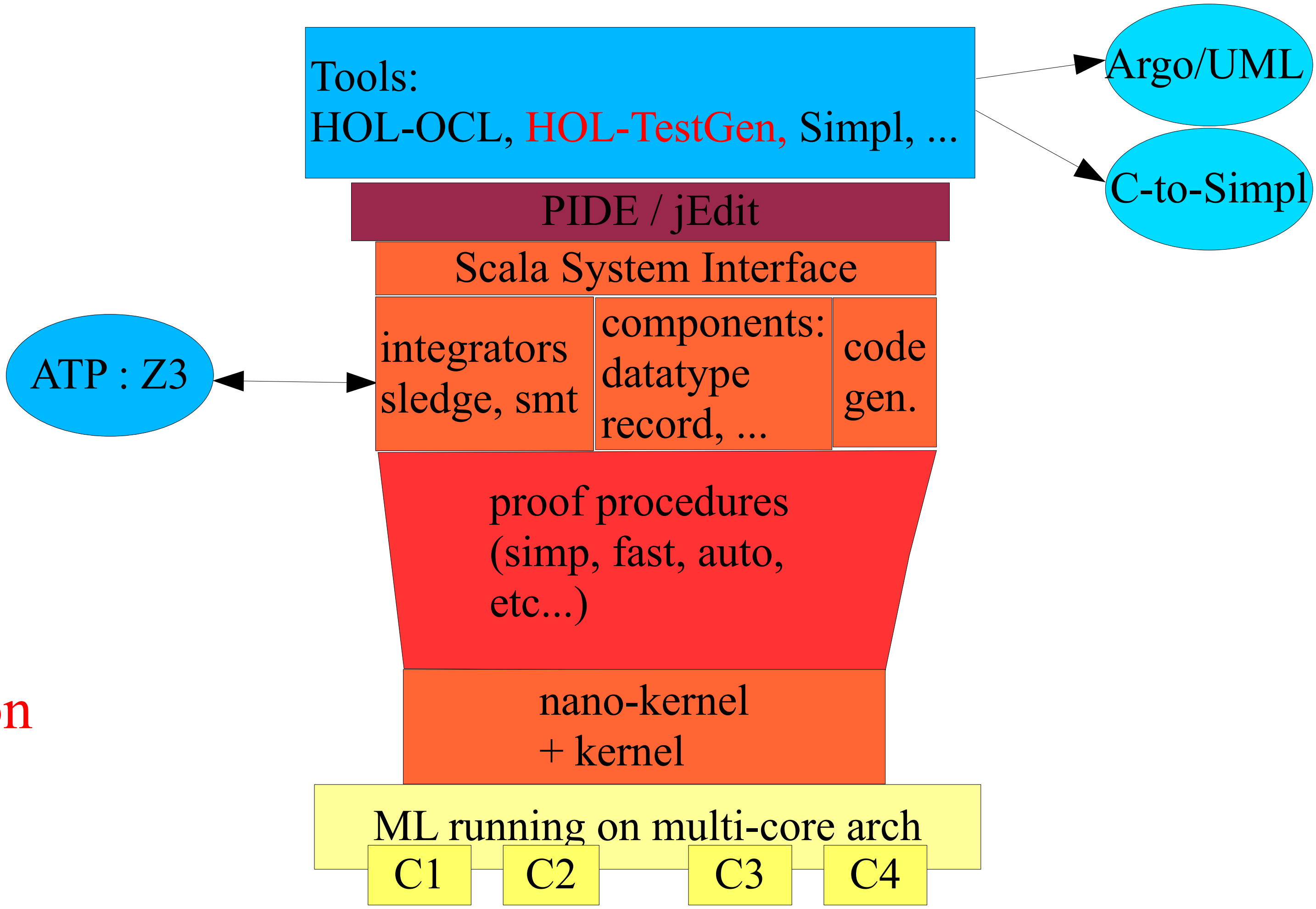
HOL-TestGen and its Business-Case

- 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
  - Document-centric test-development (inspired by SPECEXPLORER)

HOL-TestGen as Plugin in the Isabelle Architecture



HOL-TestGen as Plugin in the Isabelle Architecture



Advantage:

- Reuse of powerful components in unique, interactive integrated environment
- seamless integration of test and proof activities

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 $\rightarrow$ is_sorted ($PUT\ a\ x$)"

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

Introduction to 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 ?

Introduction to Sequence Testing

Testability Hypothesis in Sequence Testing

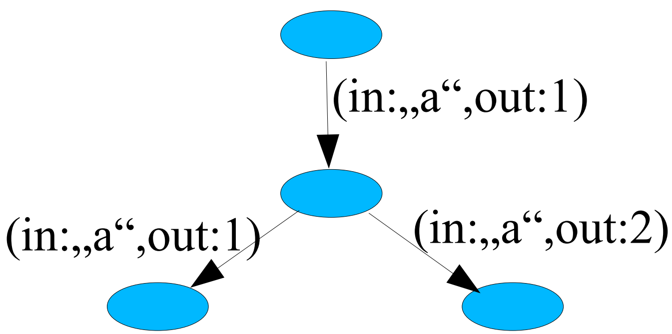
1. The tester can reset the system under test (the SUT) into a known initial state,
2. the tester can stimulate the SUT only via the operation-calls and input of a known interface; while the internal state of the SUT is hidden to the tester, the SUT is assumed to be only controlled by these stimuli, and
3. the SUT behaves deterministic with respect to an observed sequence of input-output pairs (it is input-output deterministic).

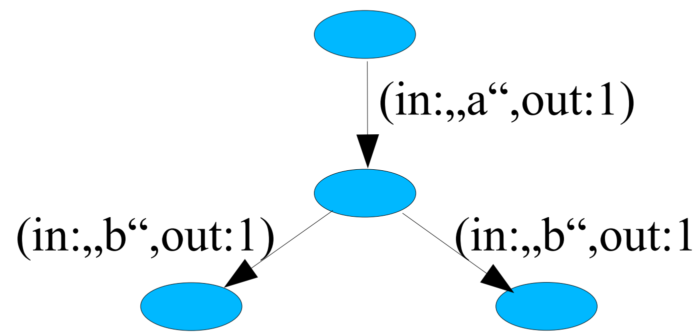
Introduction to Sequence Testing

- Some notions of traditional sequence testing
  - Input-Output Automata, e.g. $A = (\sigma, \tau::(\sigma \times (\iota \times o) \times \sigma)set)$,
    - σ is the type of states
    - ι the type of inputs (input events)
    - o the type of outputs (output events)
    - τ the set of input-output-transitions.

Introduction to Sequence Testing

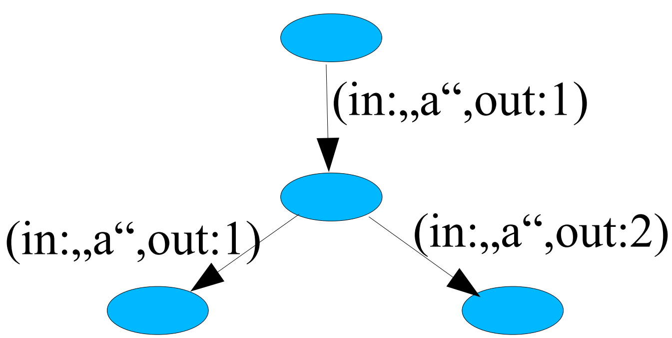
- Some notions of traditional sequence testing
  - Input-Output Automata, e.g. $A = (\sigma, \tau::(\sigma,(\iota, o),\sigma)set)$,

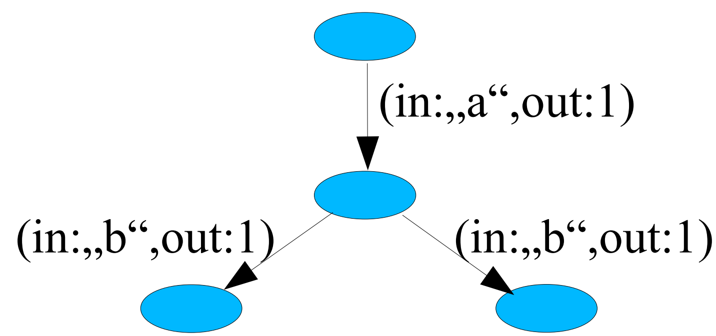




Introduction to Sequence Testing

- Some notions of traditional sequence testing
  - Input-Output Automata, e.g. $A = (\sigma, \tau::(\sigma,(\iota, o),\sigma)set)$,

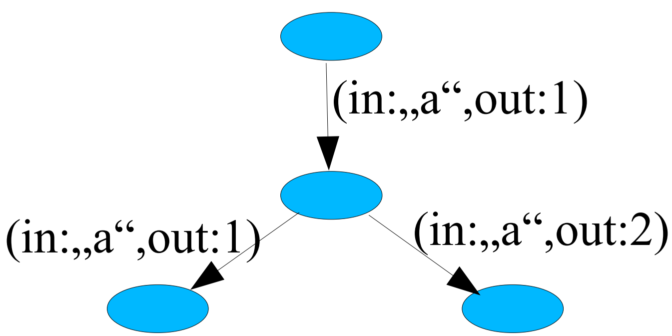


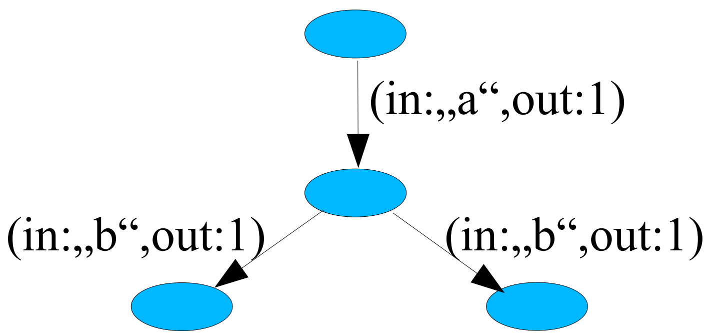


  - $t \in Trace(A) :: (\iota, o)set$ (eg. [(“a”,1)(“a”,1)])
  - set of enabled inputs after a trace:
    $In_A(t)$ (eg. In_A([(“a”,1)]) ={“a”})
  - set of possible outputs after trace and input:
    $Out_A(t,i)$ (eg. Out_A([(“a”,1)]) ={1,2})

Introduction to Sequence Testing

- Some notions of traditional sequence testing
  - Input-Output Automata, e.g. $A = (\sigma, \tau::(\sigma,(\iota, o),\sigma)set)$,





Note: IO-Determinism does NOT mean that a system is "deterministic"

Introduction to Sequence Testing

- Some notions of traditional sequence testing
  - Conformance Relations assume that a model (an Automata) is refined by an implementation (assumed to be an automata)

    $$SPEC \sqsubseteq IMPL$$

  - Well-known notions are:
    - inclusion conformance[5]: all traces in SPEC must be possible in SUT,
    - deadlock conformance[7]: for all traces $t \in Traces(SPEC)$ and $b \in In(t)$, b must be refused by SUT, and
    - input/output conformance (IOCO)[19]: for all traces $t \in Traces(SPEC)$ and all $\iota \in In(t)$, the observed output of SUT must be in $Out(t, \iota)$.

How to model and test stateful systems ?

- Use Monads !!!
  - The transition in an automaton (σ,(ι, o),σ)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:

$$\iota \Rightarrow (o \times \sigma)\ Mon_{SBE}$$

or for a deterministic transition function:

$$\iota \Rightarrow (o \times \sigma)\ Mon_{SE}$$

… 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}:

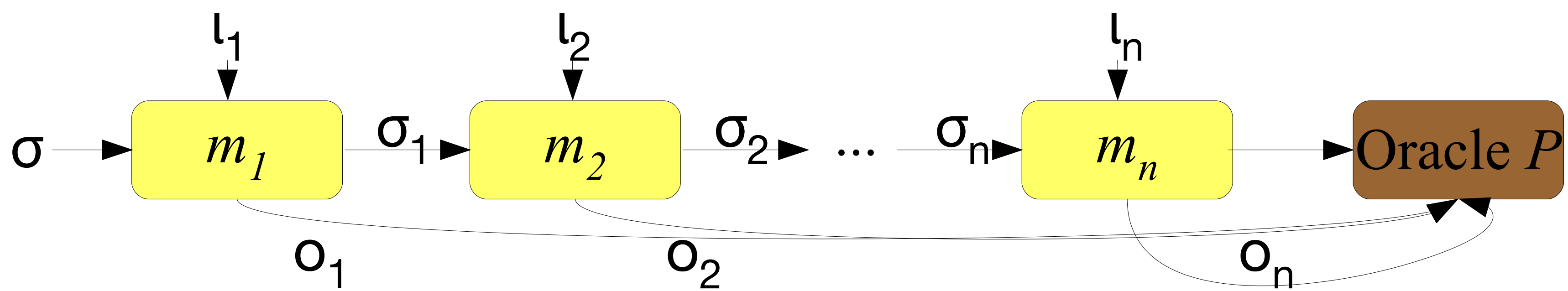
```
definition bind_SE :: "('o,'σ)MON_SE ⇒('o ⇒('o','σ)MON_SE) ⇒('o','σ)MON_SE"
where      "bind_SE f g = (λσ. case f σ of None ⇒None
                                 | Some (out, σ') ⇒g out σ')"

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:



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

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

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

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

How to model and test stateful systems ?

- 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 …

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

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

Example : MyKeOS ?

- We consider an (brutal) abstraction of an L4 Kernel IPC protocol called "MyKeOS"
- It has
  - unbounded number of tasks
  - … having an unbounded number of threads
  - … which each have a counter for a resource
  - … the atomic actions alloc, release, status (tagged by task-id, thread-id, arguments)
  - release can only release allocated ressources

Example : MyKeOS ?

- State :

$$(\text{task\_id} \times \text{thread\_id}) \rightharpoonup \text{int}$$

- Input events:

$$\begin{aligned} in_{event} = {} & \text{alloc} \quad \text{task\_id thread\_id nat} \\ & \mid \text{release task\_id thread\_id nat} \\ & \mid \text{status} \quad \text{task\_id thread\_id} \end{aligned}$$

- Output events:

$$out_{event} = \text{alloc\_ok} \mid \text{release\_ok} \mid \text{status\_ok nat}$$

- System Model SYS: interprets input event in a state and yields an output event and a successor state if successful, an exception otherwise.

Example : MyKeOS (0)

$$\sigma_0 \models s \leftarrow \text{mbind [alloc tid 1 m'',} \\ \text{release tid 0 m',} \\ \text{release tid 1 m''',} \\ \text{status tid 1] SYS;} \\ \text{unit(x = s)}$$

Example : MyKeOS (0)

$\sigma_0 \models s \leftarrow$ mbind [alloc tid 1 m'',
release tid 0 m',
release tid 1 m''',
status tid 1] SYS;
unit(x = s)

Example : MyKeOS (1)

$(tid, 1) \in dom\ \sigma_0 \Longrightarrow$

$\sigma'_0 = \sigma_0((tid, 1) \mapsto the\ (\sigma_0\ (tid, 1)) + int\ m'') \Longrightarrow$

$\sigma'_0 \models s \leftarrow mbind\ [release\ tid\ 0\ m',$
$release\ tid\ 1\ m''',$
$status\ tid\ 1]\ SYS;$
$unit(x = alloc\_ok\ \#\ s)$

Example : MyKeOS (2)

$(tid, 1) \in dom\ \sigma_0 \Longrightarrow$

$\sigma'_0 = \sigma_0((tid, 1) \mapsto the\ (\sigma_0\ (tid, 1)) + int\ m'') \Longrightarrow$

$\sigma'_0 \models s \leftarrow mbind$ [release tid 0 m',
release tid 1 m''',
status tid 1] SYS;
unit(x = alloc_ok # s)

Example : MyKeOS (2)

$(tid, 1) \in \text{dom}\ \sigma_0 \Longrightarrow$

$\sigma'_0 = \sigma_0((tid, 1) \mapsto \text{the}\ (\sigma_0\ (tid, 1)) + \text{int}\ m'') \Longrightarrow$

$\text{int}\ m' \leq \text{the}\ ((\sigma_0((tid,1) \mapsto \text{the}(\sigma_0(tid,1)) + \text{int}\ m''))(tid,0)) \Longrightarrow$

$\sigma''_0 = \sigma'((tid, 0) \mapsto \text{the}\ (\sigma'(tid, 0)) - \text{int}\ m') \Longrightarrow$

$\sigma''_0 \models s \leftarrow \text{mbind}\ [\text{release tid 1 m'''},$
$\text{status tid 1}]\ \text{SYS};$
$\text{unit}(x = \text{alloc\_ok}\ \#\ \text{release\_ok}\ \#\ s)$

Example : MyKeOS (3)

$(tid, 1) \in \text{dom}\ \sigma_0 \Longrightarrow$

$\sigma'_0 = \sigma_0((tid, 1) \mapsto \text{the}\ (\sigma_0\ (tid, 1)) + \text{int}\ m'') \Longrightarrow$

$\text{int}\ m' \leq \text{the}\ ((\sigma_0((tid,1) \mapsto \text{the}(\sigma_0(tid,1)) + \text{int}\ m''))(tid,0)) \Longrightarrow$

$\sigma''_0 = \sigma'((tid, 0) \mapsto \text{the}\ (\sigma'(tid, 0)) - \text{int}\ m') \Longrightarrow$

$\sigma''_0 \models s \leftarrow$ mbind [release tid 1 m''',
status tid 1] SYS;
unit(x = alloc_ok # release_ok # s)

Example : MyKeOS (3)

$(tid, 1) \in dom\ \sigma_0 \Longrightarrow$

$\sigma'_0 = \sigma_0((tid, 1) \mapsto the\ (\sigma_0\ (tid, 1)) + int\ m'') \Longrightarrow$

$int\ m' \leq the\ ((\sigma_0((tid,1) \mapsto the(\sigma_0(tid,1))+int\ m''))(tid,0)) \Longrightarrow$

$\sigma''_0 = \sigma'((tid, 0) \mapsto the\ (\sigma'(tid, 0))\ -\ int\ m') \Longrightarrow$

$\ldots \Longrightarrow \ldots \Longrightarrow$

$\sigma'''_0 \models s \leftarrow mbind\ [status\ tid\ 1]\ SYS;$

$\quad unit(x = alloc\_ok\ \#\ release\_ok\ \#\ release\_ok\ \#\ s)$

Example : MyKeOS (4)

$(tid, 1) \in \mathrm{dom}\ \sigma_0 \Longrightarrow$

$\sigma'_0 = \sigma_0((tid, 1) \mapsto \mathrm{the}\ (\sigma_0\ (tid, 1)) + \mathrm{int}\ m'') \Longrightarrow$

$\mathrm{int}\ m' \leq \mathrm{the}\ ((\sigma_0((tid,1) \mapsto \mathrm{the}(\sigma_0(tid,1)) + \mathrm{int}\ m''))(tid,0)) \Longrightarrow$

$\sigma''_0 = \sigma'((tid, 0) \mapsto \mathrm{the}\ (\sigma'(tid, 0)) - \mathrm{int}\ m') \Longrightarrow$

$\ldots \Longrightarrow \ldots \Longrightarrow$

$\sigma'''_0 \models s \leftarrow$ mbind [status tid 1] SYS;
unit(x = alloc_ok # release_ok # release_ok # s)

Example : MyKeOS (5)

$(tid, 1) \in dom\ \sigma_0 \Longrightarrow$

$\sigma'_0 = \sigma_0((tid, 1) \mapsto the\ (\sigma_0\ (tid, 1)) + int\ m'') \Longrightarrow$

$int\ m' \leq the\ ((\sigma_0((tid,1) \mapsto the(\sigma_0(tid,1))+int\ m''))(tid,0)) \Longrightarrow$

$\sigma''_0 = \sigma'((tid, 0) \mapsto the\ (\sigma'(tid, 0)) - int\ m') \Longrightarrow$

$\ldots \Longrightarrow \ldots \Longrightarrow \ldots \Longrightarrow \ldots \Longrightarrow$

$\sigma'''_0 \models s \leftarrow mbind\ []\ SYS;$

$unit(x = alloc\_ok\ \#\ release\_ok\ \#\ release\_ok\ \#$
$status\_ok\ (the(\sigma'''_0\ (tid,1)))\ \#\ s)$

Example : MyKeOS (6)

$(tid, 1) \in dom\ \sigma_0 \Longrightarrow$

$\sigma'_0 = \sigma_0((tid, 1) \mapsto the\ (\sigma_0\ (tid, 1)) + int\ m'') \Longrightarrow$

$int\ m' \leq the\ ((\sigma_0((tid,1)\mapsto the(\sigma_0(tid,1))+int\ m''))(tid,0)) \Longrightarrow$

$\sigma''_0 = \sigma'((tid, 0) \mapsto the\ (\sigma'(tid, 0)) - int\ m') \Longrightarrow$

$\ldots \Longrightarrow \ldots \Longrightarrow \ldots \Longrightarrow \ldots \Longrightarrow$

$\sigma'''_0 \models s \leftarrow mbind\ []\ SYS;$

unit(x = alloc_ok # release_ok # release_ok # status_ok (the(σ'''_0 (tid,1))) # s)

Example : MyKeOS (6)

$(tid, 1) \in dom\ \sigma_0 \Longrightarrow$

$\sigma'_0 = \sigma_0((tid, 1) \mapsto the\ (\sigma_0\ (tid, 1)) + int\ m'') \Longrightarrow$

$int\ m' \leq the\ ((\sigma_0((tid,1) \mapsto the(\sigma_0(tid,1))+int\ m''))(tid,0)) \Longrightarrow$

$\sigma''_0 = \sigma'((tid, 0) \mapsto the\ (\sigma'(tid, 0)) - int\ m') \Longrightarrow$

$... \Longrightarrow ... \Longrightarrow ... \Longrightarrow ... \Longrightarrow$

$\sigma'''_0 \models unit(x = [alloc\_ok, release\_ok, release\_ok, status\_ok\ (the(\sigma'''_0\ (tid,1)))])$

Example : MyKeOS (7)

$(tid, 1) \in \mathrm{dom}\ \sigma_0 \Longrightarrow$

$\sigma'_0 = \sigma_0((tid, 1) \mapsto \mathrm{the}\ (\sigma_0\ (tid, 1)) + \mathrm{int}\ m'') \Longrightarrow$

$\mathrm{int}\ m' \leq \mathrm{the}\ ((\sigma_0((tid,1) \mapsto \mathrm{the}(\sigma_0(tid,1)) + \mathrm{int}\ m''))(tid,0)) \Longrightarrow$

$\sigma''_0 = \sigma'((tid, 0) \mapsto \mathrm{the}\ (\sigma'(tid, 0)) - \mathrm{int}\ m') \Longrightarrow$

$\ldots \Longrightarrow \ldots \Longrightarrow \ldots \Longrightarrow \ldots \Longrightarrow$

x = [alloc_ok, release_ok, release_ok, status_ok (the(σ'''_0 (tid,1)))])

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 premisse 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*.

Theory

- This motivates the notion of a "Generalized Monadic Test-Refinement"

$$(S \sqsubseteq_{\langle \Sigma_0, CC, conf \rangle} I) =$$

$$(\forall\ \sigma_0 \in \Sigma_0.\ \forall\ \iota s \in CC.\ \forall\ res.$$

$$(\sigma_0 \models (os \leftarrow mbind\ \iota s\ S;\ return\ (conf\ \iota s\ os\ res)))$$

$$\longrightarrow$$

$$(\sigma_0 \models (os \leftarrow mbind\ \iota s\ I;\ return\ (conf\ \iota s\ os\ res))))$$

Theory

- This motivates the notion of a "Generalized Monadic Test-Refinement"

  With conf set to:

  - (λ is os x. length is = length os ∧ os=x)

    ==> Inclusion Test
  - (λ is os x. length is > length os ∧ os=x)

    ==> Deadlock Refinement
  - (λ is os x. length is = length os ∧
    post_cond (last os) ∧ os=x)

    ==> IOCO Refinement (without quiescense)

Theory

- This motivates the notion of a
  "Generalized Monadic Test-Refinement"

  One can now PROVE equivalences between different members of the test-refinement families

  … and prove alternative forms for efficiency optimizations of the generated test-driver code.

Practice : How to test concurrent programs ?

- Assumption: Code compiled for LINUX and instrumented for debugging (gcc -d)
- Assumption: No dynamic thread creation (realistic for our target OS); 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.

Practice : How to test concurrent programs ?

```
thread IP4_send(tid_rec, thid_rec){
        if (defined(tid_rec) &&
            defined(thid_rec)) {
                ...
                grab_lock();
                atom: IPC_sendinit
                release_lock();
                ...
                if(curr_tid_hasRWin_tid_rec){
                        ...
                        grab_lock();
                        atom: IPC_prep
                        release_lock();
                        . . .
                        . . .
                }
                else{ return(ERROR_22);}
        }
        else{ return(ERROR_35);}
}
```

```
thread IP4_receive(tid_snd, thid_snd){
        if (defined(tid_snd) &&
            defined(thid_snd)) {
                ...
                grab_lock();
                atom: IPC_rec_rdy
                release_lock();
                ...
                if(curr_tid_hasRin_tid_rec)  {
                        ...
                        grab_lock();
                        atom: IPC_wait
                        release_lock();
                        . . .
                        . . .
                }
                else{ return(ERROR_59);}
        }
        else{ return(ERROR_21);}
}
```

Practice : How to test concurrent programs ?

```
thread IP4_send(tid_rec, thid_rec){
        if (defined(tid_rec) &&
           defined(thid_rec)) {
                ...
                grab_lock();
● "switch 2"    [atom: IPC_sendinit]
                release_lock();
                ...
                if(curr_tid_hasRWin_tid_rec){
                        ...
                        grab_lock();
                        [atom: IPC_prep]
                        release_lock();
                        . . .
                        . . .
                }
                else{ return(ERROR_22);}
        }
        else{ return(ERROR_35);}
}
```

```
thread IP4_receive(tid_snd, thid_snd){
● "switch 1"  if (defined(tid_snd) &&
           defined(thid_snd)) {
                ...
                grab_lock();
                [atom: IPC_rec_rdy]
● "switch 1"    release_lock();
                ...
                if(curr_tid_hasRin_tid_rec)  {
                        ...
                        grab_lock();
                        [atom: IPC_wait]
● "switch 1"            release_lock();
                        . . .
                        . . .
                }
                else{ return(ERROR_59);}
        }
        else{ return(ERROR_21);}
}
```

Practice : How to test concurrent programs ?

```
thread IP4_send(tid_rec, thid_rec){
      if (defined(tid_rec) &&
          defined(thid_rec)) {
              ...
              grab_lock();
              atom: IPC_sendinit
              release_lock();
              ...
              if(curr_tid_hasRWin_tid_rec){
                     ...
                     grab_lock();
                     atom: IPC_prep
                     release_lock();
                     . . .
                     . . .
              }
              else{ return(ERROR_22);}
      }
      else{ return(ERROR_35);}
}
```

"switch 2"

"switch 1"

"switch 1"

"switch 1"

```
thread IP4_receive(tid_snd, thid_snd){
      if (defined(tid_snd) &&
          defined(thid_snd)) {
              ...
              grab_lock();
              atom: IPC_rec_rdy
              release_lock();
              ...
              if(curr_tid_hasRin_tid_rec)  {
                     ...
                     grab_lock();
                     atom: IPC_wait
                     release_lock();
                     . . .
                     . . .
              }
              else{ return(ERROR_59);}
      }
      else{ return(ERROR_21);}
}
```

Practice : How to test concurrent programs ?

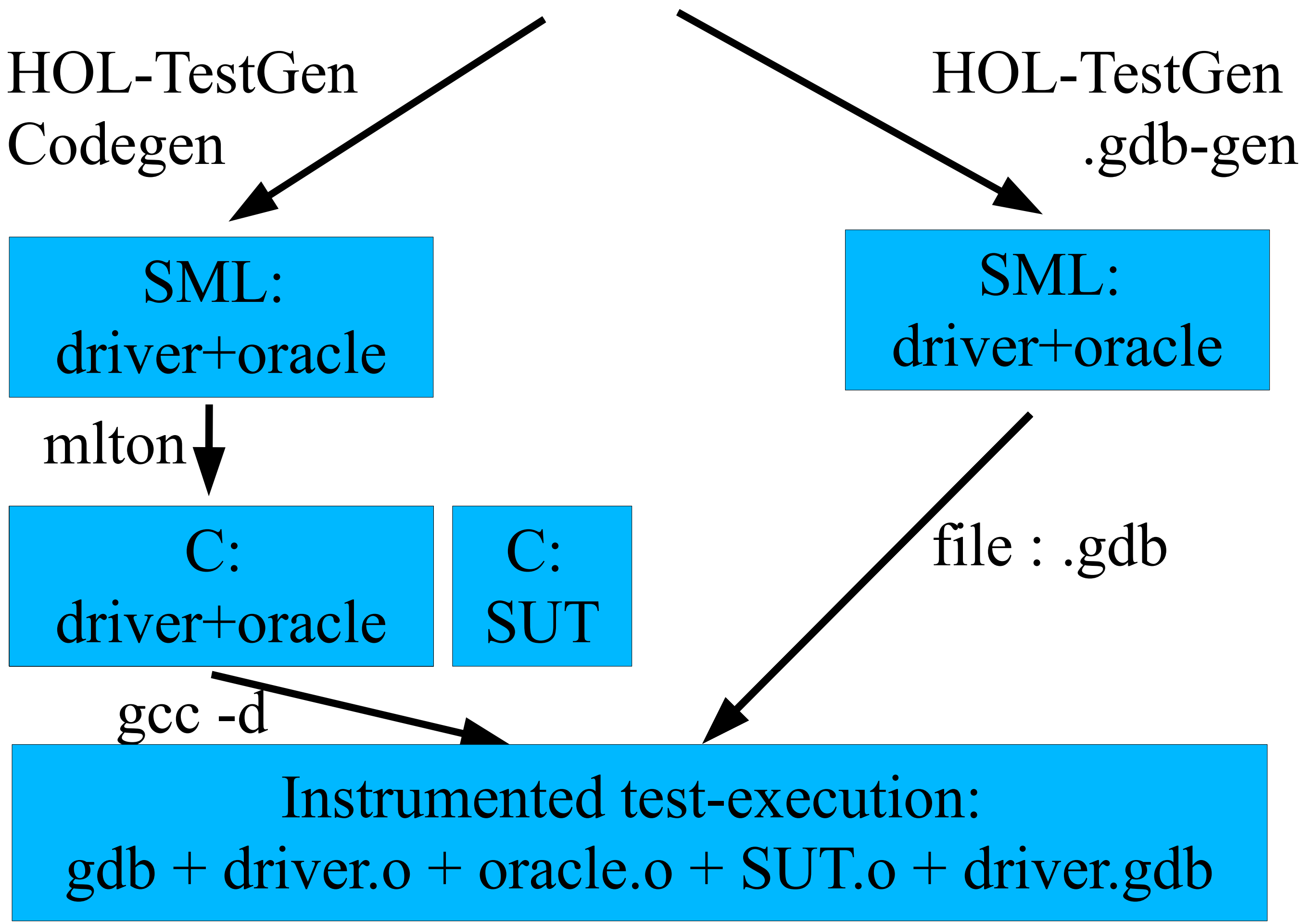
- Computing the input sequence as interleaving of atomic actions of system-API-Calls:

$$[\iota_1,\ldots,\iota_n] \in \text{interleave} \ (\text{IPC\_send } t_2 \ th_3) \ (\text{IPC\_receive } t_1 \ th_7)$$

where ι_j
a

Practice : How to test concurrent programs ?

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



Conclusion

Monadic approach to sequence testing:

1. no surrender to finitism and constructivism
2. sensible shift from syntax to semantics: computations + compositions, not nodes + arcs
3. explicit difference between input and output,
4. theoretical and practical framework of numerous conformance notions,
5. new ways to new calculi of symbolic evaluation

Conclusion

Testing Bank : a protocol with 3 operations …

1. Optimized split + prune essential.
2. Symbolic execution can be effectively realised with e-matching.
3. 10^8 protocol-load IS FEASABLE in Isabelle …
4. … in a well-designed symbolic execution process, the computation load is in the normalization(but this can be highly parallelized)

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 !