Theorem-prover based Testing with HOL-TestGen

Burkhart Wolff[1]

[1]Université Paris-Sud, LRI, Orsay, France
wolff@lri.fr

M2R: Test des Systemes Informatiques
Orsay, 28 Jan 2010

Outline

1. Revision: Apparent Limitations of Present Approaches
2. Nondetermistic Sequence Test
3. Reactive Sequences with Observers
4. Example: FTP Protocol
5. Summing Up

Outline

1. Revision: Apparent Limitations of Present Approaches
2. Nondetermistic Sequence Test
3. Reactive Sequences with Observers
4. Example: FTP Protocol
5. Summing Up

Outline

1. Revision: Apparent Limitations of Present Approaches
2. Nondetermistic Sequence Test
3. Reactive Sequences with Observers
4. Example: FTP Protocol
5. Summing Up

Apparent Limitations of HOL-TestGen

So far, we have used HOL-TestGen only for test specifications of the form:

$$pre\ x \longrightarrow post\ x\ (prog\ x)$$

- **No Non-determinism**.

Apparent Limitations of HOL-TestGen

So far, we have used HOL-TestGen only for test specifications of the form:

$$pre\ x \longrightarrow post\ x\ (prog\ x)$$

- post must indeed be executable; however, the pre-post style of specification represents a *relational* description of *prog*.

Apparent Limitations of HOL-TestGen

So far, we have used HOL-TestGen only for test specifications of the form:

$$pre\ x \longrightarrow post\ x\ (prog\ x)$$

- post must indeed be executable; however, the pre-post style of specification represents a *relational* description of *prog*.
- **No Automata** - No Tests for Sequential Behaviour.

Apparent Limitations of HOL-TestGen

So far, we have used HOL-TestGen only for test specifications of the form:

$$pre\ x \longrightarrow post\ x\ (prog\ x)$$

- post must indeed be executable; however, the pre-post style of specification represents a *relational* description of *prog*.
- HOL has lists and recursive predicates; thus sets of lists, thus languages . . .

Apparent Limitations of HOL-TestGen

So far, we have used HOL-TestGen only for test specifications of the form:

$$pre\ x \longrightarrow post\ x\ (prog\ x)$$

- post must indeed be executable; however, the pre-post style of specification represents a *relational* description of *prog*.
- HOL has lists and recursive predicates; thus sets of lists, thus languages . . .
- No possibility to describe **reactive tests**.

Apparent Limitations of HOL-TestGen

So far, we have used HOL-TestGen only for test specifications of the form:

$$pre\ x \longrightarrow post\ x\ (prog\ x)$$

- post must indeed be executable; however, the pre-post style of specification represents a *relational* description of *prog*.
- HOL has lists and recursive predicates; thus sets of lists, thus languages . . .
- HOL has Monads. And therefore means for IO-specifications.

Outline

1. Revision: Apparent Limitations of Present Approaches
2. Nondetermistic Sequence Test
3. Reactive Sequences with Observers
4. Example: FTP Protocol
5. Summing Up

A Deterministic Sequence Test Example I

Example: A little Bank - Acount System.

internal var register : table[client, nat]integer

op deposit (c : client, no : account_no, amount:nat) : unit
    pre (c,no) : dom(register)
    post register'=register[(c,no) := register(c,no) + amount]
op balance (c : client, no : account_no) : int
    pre (c,no) : dom(register)
    post register'=register and result = register(c,no)
op withdraw(c : client, no : account_no, amount:nat) : unit
    pre (c,no) : dom(register) and register(c,no) >= amount
    post register'=register[(c,no) := register(c,no) − amount]

A Non-Determin. Sequence Test Example II

Example: A Bank - Acount System with

```
internal var register : table[client, nat]integer
op  init(c : client, no : account_no) : unit
op  deposit (c : client, no : account_no, amount:nat) : unit
      pre  (c,no) : dom(register)
      post register'=register[(c,no) := register(c,no) + amount]
op  balance (c : client, no : account_no) : int
      pre  (c,no) : dom(register)
      post register'=register and result = register(c,no)
op  withdraw(c : client, no : account_no, amount:nat) : int
      pre  (c,no) : dom(register) and register(c,no) >= amount
      post 1<=result and result <= amount and
                register'=register[(c,no) := register(c,no) − result]
```

A Non-Determin. Sequence Test Example II

1. **Old Problem:** Only the public interface (i. e. the operations deposit, balance and withdraw. The internal (hidden) state is not accessible.
2. **Old Problem:** we can therefore only control the state by *sequences* of operation calls, not just produce data and leave it to one operation call as in unit tests.
3. **New Problem:** the operation `withdraw` may non-deterministically change the state (which can still be indirectly observed via outputs); we can therefore not pre-compute all input sequences.
4. The problem of initial states is solved by an explicit init-action creating an account for a client with an account number. (For convenience — but still realistic.)

A Non-Determin. Sequence Test Example II

1. Modified Test-Purpose :

   (init c no) (withdraw c no _ | deposit c no _)∗ (balance c no)

2. Modified Test-Specification:

   test_spec test_balance2:
   assumes test_purpose : "test_purpose c no ιs"
   shows _ $\models$(os $\leftarrow$mbind ιs SUT;
   return ($|\iota$s$|$ = $|$os$|$ $\wedge$ $\forall$i $\in$ {1..$|$os$|$}. post' i ιs os))

3. **Note**: This works only for those parts post' of the post-conditions that do not depend on the (not observable) internal state σ.

4. **Note**: For output-deterministic specifications post' can be defined, but the construction is neither necessarily constructive nor executable (=> involves theorem proving

A Non-Determin. Sequence Test Example II

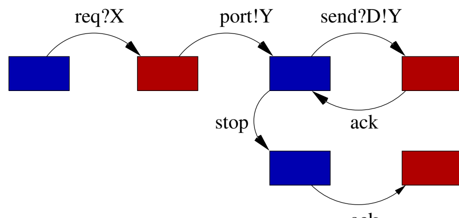
Note: We did not use anywhere the concrete state σ of the SUT::$\iota \longrightarrow$ (o,σ)MON_{SE}, we can therefore just pass a dummy (for example, the type unit).

Outline

1. Revision: Apparent Limitations of Present Approaches
2. Nondetermistic Sequence Test
3. Reactive Sequences with Observers
4. Example: FTP Protocol
5. Summing Up

Motivation: A Reactive System Example I

- A toy client-server system, a simplified FTP protocol:



a channel is requested within a bound X, a channel Y is chosen by the server, the client communicates along this channel . . .

Motivation: A Reactive System Example I

- A toy client-server system, a simplified FTP protocol:

$$
\begin{aligned}
&\mathrm{req}?X \rightarrow \mathrm{port}!Y[Y < X] \rightarrow \\
&\qquad (\mathrm{rec}\, N.\, \mathrm{send}!D.Y \rightarrow \mathrm{ack} \rightarrow N \\
&\qquad\qquad \Box\, \mathrm{stop} \rightarrow \mathrm{ack} \rightarrow \mathrm{SKIP})
\end{aligned}
$$

  a channel is requested within a bound X, a channel Y is chosen by the server, the client communicates along this channel . . .

Motivation: A Reactive System Example I

- A toy client-server system, a simplified FTP protocol:

$$\begin{aligned}&\text{req}?X \longrightarrow \text{port}!Y[Y < X] \longrightarrow \\ &\qquad (\text{rec}\, N.\, \text{send}!D.Y \longrightarrow \text{ack} \longrightarrow N \\ &\qquad\qquad \Box\, \text{stop} \longrightarrow \text{ack} \longrightarrow \text{SKIP})\end{aligned}$$

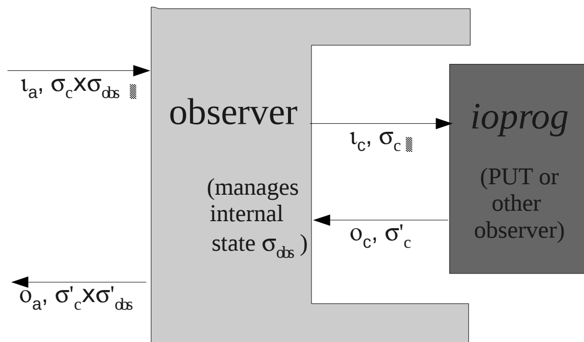
  a channel is requested within a bound X, a channel Y is chosen by the server, the client communicates along this channel ...

Motivation: A Reactive System Example II

Observation:

X and Y are only known at runtime!

- a test-driver is needed that manages a serialization of tests at test run time.
- ... including use an environment that keeps track of the instances of X and Y?
- **Infrastructure:** An **observer** maps
  abstract events (req X, port Y, ...) in traces
  to
  concrete events (req 4, port 2, ...) in runs!


ιa, σcXσobs
observer
(manages
internal
state σobs)
ιc, σc
oc, σ'c
ioprog
(PUT or
other
observer)
oa, σ'cXσ'obs

A formal definition looks as follows:

definition observer :: "[$\sigma_{obs} \Rightarrow o_c \Rightarrow \sigma_{obs}$,
$\sigma_{obs} \Rightarrow \iota_a \Rightarrow \iota_c$,
$\sigma_{obs} \Rightarrow \sigma \Rightarrow \iota_c \Rightarrow o_c \Rightarrow$ bool]
$\Rightarrow (\iota_c \Rightarrow (o_c, \sigma)\mathrm{MON}_{SE})$
$\Rightarrow (\iota_a \Rightarrow (o_c, \sigma_{obs} \times \sigma)\mathrm{MON}_{SE})$ "

where "observer rebind substitute postcond ioprog $\equiv$
$(\lambda\, \iota_a.\ (\lambda\, (\sigma_{obs}, \sigma).$ **let** ι_c = substitute $\sigma_{obs}\ \iota_a$ **in**
case ioprog $\iota_c \sigma$ **of**
None $\Rightarrow$None *(∗ ioprog failure − eg. timeout ... ∗)*
| Some (o_c, σ') $\Rightarrow$(**let** σ_{obs}' = rebind $\sigma_{obs}\ o_c$
in if postcond $\sigma_{obs}'\ \sigma'\ \iota_c$ out_c
then Some$(o_c, (\sigma_{obs}', \sigma'))$
else None *(∗ postcond failure ∗)*

As can be inferred from the type of observer, the function is a a monad-transformer; it transforms the *i/o stepping function* *ioprog* into another stepping function, which is the combined sub-system consisting of the observer and, for example, a program under test *PUT*.

Thus, our concept of an *i/o stepping function* serves as an interface for varying entities in (reactive) sequence testing.

Note that we made the following testability assumptions:

1. *ioprog* behaves wrt. to the reported state and input as a function, i.e. it behaves deterministically (in the modeled state!), and
2. it is not necessary to destinguish internal failure and post-condition-failure. (Modelling Bug ? This is superfluous and blind featurism ... One could do this by introducing an own "weakening"-monad endo-transformer.)

observer can actually be decomposed into two combinators - one dealing with the management of explicit variables and one that tackles post-conditions ...

where "observer3 rebind substitute ioprog $\equiv$

$(\lambda\ \iota_a .\ (\lambda\ (\sigma_{obs},\ \sigma).$

let ι_c = substitute $\sigma_{obs}\ \iota_a$

in **case** ioprog $\iota_c \sigma$ **of**

None $\Rightarrow$None *(∗ ioprog failure − eg. timeout ...*

| Some (o_c, σ') $\Rightarrow$(**let** $\sigma_{obs}{}'$ = rebind $\sigma_{obs}\ o_c$

in Some$(o_c, (\sigma_{obs}', \sigma'))$)))"

and ...

```
where   "observer4 postcond ioprog ≡
        (λ ι. (λ σ. case ioprog ισ of
                   None ⇒None (∗ ioprog failure − eg. timeout ... ∗)
                 | Some (o, σ’) ⇒ ( if  postcond σ’ ι o
                                        then Some(o, σ’)
                                        else None (∗ postcond failure
```

Note that all three definitions of observers are *executable*.

We can build on top of the observer function definitions some theory on observers, which might pave the way for future optimizations. For example, the following decomposition theorem holds:

theorem observer_decompose:
  "observer r s (λ x. pc) io = (observer3 r s (observer4 pc io))"

The abstraction assures that pc is a function not referring to the observer state.

Outline

1 Revision: Apparent Limitations of Present Approaches

2 Nondetermistic Sequence Test

3 Reactive Sequences with Observers

4 Example: FTP Protocol

5 Summing Up

FTP Protocol Example II

We specify explicit variables and a joined type containing abstract events (replacing values by explicit variables) as well as their concrete counterparts.

```
datatype vars = X | Y
datatype data = Data
types    chan = int (∗ just to make it executable ∗)
```

Abstract and concrete events ...

datatype $\text{InEvent}_c\text{onc}$ = req chan | send data chan | stop
datatype $\text{InEvent}_a\text{bs}$ = reqA vars | sendA data vars | stopA
datatype $\text{OutEvent}_c\text{onc}$ = port chan | ack
datatype $\text{OutEvent}_a\text{bs}$ = portA vars | ackA

types InEvent = "$\text{InEvent}_a\text{bs}$ + $\text{InEvent}_c\text{onc}$"
types OutEvent = "$\text{OutEvent}_a\text{bs}$ + $\text{OutEvent}_c\text{onc}$"
types event_abs = "$\text{InEvent}_a\text{bs}$ + $\text{OutEvent}_a\text{bs}$"

The function subsitute maps abstract events containing explicit variables to concrete events by substituting the variables by values communicated in the system run.
It requires an environment (“substitution”) where the concrete values occuring in the system run were assigned to variables.

```
definition lookup :: "['a ⇀'b, 'a] ⇒ 'b"
where       "lookup env v ≡the(env v)"

consts substitute :: "[vars ⇀chan, InEvent_abs] ⇒InEvent_conc"
primrec
   "substitute env (reqA v) = req(lookup env v)"
   "substitute env (sendA d v)= send d (lookup env v)"
   "substitute env stopA      = InEvent_conc.stop"
```

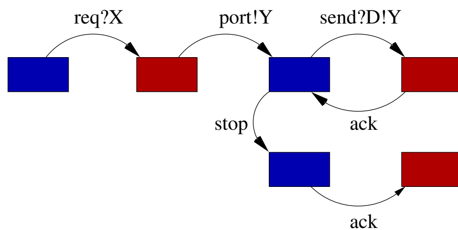
This environment is the *observer state* σ_{obs}.

The function rebind extracts from concrete output events the values and binds them to explicit variables in env. (= σ_{obs})
The predicate rebind only stores occurrences of input-events (marked by ?) in the protocol into the environment; output (!)-occurences were ignored.

```
consts rebind :: "[vars ⇀chan, OutEvent_conc] ⇒vars ⇀chan"
primrec
   "rebind env (port n)            = env(Y ↦n)"
   "rebind env OutEvent_conc.ack = env"
```

In a way, rebind can be viewed as an abstraction of the concrete log produced at runtime.

Revisit the protocol automaton:



Test-purpose specification (= protocol specification) is as follows (we view the enumeration type A=0 as abbreviation).

```
consts accept' :: "nat ×event_abs list ⇒bool"
recdef accept' "measure(λ (x,y). length y)"
       "accept'(A,(Inl(reqA X))#S)  = accept'(B,S)"
       "accept'(B,(Inr(portA Y))#S) = accept'(C,S)"
       "accept'(C,(Inl(sendA d Y))#S) = accept'(D,S)"
       "accept'(D,(Inr(ackA))#S)    = accept'(C,S)"
       "accept'(C,(Inl(stopA))#S)   = accept'(E,S)"
       "accept'(E,[Inr(ackA)])      = True"
       "accept'(x,y)                = False"

constdefs
        accept :: "event_abs list ⇒bool"
```

Actually, this is merely an academic exercise - we use for testing merely the subsequent protocol automaton:

We proceed by modeling a subautomaton of the protocol automaton accept.

```
consts   stim_trace' :: "nat ×InEvent_abs list ⇒bool"
recdef   stim_trace' "measure(λ (x,y). length y)"
        "stim_trace'(A,(reqA X)#S) = stim_trace'(C,S)"
        "stim_trace'(C,(sendA d Y)#S) = stim_trace'(C,S)"
        "stim_trace'(C,[stopA])      = True"
        "stim_trace'(x,y)                   = False"

constdefs stim_trace :: "InEvent_abs list ⇒bool"
        "stim_trace s ≡stim_trace'(A,s)"
```

```
consts  postcond' :: "((vars ⇀int)×σ×InEvent_conc×OutEvent_con
bool"

recdef  postcond' "{}"
        "postcond' (env, _, req n, port m) = (m <= n)"
        "postcond' (env, _, send z n, ack) = (n = lookup env Y)"
        "postcond' (env, _, stop, ack)       = True"
        "postcond' (env, _, y, z)                  = False"

constdefs postcond :: "(vars ⇀int) ⇒'σ ⇒InEvent_conc ⇒OutEve
bool"
        "postcond env σy z ≡postcond' (env, σ, y, z)"
```

test_spec "stim_trace ιs $\Longrightarrow$
(empty[X$\mapsto$ x],())
$\models$(os$\leftarrow$(mbind ιs(observer$_2$ rebind substitute postcond ioprog));
result(length ιs = length os))"

where ioprog is the program under test. The initial state consists of a suitably initialized observer state (the client-controlled X must be initialized), whereas we provide for the server-side state σ, which is nowhere used in the model (in particular not in postcond) and therefore polymorphic, is instantiated by the dummy type unit and its element ()).

1. ([X $\mapsto$?X1], ())

   $\models$(os $\leftarrow$mbind [reqA X,stop] (observer$_2$ rebind substitute pos

   result(2=length os)

3. ([X $\mapsto$?X2], ())

   $\models$(os $\leftarrow$mbind [reqA X,sendA Data Y,stop] (observer$_2$ rebind

   result(3 = length os))

5. ([X $\mapsto$?X3], ())

   $\models$(os $\leftarrow$mbind [reqA X, sendA Data Y,sendA Data Y,stop] (obs

   result(4 = length os))

7. ([X $\mapsto$?X4], ())

   $\models$(os $\leftarrow$mbind [reqA X,sendA Data Y,sendA Data Y,sendA Dat

   result (5 = length os))

9. ...

where we left out the test hypotheses. The meta-variables serve just as a place-holder for the initial (client-controlled) value for the X.

Outline

1 Revision: Apparent Limitations of Present Approaches
2 Nondetermistic Sequence Test
3 Reactive Sequences with Observers
4 Example: FTP Protocol
5 Summing Up

Summing Up

1. Test-Sequence generation can be formalized as a constraint-resolution problem, too.
2. Reason: We have data-types (this lists and laguages) and Monads in HOL
3. Test-drivers can be generated as well
4. Handling of Testability hypotheses implicit (control over the init-state, *PUT* a function in the sense of the specification)

Summing Up

- **Explicit Test Hypothesis** are controllable by the test-engineer (can be seen as proof-obligation!)
- In HOL, Sequence Testing and Unit Testing are the same!
- The Sequence Test Setting of HOL-TestGen is **effective** (see Firewall Test Case Study)

Summing Up

- **Explicit Test Hypothesis** are controllable by the test-engineer (can be seen as proof-obligation!)
- In HOL, Sequence Testing and Unit Testing are the same! TS pattern **Unit Test**:

$$\mathrm{pre}\, x \longrightarrow \mathrm{post}\, x (prog\, x)$$

- The Sequence Test Setting of HOL-TestGen is **effective** (see Firewall Test Case Study)

Summing Up

- **Explicit Test Hypothesis** are controllable by the test-engineer (can be seen as proof-obligation!)
- In HOL, Sequence Testing and Unit Testing are the same! TS pattern **Sequence Test**:

$$\text{accept}\, \iota s \Longrightarrow \sigma_0 \models os \leftarrow \text{mbind}\, \iota s\, prog; \text{result}\ P\, \iota s\, os$$

- The Sequence Test Setting of HOL-TestGen is **effective** (see Firewall Test Case Study)

Summing Up

- **Explicit Test Hypothesis** are controllable by the test-engineer (can be seen as proof-obligation!)
- In HOL, Sequence Testing and Unit Testing are the same! TS pattern **Reactive Sequence Test**:

$$\text{accept } \mathit{trace} \Longrightarrow$$
$$\sigma_0 \models os \leftarrow \text{mbind } \iota s \text{ (observer rebind subst } \mathit{prog}); \text{ result } P\, \iota s\, os$$

- The Sequence Test Setting of HOL-TestGen is **effective** (see Firewall Test Case Study)

Bibliography I