Model-based Sequence-Testing
with HOL-TestGen

Burkhart Wolff

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

M2R: Test des Systemes Informatiques
Orsay, 4 Jan 2011
1. Introduction to Sequence Testing
2. Foundation: State-Monads
3. Connecting Specifications and Test-Sequences
4. Test-Case Generation
5. Summing Up
6. Introduction to Reactive Sequence
7. Example: FTP Protocol
Outline

1. Introduction to Sequence Testing
2. Foundation: State-Monads
3. Connecting Specifications and Test-Sequences
4. Test-Case Generation
5. Summing Up
6. Introduction to Reactive Sequence
7. Example: FTP Protocol
So far, we have used HOL-TestGen only for test specifications of the form:

\[ \text{pre } x \rightarrow \text{post } x \ (\text{prog } x) \]

This seems to limit the HOL-TestGen approach to \textbf{UNIT}-tests.

This seems to exclude testing of systems with internal state.
Motivation: Sequence Test Example I

Example: A little Bank - Account 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.
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 . . .
A toy client-server system:

\[
\text{req}\ ?X \rightarrow \text{port}!Y[Y < X] \rightarrow
\]

\[
\left( \text{rec } N. \text{send}!D.Y \rightarrow \text{ack} \rightarrow N \quad \square \text{stop} \rightarrow \text{ack} \rightarrow \text{SKIP} \right)
\]

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:

\[
\text{req?}X \rightarrow \text{port!}Y[Y < X] \rightarrow \\
(\text{rec } N. \text{send!}D.Y \rightarrow \text{ack} \rightarrow N \\
\square \text{stop} \rightarrow \text{ack} \rightarrow \text{SKIP})
\]

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:

\[ X \text{ and } Y \text{ are only known at runtime!} \]

- 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:

\[ pre \ x \rightarrow post \ x \ (prog \ x) \]

- **No Non-determinism.**
Apparent Limitations of HOL-TestGen

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

\[ \text{pre } x \rightarrow \text{post } x \ (\text{prog } x) \]

- post must indeed be executable; however, the pre-post style of specification represents a \textit{relational} description of \textit{prog}.
Apparent Limitations of HOL-TestGen

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

\[ \text{pre } x \rightarrow \text{post } x \ (\text{prog } x) \]

- post must indeed be executable; however, the pre-post style of specification represents a \textit{relational} description of \textit{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:

\[ pre \ x \rightarrow post \ x \ (prog \ x) \]

- 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:

\[ pre \ x \rightarrow post \ x \ (prog \ x) \]

- 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:

\[ \text{pre } x \rightarrow \text{post } x (\text{prog } x) \]

- post must indeed be executable; however, the pre-post style of specification represents a \textit{relational} description of \textit{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
6. Introduction to Reactive Sequence
7. Example: FTP Protocol
The core of state-based computations:
state transitions from state $\sigma$ to $\sigma'$ emitting 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 $\phi$ in \{option, set, list, ...\}. Note that $(o, \sigma)Mon_\phi$ in itself is not a type in the Isabelle type-system (only the instances thereof).
If a type $(o, \sigma)Mon_\phi$ is completed to an algebraic structure with two operations:

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

and

\[
\text{unit}_\phi :: \alpha \Rightarrow (\alpha, \sigma)Mon_\phi
\]

satisfying the associativity and both neutrality laws:

1. **associativity:**
   \[
   \text{bind}_\phi F (\lambda y. \text{bind}_\phi G H) = \text{bind}_\phi (\text{bind}_\phi F (\lambda y. \text{bind}_\phi G) H
   \]

2. **neutrality_left:**
   \[
   \text{bind}_\phi (\text{unit} F) G = G
   \]

3. **neutrality_right:**
   \[
   \text{bind}_\phi F(\text{unit} G) = F
   \]
What is the Relevance for Computing?

1. Monads talk of the sequential “glue”, the _; _ and result in imperative languages.

2. Monads are an abstraction of “computational structures” arranging computations based on an underlying state. This can be used in (for example):
   - computations based on state
   - computations based on state involving exceptions
   - computations based on state involving backtracking
   - computations based on state involving altogether
   - ...

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 $\textit{Mon}_{SE}$ based on the type $(\sigma, \sigma) \textit{Mon}_{SE} = \sigma \Rightarrow (\sigma \times \sigma)\text{option}$.

1. It composes partial functions
2. In case a function evaluation fails (which can be viewed as “an exception occurred”), 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. $\text{unit}_{SE}$ corresponds to the usual “result” operation.
We define:

1. **definition** \( \text{bind}_{\text{SE}} :: [(o, \sigma) \text{MON}_{\text{SE}}, o \Rightarrow (o, \sigma) \text{MON}_{\text{SE}}] \Rightarrow (o, \sigma) \text{MON}_{\text{SE}} \)
   where \( \text{bind}_{\text{SE}} f g \equiv \lambda \sigma . \text{case } f \sigma \text{ of } \)
   
   None \( \Rightarrow \) None
   
   | Some (out, \sigma') \( \Rightarrow g \) out \( \sigma' \)"

2. **definition** \( \text{unit}_{\text{SE}} :: "o \Rightarrow (o, \sigma) \text{MON}_{\text{SE}}" \)
   where \( \text{unit}_{\text{SE}} e \equiv \lambda \sigma . \text{Some}(e, \sigma)" \)

where we use the syntax

\[ x \leftarrow f ; g \ x \]

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

   \[x_1 \leftarrow put_1; x_2 \leftarrow (\lambda \_. put_2); \ldots; x_n \leftarrow (\lambda \_. put_n); \]  
   \[\text{return}(post \ x_1 \ldots x_n)\]

   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:

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

   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 \texttt{multi-\ bind} combinator taking a list of \textbf{io-stepping functions} (i.e., in particular, \( put_i \)'s) and executes them while taking exceptions into account:

\begin{verbatim}
consts mbind ::\([\iota \ \text{list}, \iota \Rightarrow (o, \sigma)\text{MON}_{SE}] \Rightarrow (o \ \text{list}, \sigma)\text{MON}_{SE}

primrec "mbind [] iostep \(\sigma = \text{Some([], \(\sigma)}\)"
"mbind (a\#H) iostep \(\sigma =
   \text{(case iostep a \(\sigma \text{ of
      None } \Rightarrow \text{Some([], \(\sigma)\)
      |Some (out,\(\sigma')}) \Rightarrow (\text{case mbind H iostep \(\sigma' \text{ of
         None } \Rightarrow \text{Some([out], \(\sigma'})
        |Some(outs,\(\sigma'')) \Rightarrow \text{Some(out#outs,}\(
\sigma''))\)\)\)\)\)\)\)\)

\textbf{Note} that \texttt{mbind} has a slightly different behaviour than \texttt{bind}_{SE} wrt. exceptions!
\end{verbatim}
On this level, we can now state **valid test sequences** as a test specification of the form:

\[
\sigma_0 \models (os \leftarrow (mbind \ i \ s \ ioprog); \ return(post \ os))
\]

where the \(\sigma_0\) 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 \(i s\) 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 :: \(\sigma \Rightarrow (\)bool,\sigma)MON\_SE \Rightarrow bool \ (infix \ \models 15)\)

**where** \(\sigma \models m \equiv (m \sigma \neq \text{None} \land \text{fst}(\text{the} (m \sigma)))\)
Remark: From valid test sequence, HOL-TestGen test were generated by exploring the data-structure *input sequence* $\nu s$ up to given depths $k$ by the standard mechanisms used in unit-tests.

However, it may be convenient to specify constraints on $\nu s$, 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).

$$TP(\nu s) \implies \sigma_0 \models (os \leftarrow (\text{mbind } \nu s \text{ioprog}); \text{return}(\text{post os}))$$
A basic case for the “state transition system specification”: the state-relation-Monad $\text{Mon}_{SB}$ based on the type $(o, \sigma)\text{Mon}_{SB} = \sigma \Rightarrow (o \times \sigma)\text{set}$.

1. It composes relations on states (involving input and output)

2. In case a function evaluation fails (which can be viewed as “an exception occurred”), the execution is stopped and the state remains unchanged (roughly like PROLOG),

3. ... otherwise the execution continues with the new state.

4. $\text{unit}_{SB}$ corresponds to the usual “result” operation.
We define:

1. **definition** \( \text{bind}_{SB} :: [(\alpha, \sigma) \text{MON}_{SB}, \alpha \Rightarrow (\beta, \sigma) \text{MON}_{SB}] \Rightarrow (\beta, \sigma) \text{MON}_{SB} \)
   
   where  
   "\( \text{bind}_{SB} f g \sigma \equiv \bigcup (\lambda (\text{out, } \sigma). (g \text{ out } \sigma)) \ ' \ (f \ \sigma))" 

2. **definition** \( \text{unit}_{SB} :: o \Rightarrow (o, \sigma) \text{MON}_{SB} \)
   
   where  
   "\( \text{unit}_{SB} e \equiv \lambda \sigma. \{(e, \sigma)\}\)"

where we use the syntax

\[
x \leftarrow f ; ; g \ x
\]

for \( \text{bind}_{SB} f (\lambda x.g) \) and returns \( e \) for \( \text{unit}_{SB} e \).
In contrast to $\text{MON}_{\text{SE}}$, the operations of $\text{MON}_{\text{SB}}$ are not executable in general (why?).

On the other hand, concepts like pre- and post conditions can be easily expressed in terms of $\text{MON}_{\text{SB}}$.

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

$$\text{post}(c :: \text{client}, no :: \text{account_no}) =$$

$$\lambda \sigma.\{(\text{result}, \sigma') | \sigma = \sigma' \land \text{result} = \text{the}(\text{register}(c, no))\}$$
Outline

1. Introduction to Sequence Testing
2. Foundation: State-Monads
3. Connecting Specifications and Test-Sequences
4. Test-Case Generation
5. Summing Up
6. Introduction to Reactive Sequence
7. Example: FTP Protocol
Revisiting the Little Bank Example I

**Example:** A Little Bank - Account System.

```plaintext
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]
```

(B. Wolff Uni Paris-Sud)

HOL-TestGen: Model-based Testing

A Tutorial at the LRI 24
In order to formalize input and output implicit in such a specification, such that we can consider it uniformly 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:
   \[(\text{client } \times \text{nat}) \rightarrow \text{int}\]

2. the inputs data-type:

   \[
   \text{datatype } \text{in}_c = \text{deposit} \text{ client account}_\text{no} \text{ nat} \\
   | \text{withdraw} \text{ client account}_\text{no} \text{ nat} \\
   | \text{balance} \text{ client account}_\text{no}
   \]

3. a uniform data-type containing all outputs:

   \[
   \text{datatype } \text{out}_c = \text{deposit}_\text{O} | \text{balance}_\text{O} \text{ nat} | \text{withdraw}_\text{O}
   \]

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 $\sigma$ ($= \text{register}$), $\text{in}_c$ and $\text{out}_c$.

\begin{verbatim}
consts precond :: "register $\Rightarrow \text{in}_c \Rightarrow \text{bool}"
primrec "precond $\sigma$ (deposit c no m) = ((c,no) $\in$ dom $\sigma$)"
"precond $\sigma$ (balance c no) = ((c,no) $\in$ dom $\sigma$)"
"precond $\sigma$ (withdraw c no m) = ((c,no) $\in$ dom $\sigma$
  $\land$ (int m) $\leq$ the($\sigma$(c,no))))"
\end{verbatim}
The post-condition looks as follows:

\textbf{consts} postcond :: "register ⇒ in_c ⇒ out_c × register ⇒ bool"

\textbf{primrec}

"postcond \(\sigma\)(deposit c no m) =
\(\lambda (n,env'). (n = \text{depositO}\)
\(\land \sigma' = \sigma \quad ((c,\text{no})\mapsto \text{the}(\text{env}(c,\text{no})) + \text{int } m))\)"

"postcond \(\sigma\)(balance c no) =
\(\lambda (n,env'). (\sigma = \sigma' \land (\exists x. \text{balanceO } x = n\)
\(\land x = \text{nat}(\text{the}(\sigma(c,\text{no}))))\)"

"postcond \(\sigma\)(withdraw c no m) =
\(\lambda (n,env'). (n = \text{withdrawO}\)
\(\land \sigma' = \sigma \quad ((c,\text{no})\mapsto \text{the}(\text{env}(c,\text{no})) − \text{int } m))\)"
The following combinators — based on the Hilbert-Operator — hold the key for a conversion between monads:

**definition** impl :: \([\sigma, \iota] \Rightarrow \text{bool}, \iota \Rightarrow (o, \sigma)\text{MON}_{SB} \Rightarrow \iota \Rightarrow (o, \sigma)\text{MON}_{SE}\)

where

"impl pre post \iota =
(\lambda \sigma. \text{ if } pre \sigma \iota \text{ then } \text{Some(SOME(out,}\sigma'). \text{ post } \iota \sigma (out,\sigma')) \text{ else arbitrary)}"

**definition** strong_impl :: \([\sigma, \iota] \Rightarrow \text{bool}, \iota \Rightarrow (o, \sigma)\text{MON}_{SB} \Rightarrow \iota \Rightarrow (o, \sigma)\text{MON}_{SE}\)

where

"strong_impl pre post \iota =
(\lambda \sigma. \text{ if } pre \sigma \iota \text{ then } \text{Some(SOME(out,}\sigma'). \text{ post } \iota \sigma (out,\sigma')) \text{ else None)}"
\textbf{definition} is\_strong\_impl :: "[′σ ⇒ ′l ⇒ bool, ′l ⇒ (′o, ′σ)MON_{SB}, ′l ⇒ (′o, ′σ)MON_{SE}] ⇒ bool"

\textbf{where} "is\_strong\_impl pre post ioprog =
(∀ σ l. (¬ pre σ l ∧ ioprog l σ = None) ∨ (pre σ l ∧ (∃ x. ioprog l σ = Some x)))"

This results in the following:

\textbf{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 \textbf{implementable} specification, i.e. the fact that there is a function that maps legal input to output/state pairs, that satisfy the postcondition:

\textbf{definition} \texttt{implementable::} \([\sigma \mapsto \iota \mapsto \text{bool}, \ i \mapsto (o, \sigma)\text{MON}_{SB}] \Rightarrow \text{bool}\)

\textbf{where} \quad "implementable pre post =
\begin{align*}
  \forall \sigma \ i. \ pre \ \sigma \ i \ \rightarrow \ (\exists \ out \ \sigma'. \ post \ \iota \ \sigma \ (out, \sigma'))
\end{align*}"

This results in the following characterization theorem:

\textbf{theorem} \texttt{implementable\_charn:}

"[implementable pre post; pre \ \sigma \ i] \ \Rightarrow \ post \ \iota \ \sigma \ (the(strong\_impl \ pre \ post \ \iota \sigma))"
It is now straight-forward to “convert” our (interface encapsulated) specification into a program. Simply:

\[
\text{strong_impl \ precond \ postcond}
\]

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:

\[ \sigma_0 \models s \leftarrow \text{mbind } S (\text{strong_impl precond postcond}); \text{ return}(s = x) \]

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:

\[(\sigma \vdash (s \leftarrow \text{return } x ; \text{return } (P \ s))) = P \ x\]

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

The introduced case-statements were eliminated in the case-splitting of the test-case-generation phase.
... and the program specific symbolic evaluation rules (where \( H = (\text{strong_implprecond\ postcond}) \)):

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

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

...
Outline

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

\textbf{consts} \texttt{test\_purpose :: "[client, account\_no, \texttt{in}\_\texttt{c} list] \Rightarrow bool"}

\textbf{primrec}

"test\_purpose c no [] = False"
"test\_purpose c no (a\#R) = (\textbf{case} R of
  [] \Rightarrow a = balance c no
  | a'\#R' \Rightarrow (((\exists m. a = deposit c no m) \lor
    (\exists m. a = withdraw c no m)) \land
    test\_purpose c no R))"

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

\((\text{withdraw c no _ | deposit c no _})^* \text{ balance c no}\)
The test-specification is formalized as follows:

```
test_spec test_balance:
assumes account_defined: "(c,no) ∈ dom σ₀"
and test_purpose : "test_purpose c no ι s"
and symbolic_run_yields_x :
  "σ₀ ⊨(s ← mbind ι s (strong_impl precond postcond);
   return (s = x))"
shows " σ₀ ⊨(s ← mbind ι s SUT; return (s = x))"
```
The resulting test-theorem for $k=5$ looks follows:

1. $(\lambda a. \text{Some } 2) \models (s \leftarrow \text{mbind} [\text{balance } ?X1 ?X2] \text{ SUT}; \text{return } s = [\text{balanceO } 2])$
2. THYP ...
3. $(\lambda a. \text{Some } 5) \models (s \leftarrow \text{mbind}
   [\text{deposit } ?X3 ?X4 ?X5, \text{balance } ?X3 ?X4]
   \text{ SUT}; \text{return } s = [\text{depositO, balanceO (nat (5 + int } ?X5))]])$
4. THYP ...
5. THYP
6. int ?X6 $\leq 7 \implies (\lambda a. \text{Some } 7) \models (s \leftarrow \text{mbind}
   [\text{withdraw } ?X7 ?X8 ?X6, \text{balance } ?X7 ?X8] \text{ SUT;}
   \text{return } s = [\text{withdrawO, balanceO (nat (7 \leftarrow \text{int } ?X6))}])$
**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 \((c, no) \in \text{dom } \sigma_0\)). We must assume that a concrete user and accountnumber is defined.

2. **determinism condition** (reflected by the assumption that SUT has type \(\text{in}_c \Rightarrow (\text{out}_c, \text{register}) \text{Mon}_{SE}\)). 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
6. Introduction to Reactive Sequence
7. Example: FTP Protocol
Test-Sequence generation can be formalized as a constraint-resolution problem, too.

Reason: We have data-types (this lists and languages) and Monads in HOL

Test-drivers can be generated as well

Handling of Testability hypotheses implicit (control over the init-state, PUT a function in the sense of the specification)
Outline

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

\[
\text{req}\ ?X \rightarrow \text{port}\ !Y[Y < X] \rightarrow \quad (\text{rec}\ N.\ \text{send}\ !D.\ Y \rightarrow \text{ack} \rightarrow N \\
\quad \Box \text{stop} \rightarrow \text{ack} \rightarrow \text{SKIP})
\]

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

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

\[
\begin{align*}
\text{req?}X & \rightarrow \text{port!}Y[Y < X] \rightarrow \\
& \quad (\text{rec } N. \text{send!}D.Y \rightarrow \text{ack} \rightarrow N \\
& \quad \quad \square \text{stop} \rightarrow \text{ack} \rightarrow \text{SKIP})
\end{align*}
\]

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** ($\text{req } X, \text{ port } Y, \ldots$) in traces to **concrete events** ($\text{req } 4, \text{ port } 2, \ldots$) in runs!
Introduction to Reactive Sequence

Motivation: A Reactive Sequence Example

observer

(manages internal state $\sigma_{obs}$)

$\tau_a, \sigma_c \times \sigma_{obs}$

$ioprog$

(PUT or other observer)

$\tau_c, \sigma_c$

$\theta_a, \sigma'_c \times \sigma'_\text{obs}$
A formal definition looks as follows:

\[
\text{definition observer} :: "[σ_{obs} \Rightarrow o_c \Rightarrow σ_{obs},}
\]
\[
σ_{obs} \Rightarrow l_a \Rightarrow l_c,
\]
\[
σ_{obs} \Rightarrow σ \Rightarrow l_c \Rightarrow o_c \Rightarrow \text{bool}]
\]
\[
\Rightarrow (l_c \Rightarrow (o_c, σ)\text{MONSE})
\]
\[
\Rightarrow (l_a \Rightarrow (o_c, σ_{obs} \times σ)\text{MONSE}) "
\]

\text{where} "observer rebind substitute postcond ioprog \equiv
\]
\[
(λ l_a . (λ (σ_{obs}, σ). \text{let } l_c = \text{substitute } σ_{obs} l_a \text{ in case ioprog } l_c σ \text{ of})
\]
\[
\text{None } \Rightarrow \text{None (\textit{* ioprog failure -- eg. timeout ... *})}
\]
\[
| \text{Some } (o_c, σ') \Rightarrow (\text{let } σ_{obs}' = \text{rebind } σ_{obs} o_c
\]
\[
\text{in if postcond } σ_{obs}' σ' l_c \text{ out}_c
\]
\[
\text{then Some}(o_c, (σ_{obs}', σ'))
\]
\[
\text{else None (\textit{* postcond failure *})} \)"
As can be inferred from the type of observer, the function is 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 distinguish 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 ≡

\[
(\lambda \iota_a. (\lambda (\sigma_{\text{obs}}, \sigma). \\
\text{let } \iota_c = \text{substitute } \sigma_{\text{obs}} \iota_a \\
in \text{ case } \ioprog \iota_c \sigma \text{ of} \\
\quad \text{None } \Rightarrow \text{None (} \ast \ ioprog \text{ failure } - \text{ eg. timeout ...} \\
\quad \mid \text{Some } (o_c, \sigma') \Rightarrow (\text{let } \sigma_{\text{obs}}' = \text{rebind } \sigma_{\text{obs}} o_c \\
in \text{ Some}(o_c, (\sigma_{\text{obs}}', \sigma'))) )
\]"
and ...

**where**  "observer4 postcond ioprog ≡

(λ ν. (λ σ. **case** ioprog νσ **of**

    None ⇒ None (* ioprog failure – eg. timeout ... *)

| Some (o, σ’) ⇒ (if postcond σ’ ν 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 (\( \lambda \) 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. Introduction to Sequence Testing
2. Foundation: State-Monads
3. Connecting Specifications and Test-Sequences
4. Test-Case Generation
5. Summing Up
6. Introduction to Reactive Sequence
7. Example: FTP Protocol
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** InEvent\textsubscript{conc} = req chan | send data chan | stop

**datatype** InEvent\textsubscript{abs} = reqA vars | sendA data vars | stopA

**datatype** OutEvent\textsubscript{conc} = port chan | ack

**datatype** OutEvent\textsubscript{abs} = portA vars | ackA

**types** InEvent = "InEvent\textsubscript{abs} + InEvent\textsubscript{conc}"

**types** OutEvent = "OutEvent\textsubscript{abs} + OutEvent\textsubscript{conc}"

**types** event\textsubscript{abs} = "InEvent\textsubscript{abs} + OutEvent\textsubscript{abs}"
The function substitute 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 occurring 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* $\sigma_{obs}$. 
The function rebind extracts from concrete output events the values and binds them to explicit variables in env. (\(\sigma_{\text{obs}}\))

The predicate rebind only stores occurrences of input-events (marked by ?) in the protocol into the environment; output (!)-occurrences were ignored.

\textbf{consts} rebind :: 
\[\text{[vars } \rightarrow \text{ chan, OutEvent}_{\text{conc}}] \Rightarrow \text{vars } \rightarrow \text{ chan}\]

\textbf{primrec}
\[
\begin{align*}
\text{"rebind env (port n) } &= \text{env(Y } \mapsto \text{n)"} \\
\text{"rebind env OutEvent}_{\text{conc}}.\text{ack } &= \text{env"}
\end{align*}
\]

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 × eventₐ bs 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 :: "eventₐ bs 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**  \( \text{postcond}' :: "((\text{vars} \rightarrow \text{int}) \times \sigma \times \text{InEvent}_c \times \text{OutEvent}_c \times \text{bool}" \)

**recdef**  \( \text{postcond}' \ "\{ \}\" \)
\[
\begin{align*}
\text{postcond}' (\text{env}, _, \text{req} \ n, \text{port} \ m) &= (m \leq n) \\
\text{postcond}' (\text{env}, _, \text{send} \ z \ n, \text{ack}) &= (n = \text{lookup} \ \text{env} \ Y) \\
\text{postcond}' (\text{env}, _, \text{stop}, \text{ack}) &= \text{True} \\
\text{postcond}' (\text{env}, _, \text{y}, \text{z}) &= \text{False} \\
\end{align*}
\]

**constdefs**  \( \text{postcond} :: "(\text{vars} \rightarrow \text{int}) \Rightarrow \sigma \Rightarrow \text{InEvent}_c \Rightarrow \text{OutEvent}_c \Rightarrow \text{bool}" \)
\[
\begin{align*}
\text{postcond} \ \text{env} \ \sigma \ \text{y} \ \text{z} &\equiv \text{postcond}' (\text{env}, \ \sigma, \ \text{y}, \ \text{z}) \\
\end{align*}
\]
**test_spec** "stim_trace $\nu s \implies$

$\langle$empty$[X \mapsto x], ()\rangle$

$s = \langle$os$\leftarrow$mbind $\nu s$(observer$^2$ rebind substitute postcond ioprog));$

result(length $\nu s = \text{length os}$)"

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 $\sigma$, 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 ($()$).
Example: FTP Protocol

1. \([(X \mapsto ?X1], (\))
   \(\models (os \leftarrow \text{mbind} [\text{reqA} X, \text{stop}] (\text{observer}_2 \ \text{rebind substitute postcond ioprog); result(2 = \text{length os})}

3. \([(X \mapsto ?X2], (\))
   \(\models (os \leftarrow \text{mbind} [\text{reqA} X, \text{sendA Data Y}, \text{stop}] (\text{observer}_2 \ \text{rebind substitute postcond ioprog); result(3 = \text{length os})}

5. \([(X \mapsto ?X3], (\))
   \(\models (os \leftarrow \text{mbind} [\text{reqA} X, \text{sendA Data Y}, \text{sendA Data Y}, \text{stop}] (\text{observer}_2 \ \text{rebind substitute postcond ioprog); result(4 = \text{length os})}

7. \([(X \mapsto ?X4], (\))
   \(\models (os \leftarrow \text{mbind} [\text{reqA} X, \text{sendA Data Y}, \text{sendA Data Y}, \text{sendA Data Y}, \text{stop}] (\text{observer}_2 \ \text{rebind substitute postcond ioprog); result(5 = \text{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 \(X\).