# Monadic Sequence Testing and Explicit Test-Refinements

Achim D. Brucker[1(✉)] and Burkhart Wolff[2]

[1] Department of Computer Science, The University of Sheffield, Sheffield, UK
a.brucker@sheffield.ac.uk
[2] LRI, Univ Paris Sud, CNRS, Centrale Suplélec, Université Saclay, Orsay, France
wolff@lri.fr

**Abstract.** We present an abstract framework for sequence testing that is implemented in Isabelle/HOL-TestGen. Our framework is based on the theory of state-exception monads, explicitly modelled in HOL, and can cope with typed input and output, interleaving executions including abort, and synchronisation.

The framework is particularly geared towards symbolic execution and has proven effective in several large case-studies involving system models based on large (or infinite) state.

On this basis, we rephrase the concept of test-refinements for inclusion, deadlock and IOCO-like tests, together with a formal theory of its relation to traditional, IO-automata based notions.

**Keywords:** Monadic sequence testing framework · HOL-TestGen

## 1 Introduction

Automata-based theoretical foundations for test and model-checking techniques are omnipresent; it can be safely stated that a huge body of literature [7,16,19,20,22] uses them as a framework for conceptual argument, comparison, and scientific communication. Usually based on naïve set-theory (in the sense of Halmos [14]) and paper and pencil notations, they proved as a very intuitive and flexible framework. In our view, this omnipresence overshadows the fact that automata theory is a kind of mould into which not everything fits. This is to a lesser extent a burden on the purely theoretical side: naïve set theory is known to be inconsistent, and the sheer number of variants of automata notions makes comparisons more delicate as one might think.

Modelling communication via an automata-product is simple and tempting, but is the resulting CSP-style, synchronous communication paradigm really what we want? The automata-paradigm becomes a problem when it comes to formal, machine-checked presentations and automated reasoning over them. In settings for the latter, underlying set-theories need either to be typed or axiomatised in a system like ZFC [11]. Applications based on *automated reasoning* over these formalisations turn out to be so difficult that successful tool implementations


B.K. Aichernig and C.A. Furia (Eds.): TAP 2016, LNCS 9762, pp. 17–36, 2016.
DOI: 10.1007/978-3-319-41135-4_2

exist only for particular special-cases such as symbolic regular expression representations [17,27]. In our view, it is not a coincidence that implementations of *symbolic* versions of test-systems like TGV (e.g., STG [16]) or SIOCO [12] remained in a prototypical stage.

In this paper, we present an alternative to the automata-paradigm, as far as their application in the field of testing of behavioural models are concerned. Motivated by several projects aiming at the synthesis of test-algorithms for behavioural models with very large and usually infinite state-spaces, we developed a *Monadic Sequence Testing Framework* (MST). It is formalised in Isabelle/HOL and has been used in several major case-studies [2,4,5]. While the framework is *tuned* for mechanised deduction, in particular symbolic execution based on derived rules, it provides a number of theoretic properties which are interesting in its own. MST combines 1. generalised forms of non-deterministic automata with input and output, 2. generalises the concept of Mealy-Machines, 3. generalises the concept of extended finite state machines [13], and 4. generalises some special form of IO Automata, IO LTS's, etc. [19]. Overall, MST shares with [26] the vision of a unified framework for generalising and analysing formalism for symbolic test case generation. Due to shallow representations of programs and pre-post-condition-based program specifications, the MST approach is intrinsic symbolic; no complicated "lifting" of IO Automata or IO LTS's to symbolic versions thereof like IOSTS's is necessary.

We will introduce paper-and-pencil notions for basic automata constructions (Sect. 2), the general concept of test theories (Sect. 3). In Sect. 4, we introduce higher-order logic (HOL) Sect. 4 and sketch our formalization of Sect. 2 in HOL. Finally we introduce our monadic framework, which is demonstrated in Sect. 4.4 on a small example based on an extended infinite automata. In Sect. 5, we generalise the key-concepts of the MST one step further to a formal definition of test-refinements; it is shown that this definition is powerful enough to capture a family of widely known, but up to now unrelated concepts of (sequence) test conformance. We will show that this is of pragmatic interest for proven correct test-optimisations as well as theoretic interest due to its link to IO-automata.

## 2 A Guided Tour on Automata Notions for Testing

In this section, we provide a brief overview of behavioural automata models, focusing on on symbolic versions of automata concepts.

**The Mealy-Machine.** A *Mealy Machine (MM)* [20] is a 6-tuple $(S, S_0, \Sigma_{in}, \Sigma_{out}, T, G)$ consisting of the following: – a finite set of states $S$ – a start state (initial state) $S_0$ which is an element of $S$ – a finite set of, the input alphabet $\Sigma_{in}$ – a finite set of symbols, the output alphabet $\Sigma_{out}$ – a transition function $T : S \times \Sigma_{in} \to S$ mapping pairs of a state and an input symbol to the corresponding next state. – an output function $G : S \times \Sigma_{in} \to \Sigma_{out}$ mapping pairs of a state and an input symbol to the corresponding output symbol. In some formulations, the transition and output functions are coalesced into a single function $T : S \times \Sigma_{in}, \to$

$S \times \Sigma_{out}$. In the literature, also non-deterministic versions are discussed, where the coalesced $T$ has the form $T : S \times \Sigma_{in}, \rightarrow \mathcal{P}(S \times \Sigma_{out})$. Mealy machines are related to Moore machines [21] which are equivalent. If the finiteness constraints are removed, one speaks of a Generalised Mealy Machine (GMM).

**The Deterministic Automata.** The *deterministic finite automaton* (DFA) $M$ is a 5-tuple, $(S, S_0, \Sigma, T, F)$, consisting of – a finite set of states $S$, – an initial or start state $S_0 \in S$, – a finite set of symbols, the alphabet $\Sigma$, – a transition function $T : S \times \Sigma \rightarrow S$, and – a set of accept states $F \subseteq S$. If the finiteness-constraints are lifted, we speak of a *deterministic automaton* (DA). If $T$ is generalised to a relation $\mathcal{P}(S \times \Sigma \rightarrow S)$, one speaks of a non-deterministic finite automaton (NDFA) or a non-deterministic automaton (NDA) respectively. If the alphabet $\Sigma$ is structured as a set of pairs $\Sigma_{in} \times \Sigma_{out}$ of input-and output labels, we speak of *input-output-tagging* of the automata versions. The astute reader will notice that input-output-tagged NDFA's and NDA's can be mapped to generalised Mealy machines GMM and vice versa.

The interest into symbolic versions of these automata notions was raised surprisingly recently: Veanes et al. presented Finite Symbolic Automata as a tool (REX [27]) and investigated their theoretic properties [9].

**The Input/Output Automata.** Input-output labelled transition systems are going back to the notions of Lynch and Tuttle [19]. This line of automata definitions, which were later on referred as "labelled transition systems," emphasises the annihilation of the difference between input and output to enable some form of asynchronous communication between tester and the system under test SUT as well as some rudimentary form of time (the concept supports *silent* $\tau$ *actions* to express time elapsing while some *internal* action in the machine is performed). The theory supports in principle that a SUT can non-deterministically decide either to accept input or to emit output; in practical testing scenarios, this possibility is usually ruled out. Formally, an IO-automata is defined as a 5-tuple $(S, S_0, \Sigma, T, Task)$ consisting of: – a (not necessarily finite) set of states $S$, – a start state (initial state) $S_0$ which is an element of $S$, – an alphabet, the *signature* $\Sigma$ which is partitioned into three disjoint sets of symbols $\Sigma = in_{\mathrm{IOA}} \cup out_{\mathrm{IOA}} \cup out_{\mathrm{IOA}} \cup int_{\mathrm{IOA}}$ are called *input* actions, *output* actions, and *internal* actions, – a transition *relation* $T \subseteq S \times \Sigma \times S$, and – a task-partition *Task* which is defined as an equivalence relation on $out_{\mathrm{IOA}} \cup out_{\mathrm{IOA}} \cup int_{\mathrm{IOA}}$. In contrast to input-output tagged NDA's, where $\Sigma$ is the *Cartesian product* of input and output, IO Automata construct $\Sigma$ as *disjoint union*.

The task partition is used to define fairness conditions on an execution of the automaton. These conditions require the automaton to continue giving fair turns to each of its tasks during its execution. This component of the original formulation is often dropped and replaced by other ones in related approaches [15,24].

**Symbolic IO Transition Systems.** A Symbolic IO Transition System (IOSTS) [22] is a tuple $(D, \Theta, S, S_0, \Sigma, T)$ where – $D$ is a finite set of typed

data, partitioned into a set $V$ of variables and a set $P$ of parameters. For $d \in D$, type($d$) denotes the type of $d$. – $\Theta$ is the initial condition, a Boolean expression on $V$, – $S$ is a nonempty, finite set of states and $S_0 \in S$ is the initial state. – $\Sigma$ is a nonempty, finite set of symbols, which is the disjoint union of a set $\Sigma?$ of input actions and a set $\Sigma!$ of output actions. For each action $a \in \Sigma$, its signature $sig(a) = (p_1, \ldots, p_k) \in P^k (k \in \mathbb{N})$ is a tuple of parameters. – $T$ is a set of transitions. Each transition is a tuple $(s, a, G, A, s)$ made of: a location $s \in S$, called the origin of the transition, an action $a \in \Sigma$, called the action of the transition, a Boolean expression $G$ on $V \cup sig(a)$, called the guard, an assignment $A$, which is a set of expressions of the form $(x := A^x)_{x \in V}$ such that, for each $x \in V$, the right-hand side $A^x$ of the assignment $x := A^x$ is an expression on $V \cup sig(a)$, a location $s \in S$ called the destination of the transition. Similar attempts to generalise IO Automata to symbolic versions of IO-LTL's are [12].

**Extended Finite State Machines.** An extended finite state machine (EFSM) [7] is a 7-tuple $M = (S, D, I, O, F, U, T)$ where – $S$ is a set of symbolic states, – $I$ is a set of input symbols, – $O$ is a set of output symbols, – $D$ is an n-dimensional linear space $D_1 \times \cdots \times D_n$, – $F$ is a set of enabling functions $f_i : D \to \{0, 1\}$, – $U$ is a set of update functions $u_i : D \to D$, – $T$ is a transition relation, $T : S \times F \times I \to S \times U \times O$ EFSM's have been motivated from the very beginning by (symbolic) testing techniques [7].

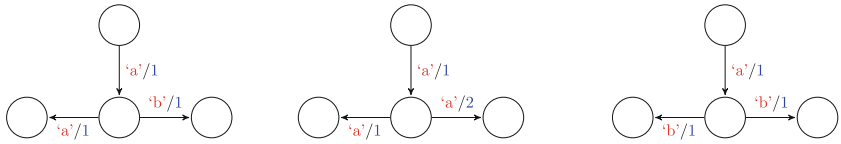
Many variants have been discussed in the literature that attempt to give a concrete syntax (e.g., a term-language, just assignments) for $F$ and $U$; however, we will refrain from this and try to keep our MST framework abstract on the level of functions and not their syntactic representations.

**Some Common Notions of Automatons.** We distinguish the notion of a *trace*: *Traces*($A$) contains the set of lists of symbols $[a_1, a_2, a_3, \ldots]$ in $A$ (which is an arbitrary automaton DA, NDA, DFA, NDFA, ...), which describe a path in $A$. Here, we consider the case of an ESFM similar to an input-output tagged DA or NDA. A *run* is a list of triples $[(s_1, a_1, s_2), (s_2, a_2, s_3), \ldots]$ which describes a path in $A$; *Run*($A$) contains the set of runs in A. With $States_A(t)$ we denote the set of reachable states after a trace $t \in Trace(A)$. If $t \in Trace(A)$ ($A$ is an input-output tagged DA or NDA, IO Automaton, IOSTS, EFSM), we denote with $In_A(t)$ the set of possible input symbols after $t$; with $Out_A(t)$ the set of possible output symbols. We call an automaton *IO-deterministic*, iff for each trace $t \in Trace(A)$, there is at most one reachable state after $t : |States_A(t)| \leq 1$.

For automata $A$ (which is again an input-output tagged DA or NDA, IO Automaton, IOSTS, EFSM), we define the notion of input-sequences of a trace as projection of traces into its input components: if $t = [(i_1, o_1), (i_2, o_2), (i_3, o_3), \ldots]$ is in $Trace(A)$, then $[i_1, i_2, i_3, \ldots]$ is the corresponding input-sequence of $t$.

In other words, the relation between a sequence of input-output pairs and the resulting system state must be a function.

There is a large body of theoretical work replacing the latter testability hypothesis by weaker or alternative ones (and avoiding the strict alternates of




(a) IO-Deterministic SUT. (b) IO-Deterministic SUT. (c) Non-IO-Deterministic SUT.

**Fig. 1.** IO-Determinism and Non-IO-Determinism

input and output, adding asynchronous communication between tester and SUT, or adding some notion of time), but most practical approaches do assume it as we do throughout this paper. There are approaches (including our own [3]) that allow at least a limited form of access to the final (internal) state of the SUT.

A sequence of input-output pairs through an automaton $A$ is called a *trace*, the set of traces is written $Trace(A)$. The function $In$ returns for each trace the set of inputs for which $A$ is enabled after this trace; in Fig. 1c for example, $In$ [("$a$", 1)] is just {"$b$"}, in Fig. 1a, just {"$a$", "$b$"}. Dually, $Out$ yields for a trace $t$ and input $\iota \in In(t)$ the set of outputs for which $A$ is enabled after $t$; in Fig. 1b for example, $Out$([("$a$", 1)],"$a$") this is just $\{1, 2\}$.

## 3 A Gentle Introduction to Sequence Testing Theory

Sequence testing is a well-established branch of formal testing theory having its roots in automata theory. The methodological assumptions (sometimes called *testability hypothesis* in the literature) are summarised as follows:

1. The tester can reset the system under test ($SUT$) 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,
3. the SUT behaves deterministic with respect to an observed sequence of input-output pairs (it is *IO-deterministic*).

The latter two assumptions assure the reproducibility of test executions. The latter condition does *not* imply that the SUT is deterministic: for a given input $\iota$, and in a given state $\sigma$, the SUT may non-deterministically choose between the successor states $\sigma'$ and $\sigma''$, provided that the pairs $(o', \sigma')$ and $(o'', \sigma'')$ are distinguishable. Thus, a SUT may behave non-deterministically, but must make its internal decisions observable by appropriate output.

Equipped with these notions, it is possible to formalise the intended *conformance relation* between a system specification (given as automaton SPEC labelled with input-output pairs) and a SUT. The following notions are known in the literature: – *inclusion conformance* [18]: all traces in SPEC must be possible in SUT, – *deadlock conformance* [10]: for all traces $t \in Traces$(SPEC) and $b \notin In(t)$, $b$ must be refused by SUT, and – *input/output conformance (IOCO)* [25]: for all traces $t \in Traces$(SPEC) and all $\iota \in In(t)$, the observed output of the SUT must be in $Out(t, \iota)$.

# 4 Monadic Sequence Testing Framework

## 4.1 Higher-Order Logic and Isabelle/HOL

*Higher-order logic* (HOL) [1,8] is a classical logic based on a simple type system. Types have been extended by Hindley/Milner style polymorphism: they consist of type variables $'\alpha, '\beta, '\gamma, \ldots$ and and type constructors such as $\_ \Rightarrow \_$, _set, _list, $\_ \times \_$, $\_ + \_$, bool, nat, ... (for function space, typed sets, lists, Cartesian products, disjoint sums, Boolean, natural numbers, etc.) with type classes similar to Haskell: $('\alpha :: \text{linorder})\text{list}$ constrains the set of possible types, for example, to those types that posses an ordering symbol which satisfies the properties of a linear order. The simple-typed $\lambda$-calculus underlying Isabelle enforces that any $\lambda$-expression $e$ must be typed by a type-expression $\tau$; we write $e :: \tau$ for $e$ is well-typed and has type $\tau$. Being based on a polymorphically typed $\lambda$-calculus, HOL can be viewed as a combination of a programming language such as SML or Haskell, and a specification language providing powerful logical quantifiers ranging over elementary and function types.

HOL provides the usual logical connectives, e.g., $\_ \wedge \_$, $\_ \rightarrow \_$, $\neg \_$ as well as the object-logical quantifiers $\forall x.\ P\,x$ and $\exists x.\ P\,x$; in contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centred around extensional equality $\_ = \_ :: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$.

Isabelle/HOL offers support for extending theories in a logically safe way: a theory extension is *conservative* if the extended theory is consistent provided that the original theory was consistent. Conservative extensions can be *constant definitions*, *type definitions*, *datatype definitions*, *primitive recursive definitions* and *well-founded recursive definitions*.

For example, the polymorphic option-type is defined as:

**datatype** $'\alpha$ option = None | Some(the:$'\alpha$)

which implicitly introduces the constructors None and Some, the selector the as well as a number of lemmas over this data-type (e.g., None ≠Some x,the (Some x)= x, induction). The option type is also used to model partial functions $'\alpha \rightharpoonup '\beta$ which is synonym to $'\alpha \Rightarrow '\beta$ option.

## 4.2 Formal Presentations of Automata: Direct Approach

A record of $n$ fields is an $n$-ary Cartesian where the components have names. Equipped with this machinery, it is for example simple to formalise the concepts of, e.g., NA, NDA, ESFM, as introduced semi-formally in Sect. 2. For example, we can define NA by:

```
record ('α, 'σ) DA =  init   :: "'σ"
                      step   :: "'σ ×'α ⇒'σ"
                      accept :: "'σ set"
```

The record specification construct implicitly introduces constructor functions (we may write ⦇init = 0, step = λ(s,a). s + a mod 4, accept = {1,3}⦈ for a an

deterministic automaton implicitly typed: (nat, nat) DA) as well as selector and update functions enabling us to write: $da(\!|\text{accept} = (\text{accept } da) - \{1\}\ |\!)$.

Constraining the (general, infinite) DA to the more common DFA is straightforward: One can define the type-class of "all finite types" in Isabelle/HOL by

**class** fin = **assumes** finite: " finite ({x::'$\alpha$ set. True})"

where the carrier-set of a type '$_\alpha$ is restricted to be finite ( finite is a library concept). Thus, it is possible to formalise the DFA by adding type class constraints such as ('$\alpha$:: fin ,' $\sigma$ :: fin )DA.

### 4.3 Formal Presentations of Automata: The Monadic Approach

As shown before, the obvious way to model the state transition relation $T$ of an NDA is by a relation of the type $(\sigma \times (\iota \times o) \times \sigma)$ set, (or, for the case of the partial DA: $(\sigma \times \iota(\rightharpoonup o \times \sigma)$ option). Now, types can be *isomorphic*, i.e. there exists a bijection of the underlying carrier-sets. This is the case for types like '$\alpha$×'$\beta$ to '$\beta$×'$\alpha$ (Cartesian isomorphism) as well as: '$\alpha$×'$\beta$⇒'$\gamma$ to '$\alpha$⇒'$\beta$⇒'$\gamma$ (Currying) as well as '$\alpha$ set to '$\alpha$⇒bool (foundational in HOL). Thus, one can also model the transition relation isomorphically via:

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

or for a case of a partial deterministic transition function:

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

In a theoretic framework based on classical higher-order logic (HOL), the distinction between "deterministic" and "non-deterministic" is actually much more subtle than one might think, and a more detailed discussion is necessary here. First, even in an (infinite) DA setting where the transition is a *function*, the modelled SUT is not necessarily deterministic with respect to its *input sequence*, as the difference between Fig. 1b and c reveals. Actually, provided that sufficient information can be drawn from the output (recall that we assume the SUT to be input-output deterministic), an arbitrary pre-post-condition style specification modelling the input-output relation of a system transition is possible. This is the "usual" kind of non-determinism we need in a specification of a program. We argue therefore that a framework like IOLTS, where systems may non-deterministically decide to accept input or to omit output were an overgeneralisation of little use. Second, a transition function can be under-specified via the Hilbert-choice operator built-in the HOL-logic and ZFC. This classical operator, written SOME x. P(x) chooses an arbitrary element $x$ for which $P$ holds true. We can only infer for y = SOME x. x∈{a,b,c} that $y$ must be $a$ or $b$ or $c$.

From the above said, it follows that transition function $T$ in NA or NDA can be isomorphically represented by:

$$step\ \iota\ \sigma = \{(o, \sigma') | \text{post}(\sigma, o, \sigma')\}$$

or respectively:

$$step\ \iota\ \sigma = \text{Some}(\text{SOME}(o, \sigma').\ \text{post}(\sigma, o, \sigma'))$$

for some post-condition post. In the former "truly non-deterministic" case *step* can and will at run-time choose different results, the latter "under-specified deterministic" version will decide in a given model always the same way: a choice that is, however, unknown at specification level and only declaratively described via post. For many systems (like system scheduler [4], processor models [3], etc.) it was possible to opt for an under-specified deterministic stepping function. The generalisation to a *partial* deterministic transition paves the way to cover EFSM's; their enabling function $F$, practically equivalent of a pre-condition of the transition, can be represented in a partial function by their non-applicability: $F(x) \equiv x \notin \mathrm{dom}(step)$.

We abbreviate functions of type $\sigma \Rightarrow (o \times \sigma)$ set or $\sigma \Rightarrow (o \times \sigma)$ option $\mathrm{MON}_{\mathrm{SBE}}(o, \sigma)$ or $\mathrm{MON}_{\mathrm{SE}}(o, \sigma)$, respectively; thus, the aforementioned state transition functions of NDA and DA can be typed by $\iota \to \mathrm{MON}_{\mathrm{SBE}}(o, \sigma)$ for the general and $\iota \to \mathrm{MON}_{\mathrm{SE}}(o, \sigma)$ for the deterministic setting.

If these function spaces were extended by the two operations *bind* and *unit* satisfying three algebraic properties, they form the algebraic structure of a *monad* that is well known to functional programmers as well as category theorists. Popularised by [28], monads became a kind of standard means to incorporate stateful computations into a purely functional world.

Throughout this paper, we will choose as basis for our Monadic Testing Framework under-specified deterministic stepping functions. Consequently, we will concentrate on the $\mathrm{MON}_{\mathrm{SE}}(o, \sigma)$ monad which is called the *state-exception monad* in the literature.

The algebraic structure of a Monad comes with two operations *bind* and *unit*; like functional or relational compositions $f \circ g$ resp. $ROS$, *bind* can be seen as the "glue" between computations, while *unit* represents a kind of neutral element. *bind* generalizes sequential composition by adding value passing; together with *unit*, which embeds a atomic value into a computation, it can be defined for the special-case of the state-exception monad in HOL as follows:

```
definition bindSE :: "('o,'σ)MONSE ⇒('o ⇒('o','σ)MONSE) ⇒('o','σ)MONSE"
where      "bindSE f g = (λσ. case f σ of None ⇒None
                                        | Some (out, σ') ⇒ g out σ')"
definition unitSE :: "'o ⇒('o, 'σ)MONSE" ("(return _)" 8)
where      "unitSE e = (λσ. Some(e,σ))"
```

Generalizing $f \circ g$, $\mathrm{bind}_{\mathrm{SE}}$ takes input and output also into account (in the sense that a later computation may have the output of prior computations as input, and that a prior computation may fail (case None in the case distinction). Following Haskell notation, we will write $x \leftarrow m_1; m_2$ equivalently for $\mathrm{bind}_{\mathrm{SE}} m_1 (\lambda x. m_2)$. Moreover, we will write return for $\mathrm{unit}_{\mathrm{SE}}$.

This definition of $\mathrm{bind}_{\mathrm{SE}}$ and $\mathrm{unit}_{\mathrm{SE}}$ satisfy the required monad laws:

```
bind_left_unit:   (x ← return c; P x) = P c
bind_right_unit:  (x ← m; return x) = m
bind_assoc:       (y ← (x ←m; k x); h y) = (x ←m; (y ←k x; h y))
```

The concept of a *valid monad execution*, written $\sigma \models m$, can be expressed as follows: an execution of a monad computation $m$ of type $(\mathrm{bool}, \sigma)$ $\mathrm{MON}_{\mathrm{SE}}$ is

valid iff its execution is performed from the initial state $\sigma$, no exception occurs and the result of the computation is true. More formally, $\sigma \models m$ holds iff $(m\ \sigma \neq \text{None} \wedge \text{fst}(\text{the}(m\ \sigma)))$, where fst and snd are the usual *first* and *second* projection into a Cartesian product.

We define a *valid test-sequence* as a valid monad execution of a particular format: it consists of a series of monad computations $m_1 \ldots m_n$ applied to inputs $\iota_1 \ldots \iota_n$ and a post-condition $P$ wrapped in a return depending on observed output. It is formally defined as follows:

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

Since each individual computation $m_i$ may fail, the concept of a valid test-sequence corresponds to a feasible path in an NDA, (partial) DA, ESFM or a GMM, that leads to a state in which the observed output satisfies $P$.

The notion of a valid test-sequence has two facets: On the one hand, it is executable, i.e., a *program*, iff $m_1, \ldots, m_n, P$ are. Thus, a code-generator can map a valid test-sequence statement to code, where the $m_i$ where mapped to operations of the SUT interface. On the other hand, valid test-sequences can be treated by a particular simple family of symbolic executions calculi, characterised by the schema (for all monadic operations $m$ of a system, which can be seen as the its step-functions):

$$\frac{}{(\sigma \models \text{return}P) = P} \qquad \frac{C_m\ \iota\ \sigma \qquad m\ \iota\ \sigma = \text{None}}{(\sigma \models ((s \leftarrow m\ \iota; m'\ s))) = \text{False}} \tag{1}$$

$$\frac{C_m\ \iota\ \sigma \qquad m\ \iota\ \sigma = \text{Some}(b, \sigma')}{(\sigma \models s \leftarrow m\ \iota; m'\ s) = (\sigma' \models m'\ b)} \tag{2}$$

This kind of rules is usually specialised for concrete operations $m$; if they contain pre-conditions $C_m$ (constraints on $\iota$ and state), this calculus will just accumulate them and construct a constraint system to be treated by constraint solvers used to generate concrete input data in a test.

### 4.4 Example: Bank

To present the effect of the symbolic rules during symbolic execution, we present a model of toy bank that allows for checking the account balance as well as for depositing and withdrawing money. State of the bank system is modelled as a map from client and account information to the account balance:

```
type_synonym client      = string
type_synonym account_no  = int
type_synonym data_base   = (client × account_no) ⇀ int
```

Our Bank example provides only three input actions for checking the *balance* as well as *deposit* and *withdraw* money. Our model can be viewed as a transaction system, in which a series of atomic operations caused by different subjects can be executed in an interleaved way.

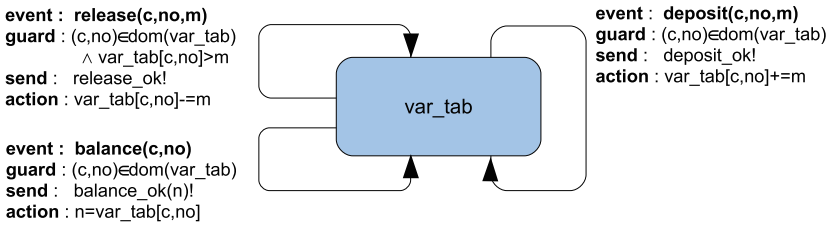



**Fig. 2.** SPEC: An Extended Finite State Machine for they toy Bank

**datatype** in_c = deposit client account_no nat
| withdraw client account_no nat
| balance client account_no

The output symbols are:

**datatype** out_c = deposit_ok | withdraw_ok | balance_ok nat

Figure 2 shows an extended finite state-machine (EFSM), the operations of our system model SPEC. A transcription of an EFSM to HOL is straight-forward and omitted here. However, we show a concrete symbolic execution rule derived from the definitions of the SPEC system transition function, e.g., the instance for Eq. 2:

$$\frac{(c, no) \in \mathrm{dom}(\sigma) \qquad \mathrm{SPEC}\ (\mathrm{deposit}\ c\ no\ m)\ \sigma = \mathrm{Some}(\mathrm{deposit\_ok}, \sigma')}{(\sigma \models s \leftarrow \mathrm{SPEC}\ (\mathrm{deposit}\ c\ no\ m); m'\ s) = (\sigma' \models m'\ \mathrm{deposit\_ok})}$$

where $\sigma = var\_tab$ and $\sigma' = \sigma((c, no) := (\sigma(c, no) + m))$. Thus, this rule allows for computing $\sigma$, $\sigma'$ in terms of the free variables $var\_tab$, $c$, $no$ and $m$. The rules for withdraw and balance are similar. For this rule, SPEC (deposit $c$ $no$ $m$) is the concrete stepping function for the input event deposit $c$ $no$ $m$, and the corresponding constraint $C_{\mathrm{SPEC}}$ of this transition is $(c, no) \in \mathrm{dom}(\sigma)$.

The symbolic execution is deterministic in the processing of valid test-sequences and computes in one sweep all the different facets: checking enabling conditions, computing constraints for states and input and computing symbolic representations for states and output. Since the core of this calculus is representable by a matching process (rather than a unification process), the deduction aspects can be implemented in systems supporting HOL particularly efficiently.

**A Simulation of Test-Driver Generation by Symbolic Execution.** We state a family of test conformance relations that link the specification and abstract test drivers. The trick is done by a coupling variable *res* that transport the result of the symbolic execution of the specification SPEC to the attended result of the SUT.

$$\begin{aligned} \sigma \ &\models o_1 \leftarrow \mathrm{SPEC}\ \iota_1; \ldots; o_n \leftarrow \mathrm{SPEC}\ \iota_n; \mathrm{return}(res = [o_1 \cdots o_n]) \\ \longrightarrow \sigma \ &\models o_1 \leftarrow \mathrm{SUT}\ \iota_1; \ldots; o_n \leftarrow \mathrm{SUT}\ \iota_n; \mathrm{return}(res = [o_1 \cdots o_n]) \end{aligned}$$

Successive applications of symbolic execution rules allow to reduce the premise of this implication to $C_{\text{SPEC}}\ \iota_1\ \sigma_1 \longrightarrow \ldots \longrightarrow C_{\text{SPEC}}\ \iota_n\ \sigma_n \longrightarrow res = [a_1 \cdots a_n]$ (where the $a_i$ are concrete terms instantiating the bound output variables $o_i$), i.e., the constrained equation $res = [a_1 \cdots a_n]$. The latter is substituted into the conclusion of the implication. In our previous example, case-splitting over input-variables $\iota_1$, $\iota_2$ and $\iota_3$ yields (among other instances) $\iota_1 = \text{deposit}\ c_1\ no_1\ m$, $\iota_2 = \text{withdraw}\ c_2\ no_2\ n$ and $\iota_3 = \text{balance}\ c_3\ no_3$, which allows us to derive automatically the constraint:

$$(c_1, no_1) \in \text{dom}(\sigma) \longrightarrow (c_2, no_2) \in \text{dom}(\sigma') \wedge n < \sigma'(c_2, no_2) \longrightarrow$$
$$(c_3, no_3) \in \text{dom}(\sigma'') \longrightarrow res = [\text{alloc\_ok}, \text{release\_ok}, \text{status\_ok}(\sigma''(c_3, no_3)]$$

where $\sigma' = \sigma((c_1, no_1) := (\sigma(c_1, no_1) + m)))$ and $\sigma'' = \sigma'((c_2, no_2) := (\sigma(c_2, no_2) - n)))$.

In general, the constraint $C_{\text{SPEC}_i}\ \iota_i\ \sigma_i$ can be seen as an *symbolic abstract test execution*; instances of it (produced by a constraint solver such as Z3 integrated into Isabelle) will provide concrete input data for the valid test-sequence statement over SUT, which can therefore be compiled to test driver code. In our example here, the witness $c_1 = c_2 = c_3 = 0$, $c_1 = c_2 = c_3 = 5$, $m = 4$ and $n = 2$ satisfies the constraint and would produce (predict) the output sequence $res = [\text{deposit\_ok}, \text{withdraw\_ok}, \text{balance\_ok}\ 2]$ for SUT according to SPEC. Thus, a resulting (abstract) test-driver is:

$$\sigma \models o_1 \leftarrow \text{SUT}\ \iota_1; o_2 \leftarrow \text{SUT}\ \iota_2; o_3 \leftarrow \text{SUT}\ \iota_3;$$
$$\text{return}([\text{alloc\_ok}, \text{release\_ok}, \text{status\_ok}\ 2] = [o_1, o_2, o_3])$$

A code-generator setup of HOL-TestGen compiles this abstract test-driver to concrete code in C (for example), that is linked to the real SUT implementation.

**Experimental Results Gathered from the Example.** The traditional way to specify a sequence test scenario in HOL-TestGen looks like this:

**test_spec** test_balance:
**assumes** account_def : "($c_0$,no) ∈dom $\sigma_0$"
and accounts_pos : "init $\sigma_0$" and test_purpose : "test_purpose $c_0$ no S"
and sym_exec_spec : "$\sigma_0 \models$(s ←mbind$_{\text{FailStop}}$ S SYS; return (s = x))"
**shows** "$\sigma_0 \models$(s ←mbind$_{\text{FailStop}}$ S PUT; return (s = x))"

where the assumptions of this scenario (also called *test purposes*) are:

- account_def that the initial system state $\sigma_0$ is a map that contains at least a client $c_O$ with an account no,
- the constraint $\sigma_O$ constrains the tests to those $\sigma_O$ where all accounts have a positive balance, and
- test_purpose constrains the set of possible input sequences S to those that contain only operations of client $c_)$ and two of his accounts.

We skip the formal definitions of init and test_purpose due to space reasons.

Using explicit test-refinement statements as introduced in Sect. 5, we can state the above scenario equivalently as as inclusion test as follows:

```
test_spec test_balance3:
"PUT ⊑IT(σ. init σ∧(c0,no) ∈dom σ},{ιs. test_purpose c0 no ιs}) SYS"
apply(rule inclusion_test_I_opt, simp, erule conjE)                    (* 1 *)
using[[no_uniformity]] apply(gen_test_cases 4 1 "PUT")                 (* 2 *)
apply(tactic "ALLGOALS(TestGen.REPEAT'(ematch_tac
  [@{thm balance.exec_mbindFStop_E},@{thm withdraw.exec_mbindFStop_E},
   @{thm deposit.exec_mbindFStop_E},@{thm valid_mbind'_mt}]))")        (* 3 *)
apply(auto simp: init_def)                                             (* 4 *)
using[[no_uniformity=false]]
apply(tactic "ALLCASES(uniformityI_tac @{context} [\"PUT\"])")         (* 5 *)
mk_test_suite "bank_simpleNB3"                                         (* 6 *)
(* ... *)
gen_test_data "bank_simpleNB3"                                         (* 7 *)
```

The HOL-TestGen generation process in itself has been described in detail in [6] to which the interested reader is referred. For space reasons, we can only highlight the above test-generation script in the Isar language. It starts with the stages of a test generation from the explicit test-refinement statement over elementary massage involving the test optimisation theorem inclusion_test_I_opt (see Sect. 5) labelled *(* 1 *)*, the splitting-phase of the input sequence labelled *(* 2 *)*, the symbolic execution phase labelled *(* 3 *)*, a simplification of the resulting constraints in *(* 4 *)*, the separation of the constraint systems and test-hypothesises *(* 5 *)* and the generation of the resulting test-theorem. Recall that a test-theorem captures both abstract test-cases and test-hypothesises and links them to the original test specification (see [6]). In *(* 6 *)*, an internal data-structure called *test container*—named "bank_simpleNB3" where this choice has no particular importance—is created into which the test-theorem is stored.

The call of the command **gen_test_data** performs the test-data selection phase (in our example by using Z3) for the test-container "bank_simpleNB3", i.e. it converts abstract test cases in concrete tests by finding ground solutions for the constraints in the abstract test cases. We omit the further phases that compile the test cases to concrete test-oracles in C, which were linked to the implementation of PUT which is just an uninterpreted constant in this specification.

For example, we pick from the list of the abstract test cases:

$$\forall\, x \in \mathrm{dom}\ \sigma_0.\ 0 \leq \mathrm{the}(\sigma_0\ x) \longrightarrow \sigma_0\ (c_0, no) = \mathrm{Some}\ y \longrightarrow \mathrm{int}\ n' \leq y + \mathrm{int}\ n \longrightarrow$$
$$\sigma_0 \models \mathrm{os} \leftarrow \mathrm{mbind}_{\mathrm{FailStop}}[\mathrm{deposit}\ c_0\ no\ n,\ \mathrm{withdraw}\ c_0\ no\ n',\ \mathrm{balance}\ c_0\ no]\ \mathtt{PUT};$$
$$\mathrm{unit}_{\mathrm{SE}}(os = [\mathrm{deposit\_ok}, \mathrm{withdraw\_ok}, \mathrm{balance\_ok}(\mathrm{nat}(y + \mathrm{int}\ n - \mathrm{int}\ n'))])$$

This abstract test case says: for any $\sigma_0$ which has only positive values, and a y with the balance of the account of client $c_0$ on his account no, and sufficient money on the account such that the deposit and withdraw operations can both be effectuated (mind the precondition of withdraw that the balance must be sufficiently large for the withdraw), a test-sequence deposit-withdraw-balance must lead to the observable result that all three operations succeed and produce the

result value nat(y + int n − int n'), where nat and int are HOL-library coersions between nats and integers. They are a result of our operations in the model that requires at some points natural numbers and at integers on others; this kind of complication is very common in constraints generated from programs or models.

The test-selection phase chooses, e.g., the following concrete tests from the abstract test shown above:

$$(\lambda a.\ \text{Some } 15) \models \text{os} \leftarrow \text{mbind}_{\text{FailStop}}\ [\text{deposit } c_0\ 6\ (\text{nat } 17), \text{withdraw } c_0\ 6\ (\text{nat } 30), \text{balance } c_0\ 6]\ \texttt{PUT};\ \text{unit}_{\text{SE}}\ (\text{os} = [\text{deposit\_ok}, \text{withdraw\_ok}, \text{balance\_ok}(2)]))$$

This concrete test states: if we start with a system state where any account of any client has the balance 15, then we can run on `PUT` the sequence: deposit 17 for client $c_0$ on his account no 6, withdraw 30, and we should observe that all three operations went well and the result of the final one is 2. This concrete test is now finally a computable function, i.e. a *program*; the reader interested in the technical process that compiles it into a test driver in C is referred to `Bank.thy` in the HOL-TestGen distribution.

In the following, we are interested in a few experimental measurements that we did on a conventional laptop with 2.5 GHz i7 processor and 16 Mb Ram, using Isabelle/HOL-TestGen version 1.8.0. We omit the phases *(∗ 1 ∗)* and the test-oracle generation, which were more or less constant and small in the experimental range. We vary over the first parameter of the test-splitting phase, which is 4 in the above test-script and $n$ in the following. It defines the length of the input sequences that were result of the splitting. Since we have 3 different input events in our model (deposit, withdraw, balance), the space of abstract test-cases grows asymptotically with this length by $3^n$. We count the number of seconds and the number of abstract/concrete tests found (see Table 1).

**Table 1.** Run-time and number of test cases of the bank example.

| | *(∗ 2 ∗)* | | *(∗ 3 ∗)* | | *(∗ 4 ∗)* | | *(∗ 5 ∗)* | | *(∗ 7 ∗)* | |
|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | sec | no | sec | no | sec | no | sec | no | sec | no |
| 3 | $15.1 \cdot 10^0$ | 7 | 0.9 | 7 | 0.1 | 7 | 0.8 | 7 | 0.7 | 7 |
| 4 | $63.3 \cdot 10^0$ | 15 | 1.7 | 15 | 2.1 | 15 | 2.5 | 15 | 1.8 | 15 |
| 5 | $7.2 \cdot 10^3$ | 42 | 6.1 | 42 | 10.3 | 42 | 28.0 | 42 | 3.8 | 42 |
| 6 | $> 88.0 \cdot 10^3$ | - | - | - | - | - | - | - | - | - |

The splitting phase was not optimised—this is what we usually do in larger case-studies, where we use a number of switches and screws in HOL-TestGen to basically prune the splitting process early[1]. The standard pruning catches already the constraint stemming from test_purpose that a balance-operation has

[1] The core-example of [4] can be decomposed into 70000 abstract test-cases in less than two hours on a conventional laptop in HOL-TestGen [6].

to appear at the end and that clients and account numbers are restricted; this explains why the abstract tests indicated here are below $3^n$. Note furthermore that the example is somewhat atypical since the generated abstract tests are all feasible and all together represent an easy game for the constraint solver.

## 5 A Formal Theory on Conformance Relations

This schema of a test-driver synthesis can be refined and optimised: we show three examples of the formalisation of conformance relations as well as formal proofs of their connection possible in our framework. All notions and lemmas mentioned here are formally proven in Isabelle/HOL.

**Preliminaries and Observations.** First, for iterations of stepping functions an mbind operator can be defined, which is basically a fold over $\text{bind}_{\text{SE}}$. It takes a list of inputs $\iota s = [i_1, \ldots, i_n]$, feeds it subsequently into SPEC and stops when an error occurs. The standard definition looks as follows:

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

When generalising $\text{bind}_{\text{SE}}$ to sequences of computations over an input sequence, three different variants are possible:

1. The *failsave* mbind (our default; written $\text{mbind}_{\text{FailSave}}$ if necessary). This operator has a similar semantics than a sequence of method-calls in Java with a catch-clause at the end: If an exception occurs, the rest of the sequence is omitted, but the state is maintained, and all depends on the computations afterwards in the catch clause.
2. The *failstop* mbind (written $\text{mbind}_{\text{FailStop}}$). This operator corresponds to a C-like exception handling: System halt and the entire sequence is treated as error. This variant is gained from the above by replacing Some([], σ) in the 5th line of the definition above by None.
3. The *failpurge* mbind. This variant, which we do not detail further in this paper, ignores the failing computations and executes a stuttering step instead. In the modelling of some operating system calls, we found this behaviour useful in situations when atomic actions may fail, report an error, and certain subsequent atomic actions have to be ignored to avoid error-avalanches.

With these mbind operators, valid test sequences for a stepping-function (be it from the specification SPEC or the SUT) evaluating an input sequence $\iota s$ and satisfying a post-condition $P$ can be reformulated to:

$$\sigma \models os \leftarrow \text{mbind } \iota s \text{ SPEC}; \text{return}(P\ \iota s\ os)$$

Second, revisiting the animation Sect. 4.4 and abstracting the pattern of the initial test specification, we can now formally define the concept of a test-conformance notion between an implementation $I$ and a specification $S$:

$$\begin{aligned}(I \sqsubseteq_{\langle Init, CovCrit, conf\rangle} S) \equiv (\forall \sigma_0 \in Init.\ \forall \iota s \in CovCrit.\ \forall res.\\ \sigma_0 \models os \leftarrow \text{mbind } \iota s\ S;\ \text{return}(conf\ \iota s\ os\ res)\\ \longrightarrow \sigma_0 \models os \leftarrow \text{mbind } \iota s\ I;\ \text{return}(conf\ \iota s\ os\ res))\end{aligned}$$

Here, *Init* is a set of initial states, *CovCrit* a super-set constraining the input sequences (this set can be either considered as "test purpose" or as "coverage criterion"), a *coupling variable res* establishing the link between the possible results of the symbolic execution and their use in a test-oracle of the test-execution. We call *conf* a *conformance characterisation* which represents the exact nature of the test-refinement we want to characterise.

**Inclusion Tests and Proven Correct Test-Optimisations.** This means we have a precise characterisation of inclusion conformance introduced in the previous section: We constrain the tests to the test sequences where no exception occurred (as result of a violated enabling condition) in the symbolic execution of the model. It suffices to choose for the conformance characterisation:

$$\text{conf}_{\text{IT}}\ \iota s\ os\ res \equiv (\text{length}(\iota s) = \text{length}(os) \wedge res = os)$$

With this conformance characterization, we can *define* our first explicit test-refinement notion formally by instantiating the test-refinement schema above:

$$(I \sqsubseteq_{\text{IT}\langle Init, CC\rangle} S) \equiv (I \sqsubseteq_{\langle Init, CC, \text{conf}_{\text{IT}}\rangle} S)$$

The setting for $\text{conf}_{\text{IT}}$ (IT for inclusion test) has the consequence that our symbolic executions were only successful iff possible output-sequence are as long as the input sequence. This implies that no exception occurred in possible symbolic runs with possible inputs, i.e., all enabling conditions have to be satisfied.

Now, it can be sformally proven by induction that:

$$\begin{aligned}\sigma \models os \leftarrow \text{mbind}_{\text{FailSave}}\ \iota s\ f; \text{return}(\text{length}(\iota s) = \text{length}(os) \wedge P\ \iota s\ os) =\\ \sigma \models os \leftarrow \text{mbind}_{\text{FailStop}}\ \iota s\ f; \text{return}(P\ \iota s\ os)\end{aligned}$$

This means that in inclusion test-refinements, both $\text{mbind}_{\text{FailSave}}$-occurrences can be replaced by $\text{mbind}_{\text{FailStop}}$. This has a minor and a major advantage:

– At test-execution time, the generated code is slightly more efficient (less cases to check, simpler oracle).

– At symbolic execution time, drastically simpler constraints can be generated: While $\text{mbind}_{\text{FailSave}}$ generates disjunctions for both normal behaviour (enabling condition satisfied) as well as exceptional behaviour (enabling condition violated) were generated, while $\text{mbind}_{\text{FailStop}}$ generates constraints only for normal behaviour, which are therefore simpler to solve in the test-data selection phase by a constraint solver.

A consequence is the following theorem `inclusion_test_I_opt`, which reads presented as natural deduction rule as follows:

$$\frac{\begin{bmatrix}\sigma_0 \in \mathit{Init}, \iota s \in \mathit{CC},\\ \sigma_0 \vDash \mathit{os} \leftarrow \text{mbind}_{\text{FailStop}}\ \iota s\ S; \text{unit}_{\text{SE}}(\mathit{os} = \mathit{res})\end{bmatrix}_{\sigma_0\ \iota s\ \mathit{res}}}{\begin{array}{c}\vdots\\ \sigma_0 \vDash \mathit{os} \leftarrow \text{mbind}_{\text{FailStop}}\ \iota s\ I; \text{unit}_{\text{SE}}(\mathit{os} = \mathit{res})\end{array}}{I \sqsubseteq_{IT\langle \mathit{Init}, \mathit{CC}\rangle} S}$$

**Deadlock-Inclusion.** Using pre-and postcondition predicates, it is straightforward to characterise deadlock conformance: in this kind of test, we investigate that the SUT blocks (in the sense: enabling condition violated) exactly when it should according to the specification. Such test scenarios arise, for example, if a protocol is checked that it *only* does what the specification admits. In other words, we test the absence of back-doors in the implementation of a protocol.

This kind of test is expressed in our framework by the conformance characterisation:

$$\text{conf}_{\text{DF}}\ \mathit{pre}\ \iota s\ \mathit{os}\ \mathit{res} = (\text{length}(\iota s) = \text{length}(\mathit{os}) - 1 \wedge \mathit{res} = \mathit{os} \wedge \neg \mathit{pre}(\text{last}(\iota s))$$

With this conformance characterisation, we can define our second explicit test-refinement notion formally by instantiating the test-refinement schema:

$$(I \sqsubseteq_{\text{DF}\langle \mathit{Init}, \mathit{CC}\rangle} S) \equiv (I \sqsubseteq_{\langle \mathit{Init}, \mathit{CC}, \text{conf}_{\text{DF}}\ \mathit{pre}_S\rangle} S)$$

where $\mathit{pre}_S\ \iota$ is the enabledness condition of $S$ for some input $\iota$. Here, we assume that $\mathit{pre}_S$ only depends on the input and not on the state after the execution of the input sequence. However, this can be easily remedied by a slightly more powerful pattern.[2]

**The Connection to "traditional" IO Conformance.** Another application of our formalisation is the possibility to actually put standard notions based on automata-theoretic notion into relation with our MST Framework. Of natural interest is the IO-Conformance relation mentioned earlier. We pick from a wealth of alternative definitions [23].

However, recall that our framework assumes synchronous communication between tester and SUT; and so far ignores concepts such as quiescence.

[2] The `Monads.thy`-library provides the $\text{assert}_{SE}$-operator for this purpose.

An equivalence between a ioco in the sense of [23] and IOCO in the sense of our MST Framework is therefore only possible for IO-LTS specifications of a particular form. Formalising an IO-LTS in this sense results in:

```
record ('ι, 'o, 'σ) io_lts =
       init  :: "'σ set"
       trans :: "('σ × ('ι + 'o) × 'σ) set"
```

This version of [23] just possesses a disjoint sum of input and output actions; other versions of the same author provide also one or several internal actions $'\tau$; this would result in $('\iota + 'o + '\tau)$.

We skip the straight-forward definitions for "Straces", "out" and "after" (synonym to "States" in Sect. 2) and define:

```
definition out :: "[('ι,'o,'σ) io_lts,'σ set] ⇒ ('o) set"
where      "out TS ss ≡ {a. ∃ s ∈ ss. ∃ s'. (s,Inr a,s') ∈ (trans TS)}"

definition ioco :: "[('ι,'o,'σ)io_lts,('ι,'o,'σ)io_lts] ⇒ bool" (infixl "ioco" 200)
where      "i ioco s ≡ (∀ t ∈ Straces(s). out i (i after t) ⊆ out s (s after t))"
```

On the other hand, we may formalise our own notion of IOCO conformance and relate these two. To this end we specify a conformance characterisation and the resulting third explicit test-refinement notion:

$$\mathrm{conf}_{IOCO}\ post\ \iota s\ os\ res \equiv (res = os \wedge \mathrm{length}(\iota s) = \mathrm{length}(os) \wedge post\ (\mathrm{last}\iota s))$$
$$(I \sqsubseteq_{\mathrm{IOCO}\langle Init, CC\rangle} S) \equiv (I \sqsubseteq_{\langle Init, CC, \mathrm{conf}_{\mathrm{IOCO}}\ post_S\rangle} S)$$

For the following main result of this paper, we introduce an auxiliary notion: we call an io_lts $A$ *strictly IO-alternating* iff all $t \in Straces(A)$ that finish in an input action $\iota$ all prolongations in $t'$ (that is: $t@t' \in Straces(A)$ start with an output action[3]). Moreover, we define a function two_step that serves essentially as wrapper interface to SUT that sends an input action, waits for the returned output-action and binds the latter to the rest of the computation (rather than comparing them to a pre-conceived $o$ and stating "inconclusive" if the observed output does not match to the pre-computed one as in [23].) This enables us to prove the following theorem that links Tretmanns ioco with ours by:

```
theorem ioco_VS_IOCO:
  assumes "strictly_IO_alternating S" and "io_deterministic S"
  shows "∃ S'. I ioco S = ((two_step I) ⊑IOCO⟨{x.True},{x.True}⟩ S'"
```

**Proof Sketch:** We give an existential witness for S' by defining a conversion function convert2SE that converts S into its monadic counterpart. This is done by constructing the Runs of S which must have the form $[...,(\sigma_n, \iota_m, \sigma_{n+1}), (\sigma_{n+1},'o_{m+1},\sigma_{n+2}),...]$. Thus, from the set of Runs, the relation $('\iota \times ('\sigma \rightharpoonup ('o \times '\sigma))$ can be reconstructed, which under the assumptions strictly_IO_alternating and io_deterministic represents a function.

[3] In a definition variant with $'\tau$, these actions must be skipped.

## 6 Conclusion and Future Work

We see several conceptual and practical advantages of a *monadic approach* to sequence testing, the MST Framework:

1. MST's generalise GMM's, io-tagged DA's and NDA's, as well as EFSM's; they are equivalent to particular forms of IO-LTL's and IO-STS's in IOCO conformance settings.
2. MST's can cope with non-deterministic system models (provided they are input-output-deterministic, which we consider a reasonable requirement for system testability).
3. In case of under-specification-non-determinism, substantial case-studies of substantial complexities show the feasibility of our approach [4].
4. the monadic theory models explicitly the difference between input and output, between data under control of the tester and results under control of the SUT,
5. the theory lends itself for a theoretical and practical framework of numerous conformance notions, even non-standard ones, and which gives
6. ways to new calculi for efficient symbolic evaluation enabling symbolic states (via invariants) and input events (via constraints) as well as a seamless, theoretically founded transition from system models to test-drivers.

We see several directions for future work: On the model level, the formal theory of sequence testing should be further explored and extended. It is particularly tempting to incorporate in our MST theory partial-order reduction techniques for further test refinement optimisations.

**Acknowledgement.** This work was partially supported by the Euro-MILS project funded by the European Union's Programme [FP7/2007-2013] under grant agreement number ICT-318353.

## References

1. Andrews, P.B.: Introduction to Mathematical Logic and Type Theory: To Truth through Proof, 2nd edn. Kluwer Academic Publishers, Dordrecht (2002)
2. Brucker, A.D., Brügger, L., Wolff, B.: Formal firewall conformance testing: An application of test and proof techniques. Softw. Testing Verif. Reliab. (STVR) **25**(1), 34–71 (2015)
3. Brucker, A.D., Feliachi, A., Nemouchi, Y., Wolff, B.: Test program generation for a microprocessor. In: Veanes, M., Viganò, L. (eds.) TAP 2013. LNCS, vol. 7942, pp. 76–95. Springer, Heidelberg (2013)
4. Brucker, A.D., Havle, O., Nemouchi, Y., Wolff, B.: Testing the IPC protocol for a real-time operating system. In: Gurfinkel, A., Seshia, S.A. (eds.) VSTTE 2015. LNCS, vol. 9593, pp. 40–60. Springer, Heidelberg (2016). doi:10.1007/978-3-319-29613-5_3
5. Brucker, A.D., Wolff, B.: Test-sequence generation with Hol-TestGen with an application to firewall testing. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 149–168. Springer, Heidelberg (2007)

6. Brucker, A.D., Wolff, B.: On theorem prover-based testing. Formal Aspects Comput. (FAC) **25**(5), 683–721 (2013)
7. Cheng, K.T., Krishnakumar, A.S.: Automatic functional test generation using the extended finite state machine model. In: International Design Automation Conference, DAC 1993, pp. 86–91. ACM, New York (1993)
8. Church, A.: A formulation of the simple theory of types. J. Symbolic Logic **5**(2), 56–68 (1940)
9. D'Antoni, L., Veanes, M.: Minimization of symbolic automata. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, pp. 541–554. ACM (2014)
10. Feliachi, A., Gaudel, M., Wenzel, M., Wolff, B.: The circus testing theory revisited in Isabelle/HOL. In: Formal Methods and Software Engineering, pp. 131–147 (2013)
11. Fraenkel, A., Bar-Hillel, Y.: Foundations of Set Theory. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam (1958)
12. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A symbolic framework for model-based testing. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) FATES 2006 and RV 2006. LNCS, vol. 4262, pp. 40–54. Springer, Heidelberg (2006)
13. Gill, A.: Introduction to the Theory of Finite-State Machines. McGraw-Hill, New York (1962)
14. Halmos, P.: Naive Set Theory. Undergraduate Texts in Mathematics. Springer, New York (1974)
15. Jard, C., Jéron, T.: TGV: theory, principles and algorithms. STTT **7**(4), 297–315 (2005)
16. Jéron, T.: Symbolic model-based test selection. Electr. Notes Theor. Comput. Sci. **240**, 167–184 (2009)
17. Kalaji, A.S., Hierons, R.M., Swift, S.: Generating feasible transition paths for testing from an extended finite state machine (EFSM) with the counter problem. In: Third International Conference on Software Testing, Verification and Validation, ICST, pp. 232–235. IEEE Computer Society (2010)
18. Ponce de León, H., Haar, S., Longuet, D.: Conformance relations for labeled event structures. In: Brucker, A.D., Julliand, J. (eds.) TAP 2012. LNCS, vol. 7305, pp. 83–98. Springer, Heidelberg (2012)
19. Lynch, N., Tuttle, M.: An introduction to input/output automata. CWI-Quarterly **2**(3), 219–246 (1989)
20. Mealy, G.H.: A method for synthesizing sequential circuits. Bell Syst. Tech. J. **34**(5), 1045–1079 (1955)
21. Moore, E.F.: Gedanken-experiments on sequential machines. In: Shannon, C., McCarthy, J. (eds.) Automata Studies, pp. 129–153. Princeton University Press, Princeton (1956)
22. Rusu, V., Marchand, H., Jéron, T.: Automatic verification and conformance testing for validating safety properties of reactive systems. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 189–204. Springer, Heidelberg (2005)
23. Tretmanns, J., Belifante, Z.: Automatic testign with formal methods. In: 7th European International Conference on Software Testing, Analysis and Review (EuroSTAR 1999) (1999)
24. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Soft. Concepts Tools **17**(3), 103–120 (1996)

25. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008)
26. Veanes, M., Bjørner, N.: Alternating simulation and IOCO. STTT **14**(4), 387–405 (2012)
27. Veanes, M., Bjørner, N.: Symbolic automata: the toolkit. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 472–477. Springer, Heidelberg (2012)
28. Wadler, P.: Comprehending monads. Math. Struct. Comput. Sci. **2**(4), 461–493 (1992)

http://www.springer.com/978-3-319-41134-7

Tests and Proofs
10th International Conference, TAP 2016, Held as Part of STAF 2016, Vienna, Austria, July 5-7, 2016, Proceedings
Aichernig, B.K.; Furia, C.A. (Eds.)
2016, XIV, 199 p. 51 illus., Softcover
ISBN: 978-3-319-41134-7