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, 21 Jan 2010

Outline

1. Introduction to Sequence Testing
2. Foundation: State-Monads
3. Connecting Specifications and Test-Sequences
4. Test-Case Generation
5. Summing Up

Outline

1. Introduction to Sequence Testing
2. Foundation: State-Monads
3. Connecting Specifications and Test-Sequences
4. Test-Case Generation
5. Summing Up

Motivation: Sequence Test

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

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

- This seems to limit the HOL-TestGen approach to **UNIT**-tests.
- This seems to exclude testing of systems with internal state.

Motivation: 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)

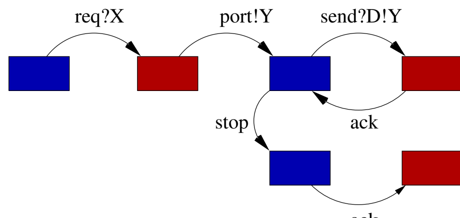
op withdraw(c : client, no : account_no, amount:nat) : unit
pre (c,no) : dom(register) and register(c,no) >= amount
post register'=register[(c,no) := register(c,no) − amount]

Motivation: Sequence Test Example II

1. **Problem:** Only the public interface (i. e. the operations deposit, balance and withdraw. The internal (hidden) state is not accessible.
2. **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 uit tests.
3. **Problem:** The spec does not speak about the initial states.

Motivation: A Reactive System Example I

- A toy client-server system:



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:

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

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:

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

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 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!

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. Introduction to Sequence Testing
2. Foundation: State-Monads
3. Connecting Specifications and Test-Sequences
4. Test-Case Generation
5. Summing Up

The core of state-based computations:

state transitions from state σ to σ' emmiting output out!

Such state-transitions can be modeled in various ways:

- as total functions: $\sigma \Rightarrow (o \times \sigma)$
- as partial functions: $\sigma \Rightarrow (o \times \sigma)$ option
- as relations: $\sigma \Rightarrow (o \times \sigma)$ set
- as finite series relation: $\sigma \Rightarrow (o \times \sigma)$ list
- as infinite series relation: $\sigma \Rightarrow (o \times \sigma)$ sequence
- ...

We write for this form of type scheme $(o, \sigma)Mon_\phi$ for ϕ in $\{, option, set, list, \ldots\}$. Note that $(o, \sigma)Mon_\phi$ in itself is not a type in the Isabelle type-system (only the instances thereof).

Background:

If a type $(o, \sigma)Mon_\phi$ is completed to an algebraic stucture with two operations :

$$\texttt{bind}_\phi :: [(\alpha, \sigma)Mon_\phi, \alpha \Rightarrow (\beta, \sigma)Mon_\phi] \Rightarrow (\beta, \sigma)Mon_\phi$$

and

$$\texttt{unit}_\phi :: \alpha \Rightarrow (\alpha, \sigma)Mon_\phi$$

satisfying the associativity and both neutrality laws:

1. **associativity**: $\texttt{bind}_\phi\ F\ (\lambda y.\texttt{bind}_\phi\ G\ H) = \texttt{bind}_\phi(\texttt{bind}_\phi\ F\ (\lambda y.\texttt{bind}_\phi\ G)\ H$
2. **neutrality_left**: $\texttt{bind}_\phi\ (unit\ F)\ G = G$
3. **neutrality_right**: $\texttt{bind}_\phi\ F(unitG) = F$

What is the Relevance for Computing?

1. Monads talk of the sequential "glue", the _;_ and `resulte` in imperative languages.
2. Monads are an abstraction of "computational structures" arranging computations based on an underying state. This can be used in (for example):
   1. computations based on state
   2. computations based on state involving exceptions
   3. computations based on state involving backtracking
   4. computations based on state involving alltogether
   5. ...
3. They have in intensively used for the study of programming and specification language semantics
4. ... some of them are executable and were intensively used in purely functional languages such as Haskell.

A basic case for "imperative programming": the *state-exception*-Monad Mon_{SE} based on the type $(o, \sigma)Mon_{SE} = \sigma \Rightarrow (o \times \sigma)option$.

1. It composes partial functions
2. In case a function evaluation fails (which can be viewed as "an exception occured"), the execution is stopped and the state remains unchanged (pretty much like Java or SML),
3. ... otherwise the execution continues with the new state.
4. $\texttt{unit}_{SE}$ corresponds to the usual "result" operation.

We define:

1. **definition** bind_{SE}::[(o,σ)MON_{SE},o⇒(o,σ)MON_{SE}]⇒(o,σ)MON_{SE}
   where "bind_{SE} f g $\equiv \lambda\ \sigma$. **case** f σ **of**
   None ⇒None
   | Some (out, σ') ⇒g out σ'"

2. **definition** unit_{SE} :: "o ⇒(o, σ)MON_{SE}"
   where "unit_{SE} e $\equiv \lambda\ \sigma$. Some(e,σ)"

where we use the syntax

$$x \leftarrow f; g\ x$$

for $\mathtt{bind}_{SE}\ f\ (\lambda x.g)$ and $\mathtt{return}\ e$ for $\mathtt{unit}_{SE}\ e$.

Test Sequences as Monadic Compositions

In the state exception monad, we can already represent a particular form of test-driver equivalent to a *test sequence*:

1. A **test sequence** has the form:

$$x_1 \leftarrow put_1; x_2 \leftarrow (\lambda_.\ put_2); \ldots; x_n \leftarrow (\lambda_.\ put_n);$$
$$\mathtt{return}(post\ x_1\ \ldots\ x_n)$$

   i. e. the program steps under test put_i do not depend from output of prior steps.

2. A **reactive test sequence** has the form:

$$x_1 \leftarrow put_1; x_2 \leftarrow put_2\ x_1; \ldots; x_n \leftarrow\ put_n\ x_1\ \ldots\ x_{n-1};$$
$$\mathtt{return}(post\ x_1\ \ldots\ x_n)$$

   i. e. the program steps under test put_i **may depend** from output of prior steps.

In order to make test-sequences amenable to HOL-TestGen, we need to represent them as data-types (so: lists of put_i). We introduce a $multi - bind$ combinator taking **a list of io-stepping functions** (i. e., in particular, put_I's) and executes them while taking exceptions into account:

```
consts mbind ::[ι list, ι ⇒ (o,σ)MON_SE] ⇒(o list,σ)MON_SE
primrec
  "mbind [] iostep σ= Some([], σ)"
  "mbind (a#H) iostep σ=
        (case iostep a σ of
           None  ⇒ Some([], σ)
          |Some (out,σ') ⇒(case mbind H iostep σ' of
                              None   ⇒ Some([out],σ')
                             |Some(outs,σ'')⇒ Some(out#outs,σ'')
```

Note that mbind has a slightly different behaviour than $bind_{SE}$ wrt. exceptions!

On this level, we can now state **valid test sequences** as a test specification of the form:

$$\sigma_0 \models (os \leftarrow (\mathtt{mbind}\ \iota s\ ioprog); \mathtt{return}(post\ os))$$

where the σ_0 is the initial state and the *validity statement* $_ \models _$ means: start computation *ioprog* in the initial state and run it sequentially over the input sequence ιs and transfer all outputs *os* to the post condition. Sequences are *valid* iff the postcondition is true. The *validity statement* is defined as follows:

definition valid :: $\sigma \Rightarrow (\text{bool},\sigma)\text{MON}_{SE} \Rightarrow \text{bool}$ (infix $\models$15)
where $\sigma \models \text{m} \equiv (\text{m}\ \sigma \neq \text{None} \wedge \text{fst}(\text{the}\ (\text{m}\ \sigma)))$

Remark: From valid test sequence, HOL-TestGen test were generated by exploring the data-structure *input sequence* ιs up to given depths k by the standard mechanisms used in unit-tests.

However, it may be convenient to specify constraints on ιs, let it be by automata, by regular expressions, by temporal formulas or by other means. In the literature, these constraints were also called **test purposes** (TP).

$$TP(\iota s) \Longrightarrow \sigma_0 \models (os \leftarrow (\texttt{mbind}\ \iota s\ ioprog); \texttt{return}(post\ os))$$

A basic case for the "state transition system specification": the *state-relation*-Monad Mon_{SB} based on the type $(o, \sigma)Mon_{SB} = \sigma \Rightarrow (o \times \sigma)set$.

1. It composes relations on states (involving input and output)
2. In case a function evaluation fails (which can be viewed as "an exception occured"), the execution is stopped and the state remains unchanged (roughly like PROLOG),
3. ... otherwise the execution continues with the new state.
4. $\texttt{unit}_{SB}$ corresponds to the usual "result" operation.

We define:

1. **definition** bind_{SB}::[$(\alpha,\sigma)\text{MON}_{SB}$,$\alpha\Rightarrow(\beta,\sigma)\text{MON}_{SB}$] $\Rightarrow(\beta,\sigma)\text{MON}_{SB}$
   where "bind_{SB} f g σ $\equiv\bigcup$((λ(out, σ). (g out σ)) ‘ (f σ))"

2. **definition** unit_{SB}:: o $\Rightarrow$(o,σ)MON_{SB}
   where "unit_{SB} e $\equiv\lambda$ σ . {(e,σ)}"

where we use the syntax

$$x \leftarrow f; ; g\ x$$

for $\texttt{bind}_{SB}\ f\ (\lambda x.g)$ and `returns` e for $\texttt{unit}_{SB}\ e$.

In contrast to MON_{SE}, the operations of MON_{SB} are not executable in general (**why?**).

On the other hand, concepts like pre- and post conditions can be easily expressed in terms of MON_{SB}.

Example: The post-condition of the operation `balance` is directly expressed in HOL as:

$$\begin{aligned} &\texttt{post}(c :: \mathit{client}, \mathit{no} :: \mathit{account_no}) = \\ &\qquad \lambda\,\sigma.\{(\mathit{result}, \sigma') \mid \sigma = \sigma' \wedge \mathit{result} = \mathit{the}(\mathit{register}(c, \mathit{no}))\} \end{aligned}$$

Outline

1. Introduction to Sequence Testing
2. Foundation: State-Monads
3. Connecting Specifications and Test-Sequences
4. Test-Case Generation
5. Summing Up

Revisiting the Little Bank 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]
```

In order to formalize input and output implicit in such a specification, such that we can consider it uniformely as "a list of input data" and "a list of output data", we need to **convert** the given interface into

1. a type for the internal state,
2. a uniform data-type containing all inputs, and
3. a uniform data-type containing all outputs.

This so-called **interface encapsulation** is a syntactic transformation and could in principle be done automatically. (Not supported yet in HOL-TestGen).

Example: Interface Encapsulation

For “Little Bank”, we have:

1. a type for the internal state `register`:
   $(\text{client} \times \text{nat}) \rightharpoonup \text{int}$
2. the inputs data-type:

   datatype **in**$_c$ = deposit client account_no nat
   | withdraw client account_no nat
   | balance client account_no

3. a uniform data-type containing all outputs:

   datatype out$_c$ = depositO| balanceO nat | withdrawO

This so-called **interface encapsulation** is a syntactic transformation and could in principle be done automatically. (Not supported yet in HOL-TestGen).

Also pre-and post-conditions of “Little Bank” were encapsulated, such that we have now a typed state transition system on σ (= register), $\mathbf{in}_c$ and out_c.

consts precond :: "register $\Rightarrow \mathbf{in}_c \Rightarrow$ bool"
primrec
"precond σ (deposit c no m) = ((c,no) $\in$ dom σ)"
"precond σ (balance c no) = ((c,no) $\in$ dom σ)"
"precond σ (withdraw c no m) = ((c,no) $\in$ dom σ
$\wedge$ (int m) $\leq$ the(σ(c,no)))"

The post-condition looks as follows:

```
consts postcond :: "register ⇒ in_c ⇒ out_c × register ⇒ bool"
primrec
 "postcond σ(deposit c no m) =
        (λ (n,env'). (n = depositO
                    ∧ σ'=σ ((c,no)↦ the(env(c,no)) + int m)))"
 "postcond σ(balance c no)    =
        (λ (n,env'). (σ=σ' ∧ (∃ x. balanceO x = n
                                ∧ x = nat(the(σ(c,no))))))"
 "postcond σ(withdraw c no m) =
        (λ (n,env'). (n = withdrawO
                    ∧ σ'=σ((c,no)↦ the(env(c,no)) − int m)))"
```

The following combinators — based on the Hilbert-Operator — hold the key for a conversion between monads:

definition impl ::$[[\sigma,\iota] \Rightarrow$bool,$\iota \Rightarrow$ (o,σ)$MON_{SB}] \Rightarrow \iota \Rightarrow$(o,$\sigma$)$MON_{SE}$
where "impl pre post ι=
(λ σ . if pre σ ι
then Some(SOME(out,σ'). post $\iota\sigma$(out,σ'))
else arbitrary)"

definition strong_impl ::$[[\sigma,\iota] \Rightarrow$bool, $\iota \Rightarrow$(o,σ)$MON_{SB}] \Rightarrow$ $\iota \Rightarrow$(o,σ)M
where "strong_impl pre post ι=
(λ σ . if pre σ ι
then Some(SOME(out,σ'). post $\iota\sigma$(out,σ'))
else None)"

```
definition is_strong_impl :: "['σ ⇒'ι ⇒ bool,
                              'ι ⇒ ('o,'σ)MON_SB,
                              'ι ⇒ ('o, 'σ)MON_SE] ⇒bool"
where   "is_strong_impl pre post ioprog =
                  (∀ σ ι. (¬pre σ ι ∧ ioprog ι σ = None) ∨
                          (pre σ ι ∧ (∃ x. ioprog ι σ = Some x)))
```

This results in the following:

```
theorem "is_strong_impl pre post (strong_impl pre post)"
```

This following characterization of implementable specifications gives the key for turning specs into programs. First, we define the concept of an **implementable** specification, i. e. the fact that there is a function that maps leagal input to output/state pairs, that satisfy the postcondition:

definition implementable::[$\sigma \Rightarrow \iota \Rightarrow$bool, $\iota \Rightarrow$(o,σ)MON_{SB}] $\Rightarrow$bool
where "implementable pre post =
(∀ σ ι. pre σ ι $\longrightarrow$(∃ out σ'. post ι σ (out,σ')))"

This results in the following characterization theorem:

theorem implementable$_c$harn:
"⟦implementable pre post; pre $\sigma\iota$⟧ $\Longrightarrow$
post ι σ (the(strong_impl pre post $\iota\sigma$))"

It is now straight-forward to “convert” our (interface encapsulated) specification into a program. Simply:

```
strong_impl precond postcond
```

does the trick.

This program will report violations of pre- and postconditions as exceptions which were then treated at run-time.

Problem: How can we use the specification to *generate* test-sequences symbolically?

Observation: Our specification is *state-deterministic*, i. e. for each observable output, there is at most one corresponding state.

For this type of specification, we can use HOL-TestGen as follows: we state:

$\sigma_0 \models$ s $\leftarrow$ mbind S (strong_impl precond postcond); return(s = x)

as a constraint, let HOL-TestGen find solutions for x, and use these solutions in the generated test drivers.

For this, we need the generic symbolic evaluation rules:

$(\sigma \models (s \leftarrow \text{return } x \,;\, \text{return } (P\ s))) = P\ x$

$(\sigma \models (s \leftarrow \text{mbind } (a\#S)\ \text{ioprog} \,;\, \text{return } (P\ s))) =$
$\quad(\textbf{case}\ \text{ioprog}\ a\ \sigma\ \textbf{of}$
$\quad\quad \text{None} \Rightarrow (\sigma \models (\text{return } (P\ [])))$
$\quad\ \mid\ \text{Some}(b,\sigma') \Rightarrow (\sigma' \models (s \leftarrow \text{mbind } S\ \text{ioprog} \,;\, \text{return } (P\ (b\#s)))))$

The introduced case-statements were eliminated in the case-splitting of the test-case-generation phase.

... and the program specific symbolic evaluation rules (where H = (strong_implprecond postcond)):

$(\sigma \models (s \leftarrow \text{mbind } ((\text{deposit c no m})\#S)\ H;\ \text{return } (P\ s))) =$
$\quad (\text{if } (c, no) \in \text{dom } \sigma$
$\quad\ \text{then } (\sigma((c, no) \mapsto \text{the } (\sigma\ (c, no)) + \text{int } m))$
$\qquad\qquad \models (s \leftarrow \text{mbind } S\ H;\ \text{return } (P\ (\text{depositO}\#s)))$
$\quad\ \text{else } (\sigma \models (\text{return } (P\ []))))$

$(\sigma \models (s \leftarrow \text{mbind } ((\text{balance c no})\#S)\ H;\ \text{return } (P\ s))) =$
$\quad (\text{if } (c, no) \in \text{dom } \sigma$
$\quad\ \text{then } (\sigma \models (s \leftarrow \text{mbind } S\ H;$
$\qquad\qquad \text{return } (P\ (\text{balanceO}(\text{nat}(\text{the } (\sigma\ (c, no))))\#s))))$
$\quad\ \text{else } (\sigma \models (\text{return } (P\ []))))$

...

Outline

1. Introduction to Sequence Testing
2. Foundation: State-Monads
3. Connecting Specifications and Test-Sequences
4. Test-Case Generation
5. Summing Up

Generating all possible input sequences is far too general: there would a lot of superfluous attempts to access a wrong account with a wrong account number, far too many initial states.

In order to reduce the number of possible input sequences, we define a *test purpose*, i. e. a predicate that constrains the number of possible input traces for one given client with an account which is initially empty.

This raises a particular *testability assumption* (at the beginning, the system is in particular initial state) which results from our lacking init method in our interface.

This test-purpose is formalized as follows:

```
consts test_purpose :: "[client, account_no, in_c list] ⇒bool"
primrec
    "test_purpose c no [] = False"
    "test_purpose c no (a#R) = (case R of
                               []  ⇒ a = balance c no
                             | a'#R' ⇒(((∃ m. a = deposit c no m) ∨
                                        (∃ m. a = withdraw c no m)) ∧
                                        test_purpose c no R))"
```

This terst-purpose formalizes that the input sequences belong to the language expressed as regular expression:

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

The test-specification is formalized as follows:

test_spec test_balance:
assumes account_defined: "(c,no) $\in$dom σ_0"
and test_purpose : "test_purpose c no ιs"
and symbolic_run_yields_x :
"σ_0 $\models$(s $\leftarrow$mbind ιs (strong_impl precond postcond);
return (s = x))"
shows " σ_0 $\models$(s $\leftarrow$mbind ιs SUT; return (s = x))"

The resulting test-theorem for k=5 looks follows:

1. $(\lambda a.\ \text{Some } 2) \models$
   (s $\leftarrow$mbind [balance ?X1 ?X2] SUT; return s = [balanceO 2])
2. THYP ...
3. $(\lambda a.\ \text{Some } 5) \models$
   (s $\leftarrow$mbind
   [deposit ?X3 ?X4 ?X5, balance ?X3 ?X4]
   SUT; return s = [depositO, balanceO (nat (5 + int ?X5))])
4. THYP ...
5. THYP
6. int ?X6 $\leq$7 $\Longrightarrow$
   $(\lambda a.\ \text{Some } 7) \models$(s $\leftarrow$mbind
   [withdraw ?X7 ?X8 ?X6, balance ?X7 ?X8] SUT;
   return s = [withdrawO, balanceO (nat (7 $-$ int ?X6))])

Caution: Which are the underlying *Testability Hypothesis* (to be clear: *not* Test-Hypotheses) of this problem ???

Well, we made two (more or less explicit) testability hypothesis underlying our test-construction, that must be assured by other means than just running the test:

1. **initialization condition** (reflected by the assumption $(c,no) \in \mathrm{dom}\ \sigma_0$). We must assume that a concrete user and accountnumber is defined.
2. **determinism condition** (reflected by the assumption that SUT has type $\mathbf{in}_c \Rightarrow (out_c, register) Mon_{SE}$). We assume that SUT behaves indeed like a function in a state in the sense of our model; we assume it is deterministic and will not have *hidden* state or engage in *hidden* state-transitions (like clocks, etc.)

Pragmatically: if we detect violations against these hypotheses during testing, we must refine our model ...

Outline

1. Introduction to Sequence Testing
2. Foundation: State-Monads
3. Connecting Specifications and Test-Sequences
4. Test-Case Generation
5. 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)

Bibliography I