Testing the IPC Protocol for a Real-Time Operating System

Achim D. Brucker[1], Oto Havle[3], Yakoub Nemouchi[2], and Burkhart Wolff[2]

[1] SAP SE, Vincenz-Priessnitz-Str. 1, 76131 Karlsruhe, Germany
achim.brucker@sap.com

[2] LRI, Univ Paris Sud, CNRS, Centrale Suplélec, Université Saclay, France
{nemouchi, wolff}@lri.fr

[3] SYSGO AG, Am Pfaffenstein 14, 55270 Klein-Winternheim, Germany
oto.havle@sysgo.com

Abstract In this paper, we adapt model-based testing techniques to concurrent code, namely for test generations of an (industrial) OS kernel called PikeOS. Since our data-models are complex, the problem is out of reach of conventional model-checking techniques. Our solution is based on symbolic execution implemented inside the interactive theorem proving environment Isabelle/HOL extended by a plugin with test generation facilities called HOL-TestGen.

As a foundation for our symbolic computing techniques, we refine the theory of monads to embed interleaving executions with abort, synchronization, and shared memory to a general but still optimized behavioral test framework.

This framework is instantiated by a model of PikeOS inter-process communication system-calls. Inheriting a micro-architecture going back to the L4 kernel, the system calls of the IPC-API are internally structured by atomic actions; according to a security model, these actions can fail and must produce error-codes. Thus, our tests reveal errors in the enforcement of the security model.

Keywords: test program generation, symbolic test case generations, black box testing, testing operating systems, certification, CC, concurrency, interleaving.

1 Introduction

The verification of systems combining soft- and hardware, such as modern avionics systems, asks for combined efforts in test and proof: In the context of certifications such as EAL5 in Common Criteria [14], the required formal security models have to be linked to system models via refinement proofs, and system models to code-level implementations via testing techniques. Tests are required for methodological reasons ("did we get the system model right? Did we adequately model the system environment?") as well as economical reasons (state of the art deductive verification techniques of machine-level code are *practically* limited to systems with ca. 10 kLOC of size, see [11]).

This paper stands in the context of an EAL5+ certification project [7] of the commercial PikeOS operating system used in avionics applications; PikeOS [18–20] is a virtualizing separation kernel in the tradition of L4-microkernels [10].

. Gurfinkel and S. A. Seshia (Eds.): VSTTE 2015, LNCS, pp. 1–20, 2015.

Our work complements the testing initiative by a model-based testing technique linking the formal system model of the PikeOS inter-process communication against the real system. This is a technical challenge for at least the following reasons:

- the system model is a transaction machine over a very rich state,
- system calls were implemented by internal, uninterruptible "atomic actions" reflecting the L4-microkernel concept; atomic actions define the granularity of our concurrency model, and
- the security model is complex and, in case of aborted system calls, leads to non-standard notions of execution trace interleaving.

To meet these challenges, we need to revise conceptual and theoretical foundations.

- We use symbolic execution techniques to cope with the large state-space; their inherent drawback to be limited to relatively short execution traces is outweighed by their expressive power,
- we extend the "monadic test-sequence approach" proposed in [2, 4] to a test-method for concurrent code. It combines an IO-automata view [13] with extended finite state machines [9] using abstract states and abstract transitions, and
- we need an adaption of concurrency notions, a "semantic view" on partial-order reduction and its integration into interleaving-based coverage criteria.

This sums up to a novel, tool-supported, integrated test methodology for concurrent OS-system code, ranging from an abstract system model in Isabelle/HOL which was not *not authored by us*, complemented by *our* embedding of the latter into our monadic sequence testing framework, *our* setups for symbolic execution down to generation of test-drivers and the code instrumentation.

2 Theoretical and Technical Foundations

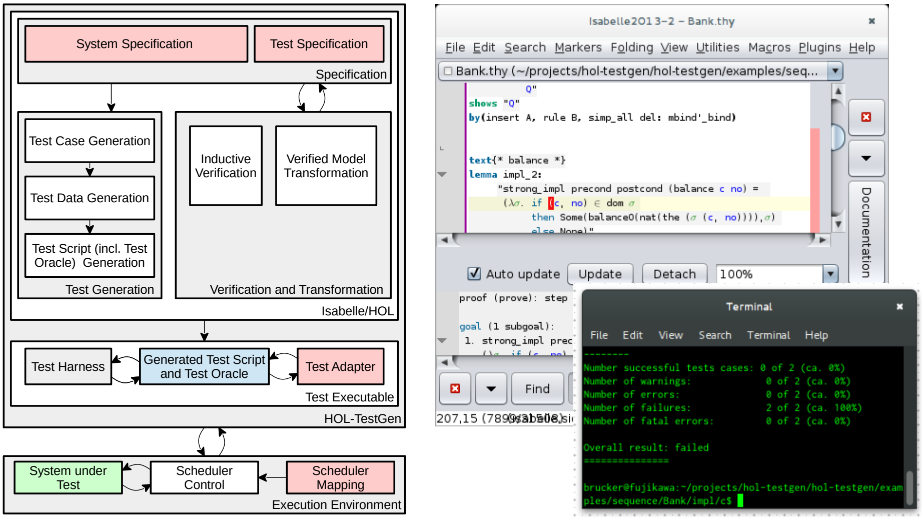
2.1 HOL-TestGen: From Formal Specifications to Testing

HOL-TestGen [3, 4] is a specification-based test case generation environment that integrates seamlessly formal verification and testing in a very unique way:

1. it is an extension of Isabelle/HOL [16] and, thus, inherits all its features (e. g., formal modeling and verification, code generation),
2. its test case generation algorithm is based on the symbolic computation engine of Isabelle and, thus, can count as highly trustworthy,
3. it generates automatically test hypothesis such as the uniformity hypothesis and thus establish a formal link between test and proof (see [4] for details).

Besides test data, HOL-TestGen also generates test drivers including the test oracles for the system under test (SUT) verifying it against the HOL specification. Fig. 1 shows on the left the HOL-TestGen architecture, and on the right a screen shot of its user interface and a test execution. The usual workflow is:

1. we model the SUT using Isabelle/HOL (*system specification*). This modeling process can leverage the full power and methodology of Isabelle, for example,



```
             Q"
shows "Q"
by(insert A, rule B, simp_all del: mbind'_bind)


text{* balance *}
lemma impl_2:
     "strong_impl precond postcond (balance c no) =
      (λσ. if (c, no) ∈ dom σ
           then Some(balance0(nat(the (σ (c, no)))),σ)
           else None)"
```

```
proof (prove): step

goal (1 subgoal):
 1. strong_impl prec
```

```
--------
Number successful tests cases: 0 of 2 (ca. 0%)
Number of warnings:             0 of 2 (ca. 0%)
Number of errors:               0 of 2 (ca. 0%)
Number of failures:             2 of 2 (ca. 100%)
Number of fatal errors:         0 of 2 (ca. 0%)

Overall result: failed
===============


brucker@fujikawa:~/projects/hol-testgen/hol-testgen/examples/sequence/Bank/impl/c$
```

Fig. 1. The HOL-TestGen Workflow.

the system specification can build upon the rich library of datatypes provided by Isabelle or properties of the system specification can be formally proven.

2. we specify the set of test goals (*test specification*), again, in Isabelle/HOL.
3. we use the test case generation implementation of HOL-TestGen to automatically generate abstract *test cases* (that may still contain, e. g., constraints of the form $0 < x < 10$) from the system specification and test specification.
4. we use constraint solvers generating *test data*, i. e., we construct ground instances for the constraints in the test cases (e. g., we choose x to be 4).
5. we generate automatically *test scripts* that execute the SUT as well as validate the test output (by *test oracles*)
6. we compile the test script, together with a generic *test harness*, which controls the test execution and collects statistics about the number of successful or failed tests, to actually execute the test.

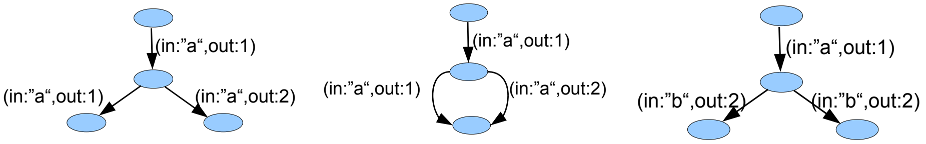
Depending on the SUT, we might need to manually write a small *test adapter* that, e. g., converts data types between the representation in the generated test scripts and the one actually used in the SUT. Moreover, for multi-threaded implementations, a *scheduler mapping* has to be provided that maps abstract threads to the critical infrastructures in the implementation. Usually, the manually written code is orders or magnitude smaller than the generated code of the testers and often reusable between different scenarios.

2.2 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 summarized as follows:

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

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 ι, and in a given state σ, SUT may non-deterministically choose between the successor states σ' and σ'', provided that the pairs (o', σ') and (o'', σ'') are distinguishable. Thus, a SUT may behave non-deterministically, but must make its internal decisions observable by appropriate output. In other words, the relation between a sequence of input-output pairs and the resulting system state must be a function.



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

Fig. 2. IO-Determinism and Non-IO-Determinism

There is a substantial body of theoretical work replacing the latter testability hypothesis by weaker or alternative ones (and avoiding the strict alternates of input and output, and 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. Moreover note, that there are approaches (including our own paper [5]) 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 $\mathit{Trace}(A)$. The function In returns for each trace the set of inputs for which A is enabled after this trace; in Fig. 2c for example, $\mathit{In}\ [(\text{“}a\text{”}, 1)]$ is just $\{\text{“}b\text{”}\}$. Dually, Out yields for a trace t and input $\iota \in \mathit{In}(t)$ the set of outputs for which A is enabled after t; in Fig. 2b for example, $\mathit{Out}([(\text{“}a\text{”}, 1)], \text{“}a\text{”})$ this is just $\{1, 2\}$.

Equipped with these notions, it is possible to formalize 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* [6]: all traces in SPEC must be possible in SUT,

- *deadlock conformance* [8]: for all traces $t \in \mathit{Traces}(\text{SPEC})$ and $b \notin \mathit{In}(t)$, b must be refused by SUT, and
- *input/output conformance (IOCO)* [21]: for all traces $t \in \mathit{Traces}(\text{SPEC})$ and all $\iota \in \mathit{In}(t)$, the observed output of SUT must be in $\mathit{Out}(t, \iota)$.

2.3 Using Monadic Testing Theory

The obvious way to model the state transition relation of an automaton A is by a relation of the type $(\sigma \times (\iota \times o) \times \sigma)\,\text{set}$; isomorphically, one can also model it via:

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

or for a case of a 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: since the transition function can be underspecified via the Hilbert-choice operator, a transition function can be represented by

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

or:

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

for some post-condition post. While in the former "truly non-deterministic" case *step* can and will at run-time choose different results, the latter "underspecified deterministic" version will decide in a given model (so to speak: the implementation) always the same way: a choice that is, however, unknown at specification level and only declaratively described via post. For the system in this paper and our prior work on a processor model [5], it was possible to opt for an underspecified deterministic stepping function.

We abbreviate functions of type $\sigma \Rightarrow (o \times \sigma)\,\text{set}$ or $\sigma \Rightarrow (o \times \sigma)\,\text{option}$ $\text{MON}_{\text{SBE}}(o, \sigma)$ or $\text{MON}_{\text{SE}}(o, \sigma)$, respectively; thus, the aforementioned state transition functions of io-automata can be typed by $\iota \to \text{MON}_{\text{SBE}}(o, \sigma)$ for the general and $\iota \to \text{MON}_{\text{SE}}(o, \sigma)$ 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. Popularized by [22], monads became a kind of standard means to incorporate stateful computations into a purely functional world.

Since we have an underspecified deterministic stepping function in our system model, we will concentrate on the latter monad which is called the *state-exception monad* in the literature.

The operations *bind* (representing sequential composition with value passing) and *unit* (representing the embedding of a value into a computation) are defined for the special-case of the state-exception monad as follows:

```
definition bindSE :: "('o,'σ)MONSE ⇒('o ⇒('o','σ)MONSE) ⇒('o','σ)MONSE"
where      "bindSE f g = (λσ. case f σof None ⇒None
                                | Some (out, σ') ⇒g out σ')"

definition unitSE :: "'o ⇒('o, 'σ)MONSE" ("(return _)" 8)
where      "unitSE e = (λσ. Some(e,σ))"
```

We will write $x \leftarrow m_1;\, m_2$ for the sequential composition of two (monad) computations m_1 and m_2 expressed by $\mathrm{bind}_{\mathrm{SE}}\, m_1(\lambda\, x.m_2)$. 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))
```

On this basis, the concept of a *valid monad execution*, written $\sigma \models m$, can be expressed: an execution of a Boolean (monad) computation m of type $(\mathrm{bool}, \sigma)\ \mathrm{MON}_{\mathrm{SE}}$ is valid iff its execution is performed from the initial state σ, no exception occurs and the result of the computation is true. More formally, $\sigma \models m$ holds iff $(m\ \sigma \neq \mathrm{None} \wedge \mathrm{fst}(\mathrm{the}(m\ \sigma)))$, where fst and snd are the usual *first* and *second* projection into a Cartesian product and the the projection in the Some-variant of the option type.

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:

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

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, characterized by the schema (for all monadic operations m of a system, which can be seen as the its step-functions):

$$\frac{}{(\sigma \models \mathrm{return}\, P) = P} \tag{1a}$$

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

$$\frac{C_m\ \iota\ \sigma \qquad m\ \iota\ \sigma = Some(b, \sigma')}{(\sigma \models s \leftarrow m\ \iota; m'\ s) = (\sigma' \models m'\ b)} \tag{1c}$$

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

An Example: MyKeOS. To present the effect of the symbolic rules during symbolic execution, we present a toy OS-model (our functional PikeOS including our symbolic execution process, theories on interleaving, memory and test scenarios has a length of more than 12 000 lines of Isabelle/HOL code; a complete presentation is therefore out of reach). MyKeOS provides only three atomic actions for *allocation* and *release* of a resource (for example a descriptor of a communication channel or a file-descriptor). A *status* operation returns the number of allocated resources. All operations are assigned to a thread (designated by `thread_id`) belonging to a task (designated by `task_id`, a Unix/POSIX-like *process*); each thread has a thread-local counter in which it stores the number (the status) of the allocated resources. The input is modeled by the data-type:

```
datatype in_c = alloc task_id thread_id nat
              | release task_id thread_id nat
              | status task_id thread_id

datatype out_c = alloc_ok | release_ok | status_ok nat
```

where `out_c` captures the return-values. Since `alloc` and `release` do not have a return value, they signalize just the successful termination of their corresponding system steps. The global table `var_tab` (corresponding to our symbolic state σ) of thread-local variables is modeled as partial map assigning to each active thread (characterized by the pair of task and thread id) the current status:

```
type_synonym thread_local_var_tab = "(task_id ×thread_id) ⇀int"
```

The operation have the precondition that the pair of task and thread id is actually defined and, moreover, that resources can only be released that have been allocated; the initial status of each defined thread is set to 0.

Depicted as an extended finite state-machine (EFSM), the operations of our system model SPEC are specified as shown in Fig. 3. A transcription of an EFSM

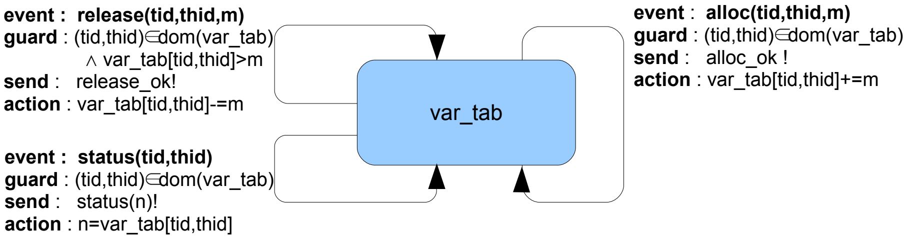

event : release(tid,thid,m)
guard : (tid,thid)∈dom(var_tab)
∧ var_tab[tid,thid]>m
send : release_ok!
action : var_tab[tid,thid]-=m
event : status(tid,thid)
guard : (tid,thid)∈dom(var_tab)
send : status(n)!
action : n=var_tab[tid,thid]
var_tab
event : **alloc(tid,thid,m)**
guard : (tid,thid)∈dom(var_tab)
send : alloc_ok !
action : var_tab[tid,thid]+=m

Fig. 3. SPEC: An Extended Finite State Machine for MyKeOS.

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 Equation 1c:

$$\frac{(\mathit{tid}, \mathit{thid}) \in \mathrm{dom}(\sigma) \qquad \mathrm{SPEC}\ (\mathrm{alloc}\ \ \mathit{tid}\ \mathit{thid}\ m)\ \sigma = \mathit{Some}(\mathrm{alloc\_ok}, \sigma')}{(\sigma \models s \leftarrow \mathrm{SPEC}\ (\mathrm{alloc}\ \ \mathit{tid}\ \mathit{thid}\ m); m'\ s) = (\sigma' \models m'\ \ \mathrm{alloc\_ok})}$$

where $\sigma = \mathit{var\_tab}$ and $\sigma' = \sigma((\mathit{tid}, \mathit{thid}) := (\sigma(\mathit{tid}, \mathit{thid}) + m))$. Thus, this rule allows for computing σ, σ' in terms of the free variables $\mathit{var\_tab}$, tid, thid and m. The rules for release and status are similar. For this rule, SPEC (alloc $\mathit{tid}\ \mathit{thid}\ m$) is the concrete stepping function for the input event alloc $\mathit{tid}\ \mathit{thid}\ m$, and the corresponding constraint C_{SPEC} of this transition is $(\mathit{tid}, \mathit{thid}) \in \mathrm{dom}(\sigma)$.

Conformance Relations Revisited. 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.

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

Successive applications of symbolic execution rules allow to reduce the premise of this implication to $C_{\mathrm{SPEC}}\ \iota_1\ \sigma_1 \longrightarrow \ldots \longrightarrow C_{\mathrm{SPEC}}\ \iota_n\ \sigma_n \longrightarrow \mathit{res} = [a_1 \cdots a_n]$ (where the a_i are concrete terms instantiating the bound output variables o_i), i. e., the constrained equation $\mathit{res} = [a_1 \cdots a_n]$. The latter is substituted into the conclusion of the implication. In our previous example, case-splitting over input-variables ι_1, ι_2 and ι_3 yields (among other instances) $\iota_1 = \mathrm{alloc}\ \ t_1\ \mathit{th}_1\ m$, $\iota_2 = \mathrm{release}\ \ t_2\ \mathit{th}_2\ n$ and $\iota_3 = \mathrm{status}\ \ t_3\ \mathit{th}_3$, which allows us to derive automatically the constraint:

$$\begin{array}{c} (t_1, \mathit{th}_1) \in \mathrm{dom}(\sigma) \longrightarrow (t_2, \mathit{th}_2) \in \mathrm{dom}(\sigma') \wedge n < \sigma'(t_2, \mathit{th}_2) \longrightarrow \\ (t_3, \mathit{th}_3) \in \mathrm{dom}(\sigma'') \longrightarrow \mathit{res} = [\mathrm{alloc\_ok}, \mathrm{release\_ok}, \mathrm{status\_ok}(\sigma''(t_3, \mathit{th}_3)] \end{array}$$

where $\sigma' = \sigma((t_1, \mathit{th}_1) := (\sigma(t_1, \mathit{th}_1) + m)))$ and $\sigma'' = \sigma'((t_2, \mathit{th}_2) := (\sigma(t_2, \mathit{th}_2) - n)))$.

In general, the constraint $C_{\mathrm{SPEC}_i}\ \iota_i\ \sigma_i$ 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 $t_1 = t_2 = t_3 = 0$, $\mathit{th}_1 = \mathit{th}_2 = \mathit{th}_3 = 5$, $m = 4$ and $n = 2$ satisfies the constraint and would produce (predict) the output sequence $\mathit{res} = [\mathrm{alloc\_ok}, \mathrm{release\_ok}, \mathrm{status\_ok}\,2]$ for SUT according to SUT. Thus, a resulting (abstract) test-driver is:

$$\begin{array}{l} \sigma \models o_1 \leftarrow \mathrm{SUT}\ \ \iota_1; \ldots; o_3 \leftarrow \mathrm{SUT}\ \ \iota_3; \\ \quad \mathrm{return}([\mathrm{alloc\_ok}, \mathrm{release\_ok}, \mathrm{status\_ok}\,2] = [o_1 \cdots o_3]) \end{array}$$

This schema of a test-driver synthesis can be refined and optimized. First, for iterations of stepping functions an ’mbind’ operator can be defined, which is basically a fold over bind_{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. Using mbind, valid test sequences for a stepping-function (be it from the specification SPEC or the SUT) evaluating an input sequence ιs and satisfying a post-condition P can be reformulated to:

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

Second, we can now formally define the concept of a test-conformance notion:

$$\begin{aligned}
&(\text{SPEC} \sqsubseteq_{\langle \text{Init},\text{CovCrit},\text{conf}\rangle} \text{SUT}) = \\
&\qquad\qquad (\forall \sigma_0 \in \mathit{Init}.\ \forall \iota\ s \in \mathit{CovCrit}.\ \forall \mathit{res}. \\
&\qquad\qquad\qquad \sigma_0 \models os \leftarrow \text{mbind}\ \ \iota s\ \ \text{SPEC};\ \ \text{return}(\mathit{conf}\ \ \iota s\ \ os\ \ \mathit{res}) \\
&\qquad\qquad\quad \longrightarrow \\
&\qquad\qquad\quad \sigma_0 \models (os \leftarrow \text{mbind}\, \iota s\ \ \mathit{SUT};\ \ \text{return}(\mathit{conf}\ \ \iota s\ \ os\ \ \mathit{res})))
\end{aligned}$$

For example, if we instantiate the conformance predicate conf by:

$$\text{conf}\ \ \iota s\ \ os\ \ \mathit{res} = (\text{length}(\iota s) = \text{length}(os) \wedge \mathit{res} = os)$$

we have a precise characterization of inclusion conformance introduced in the previous section: We constrain the tests to those test sequences where no exception occurs in the symbolic execution of the model. Symbolic execution fixes possible output-sequence (which must be as long as the input sequence since no exception occurs) in possible symbolic runs with possible inputs, which must be exactly observed in the run of the SUT in the resulting abstract test-driver.

Using pre-and postcondition predicates, it is straight-forward to characterize deadlock conformance or IOCO mentioned earlier (recall that our framework assumes synchronous communication between tester and SUT; so this holds only for a IOCO-version without quiescence). Further, we can characterize a set of initial states or express constraints on the set of input-sequences by the *coverage criteria* $\mathit{CovCrit}$, which we will discuss in the sequel.

2.4 Coverage Criteria for Interleaving

In the following, we consider input sequences ιs which were built as interleaving of one or more inputs for different processes; for the sake of simplicity, we will assume that it is always possible to extract from an input event the thread and task id it belongs to. It is possible to represent this interleaving, for example, by the following definition:

```
fun interleave :: "’a list ⇒’a list ⇒’a list set"
where "interleave [] [] = {[]}"
     |"interleave A [] = {A}"
```

```
|"interleave [] B = {B}"
|"interleave (a # A) (b # B) =
   (λx. a # x) ‘interleave A (b # B) ∪
   (λx. b # x) ‘interleave (a # A) B"
```

and by requiring for the input sequence ιs to belong to the set of interleavings of two processes `P1` and `P2`: ι`s` $\in$`interleave P1 P2`.

It is well known that the combinatorial explosion of the interleaving space represents fundamental problem of concurrent program verification. Testing, understood as the art of creating finite, well-chosen subspaces for large input-output spaces, offers solutions based on adapted coverage criteria [17] of these spaces, which refers to particular instances of `CovCrit` in the previous section. A well-defined coverage criterion [1, 23] can reduce a large set of interleavings to a smaller and manageable one. For example, consider the executions of the two threads in MyKeOS: `T = [alloc 3 1 2, release 3 1 1, status 3 1]` and `T' = [alloc 2 5 3, release 3 1 1, status 2 5]`. Since our simplistic MyKeOS has no shared memory, we simulate the effect by allowing `T'` to execute a `release`-action on the local memory of task 3, thread 1 by using its identity. In general, we are interested in all possible values of a shared program variable x at position l after the execution of a process P. To this end we will define two sets of interleavings under two different known criteria.

- **Criterion1: standard interleaving** (SIN) *the interleaving space of actions sequences gets a complete coverage iff all feasible interleavings of the actions of P are covered.*
- **Criterion2: state variable interleaving** (SVI) *the interleaving space of actions sequences gets a complete coverage iff all possible states of x at l in P are covered.*

The number of interleavings increases exponentially with the length of traces (for bounds of the combinatorial explosion, see [17]). Under SIN we derive 10 possible actions sequences, which is reduced under SVI to 3 sequences (where one leads to a crash; recall our assumption that the memory is initially 0). Unlike to SIN, SVI has provided a smaller interleaving set that cover all possible states. If we consider `var_tab[3,1]` for x when executing `status 3 1`, the possible results may be undefined, O or 1. While SIN has provided a bigger set, that cover all possible 3 states of x with redundant sequences representing the same value. In model-checking, this reduction technique is also known as partial order reduction. It is now part of the beauty of our combined test and proof approach, that we can actually formally prove that the test-sets resulting from the test-refinements:

$$\text{SPEC} \sqsubseteq_{\langle Init,SIN,conf\rangle} \text{SUT} \quad \text{and} \quad \text{SPEC} \sqsubseteq_{\langle Init,SVN,conf\rangle} \text{SUT}$$

are equivalent for a given SPEC. The core of such an equivalence proof is, of course, a proof of commutativity of certain step executions, so properties of the form:

$$o \leftarrow \text{SPEC}\ \ \iota_i; o' \leftarrow \text{SPEC}\ \ \iota_j; M\ o\ o' = o' \leftarrow \text{SPEC}\ \ \iota_j; o \leftarrow \text{SPEC}\ \ \iota_i; M\ o\ o',$$

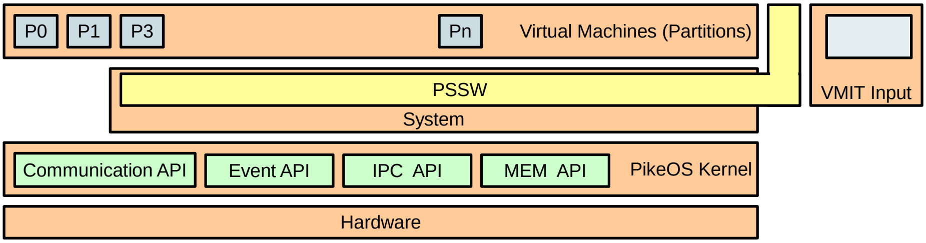


Fig. 4. PikeOS architecture.

which are typically resulting from the fact that these executions depend on disjoint parts of the state. In MyKeOS, for example, such a property can be proven automatically for all $\iota_i = \text{release}\ t\ th$ and $\iota_j = \text{release}\ t'\ th'$ with $t \neq t' \vee th \neq th'$; such reordering theorems justify a partial order on inputs to reduce the test-space. We are implicitly applying the testability hypothesis that SUT is input-output deterministic; if a input-output sequence is possible in SPEC, the assumed input-output determinism gives us that repeating the test by an equivalent one will produce the same result.

3 Application: Testing PikeOS

In the following, we will outline the PikeOS model (the full-blown model developed as part of the EUROMILS project is about 20 kLOC of Isabelle/HOL code), and demonstrate how the this model is embedded into our monadic testing theory.

3.1 PikeOS System Architecture

PikeOS is an operating system that supervises and ensures the execution and separation between software applications running on the top of various hardware platforms [19]. It stands in the tradition of so-called *separation kernels* and follows ideas of the influential L4 kernel project [12]. The PikeOS architecture comprises four layers (see Fig. 4). The *virtual machine initialization table* (VMIT) is a data-base containing the global configuration of the system and its application structure. In the VMIT, *partitions* (virtual machines), *tasks* (POSIX-like processes), their *threads*, their memory-, processor-, and time resources, communication channels as well as access-control rights on these resources were defined. Only at boot-time, partitions, processes and threads can be created via *PikeOS System Software* (PSSW); at run-time the application structure and its time-scheduling is fixed: PikeOS has no dynamic process creation. In other words: based on the VMIT configuration, the *PikeOS system software* (PSSW) will generate a set of virtual machines in the *Partitions layer* during the boot-phase. In this layer each resource partition is composed from a set of applications, and can be executed under the predefined policy and use the predefined resources of the VMIT. Applications in the resource partitions can also be used for system calls of *PikeOS kernel*. In kernel layer, the set of resource partitions is seen as a set of PikeOS *tasks*, that contain PikeOS *threads* and shares kernel resources (memory, files, processors, communication channels . . .).

The kernel provides a set of APIs used by the threads and tasks. As in Unix-like systems, special hardware—the MMU—gives application-level *tasks* the illusion to live in an own separate memory space: the virtual memory. However, all *threads* belonging to a task live in the same memory space, namely the memory space of the task they belong to. In contrast, system-level tasks can also access the physical memory and the MMU. Besides memory separation, PikeOS also offers time-separation and multi-core support.

Our work focuses on a particular part of the kernel layer providing interprocess communication (IPC), the PikeOS IPC API.

3.2 PikeOS IPC API

The IPC mechanism [19, 20] is the primary means of thread communication in PikeOS. Historically, its efficient implementation in L4 played a major role in the micro-kernel renaissance after the early 1990s. Microkernels had received a bad reputation, as systems built on top were performing poorly, culminating in the billion-dollar failure of the IBM Workplace OS. A combination of shared memory techniques—the MMU is configured such that parts of virtual memory space are actually represented by identical parts of the physical memory—and a radical redesign of the IPC primitives in L4 resulted in an order-of-magnitude decrease in IPC cost. Also in PikeOS, IPC message transfer can operate between threads which may belong to different tasks. However, the kernel controls the scope of IPC by determining, in each instance, whether the two threads are permitted to communicate with each other. IPC transfer is based on shared memory, which requires an agreement between the sender and receiver of an IPC message. If either the sending or the receiving thread is not ready for message transfer, then the other partner must wait. Both threads can specify a timeout for the maximum time they are prepared to wait and have appropriate access-control rights. Our IPC model includes eight *atomic actions*, corresponding more-or-less to code sections in the API system calls `p4_ipc_buf_send()` and `p4_ipc_buf_recv()` protected by a global system lock. If errors in these actions occur—for example for lacking access-rights—the system call is *aborted*, which means that all atomic actions belonging to the running system call as well as the call of the communication partner were skipped and execution after the system calls on both sides is continuing as normal. It is the responsibility of the application to act appropriately on error-codes reported as a result of a call.

3.3 PikeOS Model Organization

We model the protocol as composition of several operational semantics; this composition is represented by monad-transformers adding, for example, to the basic transition semantics the semantics for abort behavior. The execution of

IPC system calls is supervised by a protocol containing a number of stages corresponding to atomic actions.

3.4 Embedding the PikeOS Functional Model into the Monadic Framework

System State. In our model, the system state is an abstraction of the VMIT (which is immutable) and mutable task specific resources. It is presented by the (polymorphic) record type:

```
record ('memory,'thread_id,'thread,'sp_th_th,'sp_th_res,'errors)kstate=
  resource               :: 'memory
  current_thread         :: 'thread_id
  thread_list            :: "'thread list"
  communication_rights   :: 'sp_th_th
  access_rights          :: 'sp_th_res
  error_codes            :: 'errors
  errors_tab             ::  'thread_id ⇀ 'errors
```

Note that the syntax is very close to functional programming languages such as SML or OCaml or F#. The parameterization is motivated by the need of having different abstraction layers throughout the entire theory; thus, for example, the *resource* field will be instantiated at different places by abstract shared memory, physical memory, physical memory and devices, etc.—from the viewpoint of an operating system, devices are just another implementation of memory. In the entire theory, these different instantiations of `kstate` were linked by abstraction relations establishing formal refinements. Similarly, the field *current_ thread* will be instantiated by the model of the *ID* of the thread in the execution context and more refined versions thereof. *thread_ list* represents information on threads and there executions. The *communication _ rights* field represent the communication policy defined between the active entities (i. e., threads and tasks). The field *access_ rights* represent the access policy defined between active entities and passive entities (i. e., system resources).

For the purpose of test-case generation, we favor instances of `kstate` which are as abstract as possible and for which we derived suitable rules for fast symbolic execution.

Shared Memory Model. Shared memory is the key for the L4-like IPC implementations: while the MMU is usually configured to provide a separation of memory spaces for different tasks (a separation that does not exist on the level of physical memory with its physical memory pages, page tables, . . .), there is an important exception: physical pages may be attributed to two different tasks allowing to transfer memory content directly from one task to another.

We will use an abstract model for memory with a sharing relation between addresses. The sharing relation is used to model the IPC map operation, which establishes that memory spaces of different tasks were actually shared, such that writes in one memory space were directly accessed in the other. Under the sharing relation, our memory operations respect two properties:

1. Read memory on shared addresses returns the same value.
2. All shared addresses has the same value after writing.

We will present just the key properties of our shared memory model, where *write* is denoted by `_ :=$ _` and *read* by `_ $ _`:

```
typedef (α, β) memory = "..."

x shares(σ) x      x shares(σ) y ⟹y shares(σ) x  ...

x shares(σ) y ⟹ y ∈Domain σ ⟹ σ (x :=$ (σ $ y)) = σ
x ∈Domain σ    ⟹  σ $ x = z ⟹ σ(x:=$ z) = σ
z shares(σ) x ⟹ σ(x :=$ a) $ z = a
¬(z shares(σ) x) ⟹σ(x :=$ a) $ z = σ$ z
x shares(σ) x' ⟹ σ(x :=$ y)(x' :=$ z) = (σ(x' :=$ z))
```

or, in other words, a memory theory where addresses were considered modulo sharing.

Atomic Actions. As mentioned earlier, the execution of the system call can be interrupted or *aborted* at the border-line of code-segments protected by a lock. To avoid the complex representation of interruption points, we model the effect of these lock-protected code-segments as atomic actions. Thus, we will split any system call into a sequence of atomic actions (the problem of addressing these code-segments and influencing their execution order in a test is addressed in the next section). Atomic actions are specified by datatype as follows:

```
datatype ('ipc_stage,'ipc_dir)action_ipc = IPC 'ipc_stage 'ipc_dir
datatype p4_stage_ipc = PREP | WAIT | BUF | MAP | DONE

datatype ('thread_id ,'adresses) p4_direct_ipc =
                SEND "'thread_id" "'thread_id" "'adresses"
              | RECV "'thread_id" "'thread_id" "'adresses"

type_synonym
    ACTION_ipc = (p4_stage_ipc,(nat×nat×nat,nat list)p4_direct_ipc)action_ipc
```

Where `ACTION`$_{\texttt{ipc}}$ is type abbreviation for IPC actions instantiated by `p4_direct`$_{\texttt{ipc}}$. The type `ACTION`$_{\texttt{ipc}}$ models exactly the input events of our monadic testing framework. Thread IDs are triples of natural numbers that specify the resource partition the thread belongs to as well as the task and the individual id. The stepping function as a whole is too complex to be presented here; we refrain on the presentation of a portion of an auxilliary function of it that models just the `PREP_SEND` stage of the IPC protocol; it must check if the task and thread id of the communication partner is allowed in the VMIT, if the memory is shared to this partner, if the sending thread has in fact writing permission to the shared memory, etc. The VMIT is part of the resource, so the memory configuration, and auxiliary functions like `is_part_mem_th` allow for extracting the relevant information from it. The semantic of the different stages is described using a total functions:

```
definition PREP_SEND ::"ACTIONipc stateid⇒ ACTIONipc ⇒ACTIONipc stateid"
where "PREP_SEND σ act =
      (case act of (IPC PREP (SEND caller partner msg)) ⇒
        ...
          if is_part_mem_th (get_thread_by_id'' partner σ) (resource σ)
          then
            if IPC_params_c1 (get_thread_by_id'' partner σ)
            then ...)
```

Where PREP_SEND, WAIT_SEND, BUF_SEND, and DO NE_SEND define an operational semantic for the stages of the PikeOS IPC protocol.

Traces, Executions and Input Sequences. During our experiments, we will generate *input sequences* rather than traces. An input sequence is a list of a datatype capturing atomic action input syntactically. An *execution* is the application of a transition function over a given input sequence. Using mbind, the execution over a given input sequence *is* can be immediately constructed.

```
 definition execution = (λis ioprog σ. mbind is ioprog σ)
```

IPC Execution Function. The execution semantic of the IPC protocol is expressed using a total function:

```
fun exec_action ::"ACTIONipc stateid⇒ ACTIONipc ⇒ACTIONipc stateid"
where
  PREP_SEND_run:"exec_action σ(IPC PREP (SEND caller partner msg)) =
                  PREP_SEND σ (IPC PREP (SEND caller partner msg))"|
  (...)
```

The function is adapted to the monads using the following definition:

```
definition exec_action_Mon
where    "exec_action_Mon = (λact σ. Some (error_codes(exec_action σact),
                                             exec_action σ act))"
```

System calls. As mentioned earlier, PikeOS system calls are seen as sequence of atomic actions that respect a given ordering. Actually, each system call can perform a set of *operations*. PikeOS IPC API provides seven different calls, the most general one is the call *P4_ ipc()*. Using *P4_ ipc()*, five operations can be performed:

1. Send a copied message,
2. Receive a copied message,
3. Receive an event (not modeled),
4. Send a mapped message (not used in this paper), and
5. Receive a mapped message (not used in this paper).

The corresponding Isabelle model for the call is:

```
datatype ('thread_id, 'msg) P4_IPC_call =
   P4_IPC_call    'thread_id 'thread_id 'msg
|  P4_IPC_BUF_call 'thread_id 'thread_id 'msg
|  P4_IPC_MAP_call 'thread_id 'thread_id 'msg
   (...)
```

Communication coverage criterion. An IPC call defines a *communication* relation between two threads. In PikeOS, IPC communications can be symmetric, transitive but can not be reflexive (a thread can not send or receive an IPC message for himself). The transitivity or intransitivity of IPC communications depends mainly on the defined communication rights table and access rights table. In this section, we will define a set of Isabelle rules to derive input sequences for ipc calls. The derived input sequences express IPC communications between threads. Other rules, which are almost the same as the ones used for deriving input sequences, will be defined to derive the possible communications between threads after the execution of an IPC call. While IPC input sequences will be used in scenarios for testing information flow policy via IPC error codes, IPC communications let us to address scenarios on access control policy implemented via the two tables cited before.

To this end we define a new coverage criterion, i. e., the set of interleavings that satisfy all these constrains. The definition of the criterion is based on the functional model of PikeOS IPC (see Sec. 3.2) and our technique to reduce the set of interleaving if two actions can commute (see Sec. 2.4).

- **Criterion3: IPC communications** (IPC_{comm}) *the interleaving space of input sequences gets a complete coverage iff all IPC communications of a given SUT are covered.*

IPC communications are input sequences. An example of a communication derived under $\mathbf{IPC}_{comm}$ is:

```
[IPC PREP (SEND th_id th_id' msg), IPC PREP (RECV th_id' th_id msg),
 IPC WAIT (SEND th_id th_id' msg), IPC WAIT (RECV th_id' th_id msg),
 IPC BUF (RECV th_id' th_id msg), IPC DONE (RECV th_id' th_id msg),
 IPC DONE (SEND th_id th_id' msg)]"
```

4 Test Generation

Test scenarios. A test scenario is represented by a test specification and can have two main schemes: unit test or sequence test. An example of a test scenario is the the specification `TS_simple_example2`:

```
test_spec TS_simple_example2:
  is ∈IPC_communication ⟹
  σ₁ ⊨(outs ←mbind is(abort_lift exec_action_Mon);return(outs = x)
⟶σ₁ ⊨(outs ←mbind is SUT; return(outs = x))
```

For a σ_1 definition that contains a suitable VMIT configuration, a possible `is` is, e. g.:

```
[IPC PREP (RECV (0,0,1) (0,0,2) [0,4,5,8]),
 IPC PREP (SEND (0,0,2) (0,0,1) [0,4,5,8]),
 IPC WAIT (RECV (0,0,1) (0,0,2) [0,4,5,8]),
 IPC WAIT (SEND (0,0,2) (0,0,1) [0,4,5,8]),
 IPC BUF  (SEND (0,0,2) (0,0,1) [0,4,5,8]),
 IPC DONE (SEND (0,0,2) (0,0,1) [0,4,5,8]),
 IPC DONE (RECV (0,0,1) (0,0,2) [0,4,5,8])]
```

The sequence is an abstraction of an IPC communication between the thread with the $ID = (0, 0, 1)$ and the thread with $ID = (0, 0, 2)$ via a message $msg = [0, 4, 5, 8]$. Natural numbers inside the message are abstractions on memory addresses. The execution semantic of the input sequence is represented by our execution function *exec_ action_ Mon*. We wrap around our execution function a monad transformer $abort_{lift}$ that express the behavior of an abort. The equality specify our conformance relation between SUT outputs and the model outputs. After using our symbolic execution process the `out` of this test case is:

```
[NO_ERRORS,
 NO_ERRORS,
 ERROR_IPC error_IPC_1_in_WAIT_RECV,
 ERROR_IPC error_IPC_1_in_WAIT_RECV,
 ERROR_IPC error_IPC_1_in_WAIT_RECV,
 ERROR_IPC error_IPC_1_in_WAIT_RECV,
 ERROR_IPC error_IPC_1_in_WAIT_RECV]
```

The error-codes observed in the sequence is related to IPC. The error-codes was returned in the stage *WAIT_ RECV*. The interpretation of this error-codes is that the thread has not the rights to communicate with his partner. We can observe the behavior of our abort operator in this sequence of error-codes; All stages following `WAIT_RECV` are purged (not executed), and the same error is returned instead. We focus on error-codes in our scenarios, since error-codes represent a potential for undesired information flow: for example, un-masked error-messages may reveal the structure of tasks and threads of a foreign partition in the system; a revelation that the operating system as separation kernel should prevent.

Generating Test Drivers. In this section we address the problem to compile "abstract test-drivers" as described in the previous sections into concrete code and code instrumentations that actually execute these tests.

HOL-TestGen can generate test scripts (recall Fig. 1) in SML, Haskell, Scala and F#. For our application, we generate SML test scripts and use MLton (`www.mlton.org`) for building the test executable: MLton 1. provides a foreign function interface to C and 2. is easily portable to small POSIX system.

In more detail, we generate two SML structures *automatically* from the Isabelle theories. The first structure, called `Datatypes`, contains the datatypes that are used by the interface of the SUT. In our example, this includes, e. g., `IPC_protocol` and `P4_IPC_call`. The second structure, called `TestScript`, contains a list of all generated test cases as well the *test oracle*, i. e., the algorithms necessary to decide if a test result complies to the specification or not.

In addition, for testing C code, we need to provide a small SML structure (ca. 20 lines of code), called `Adapter`, that serves two purposes: 1. the configuration of the foreign function, e. g., the mapping from SML datatypes to C datatypes and 2. the concretization of abstractions to bridge the gap between an abstract test model and the concrete SUT. The `Adapter` structure only needs to be updated after significant changes to either the system specification or the system under test.

For testing concurrent, i. e., multi-threaded, programs we need to solve a particular challenge: *enforcing certain thread execution orders* (a certain scheduling) during test execution. There are, in principle, three different options available to control the scheduler during test execution: 1. instrumenting the SUT to make the thread switching deterministic and controllable, 2. using a deterministic scheduler that can be controlled by test driver, or 3. using the features of debuggers, such as the GNU debugger (gdb), for multi-threaded programs.

In our prototype for POSIX compliant systems, we have chosen the third option: we execute the SUT within a gdb session and we use the gdb to switch between the different threads in a controlled way. We rely on two features of gdb (thus, out approach can be applied to any other debugger with similar features), namely: 1. the possibility to attach to break points in the object code scripting code that is executed if a break point is reached and 2. the complete control of the threading, i. e., gdb allows to switch explicitly between threads while ensuring that only the currently active thread is executed (using the option `set scheduler-locking on`).

This approach has the advantage that we neither need to modify the SUT nor do we need to develop a custom scheduler. We only need to generate a configuration for controlling the debugger. The necessary gdb command file is generated automatically by HOL-Testgen based on a mapping of the abstract thread switching points to break points in the object code. The break points at the entry points allows us to control the thread creation, while the remaining break points allow us to control the switching between threads. Thus, we only need the SUT compiled in debugging mode and this mapping. In this sense, we still have a "black-box" testing approach.

Moreover, Using gdb together with `taskset`, we ensure that all threads are executed on the same core; in our application, we can accept that the actual execution in gdb changes the timing behavior. Moreover, we assume a sequential memory model, so our approach does not cover TLB-related race conditions occurring in multi-core CPU's.

5 Conclusion

Related Work. There is a wealth of approaches for tests of behavioral models; they differ in the underlying modeling technique, the testability and test hypothesis', the test conformance relation etc.; in Sec. 2 we mention a few. Unfortunately, many works make the underlying testability hypothesis' not explicit which makes a direct comparison difficult and somewhat vague. For the space of testability assumptions used here (the system is input-output deterministic, is adequately modeled as underspecified deterministic system, synchronous coupling between tester and SUT suffices), to the best of our knowledge, our approach is unique in its integrated process from theory, modeling, symbolic execution down to test-driver generation.

With respect to the test-driver approach, this work undeniably owes a lot Microsoft's CHESS project [15], which promoted the idea to actually control the scheduler of real systems and use partial-order reduction techniques to test systematically concurrent executions for races in applications of realistic size (e. g., IE, Firefox, Apache). For our approach, controlling the scheduler is the key to justify the presentation of the system as underspecified-deterministic transition function.

Conclusion and Future Work. We see several conceptual and practical advantages of a *monadic approach* to sequence testing:

1. a monadic approach resists the tendency to surrender to finitism and constructivism at the first-best opportunity; a tendency that is understandably wide-spread in model-checking communities,
2. it provides a sensible shift from syntax to semantics: instead of a first-order, intentional view in *nodes* and *events* in automata, the heart of the calculus is on *computations* and their *compositions*,
3. 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,
4. the theory lends itself for a theoretical and practical framework of numerous conformance notions, even non-standard ones, and which gives
5. ways to new calculi of 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 (as given in the HOL-TestGen library theories `Monad.thy` and `TestRefinements.thy`) providing connections between monads, rules for test-driver optimization, different test refinements, etc., is worth further development. On a test-theoretical level, our approach provides the basis for a comparison on test-methods, in particular ones based on different testability hypothesis'.

Pragmatically, our test driver setup needs to be modified to be executable on the PikeOS system level. For this end, we will need to develop a host-target setup (see Sec. 4). Finally, we are interested in extending our techniques to actually test information flow properties; since error-codes in applications may reveal internal information of partitions (as, for example, the number of its tasks and threads), this seems to be a rewarding target. For this purpose, not only action sequences need to be generated during the constraint solving process, but also (abstract) VMITs.

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] P. Ammann, J. Offutt, and W. Xu. *Formal methods and testing.* Springer, 2008.

[2] A. D. Brucker and B. Wolff. Test-sequence generation with HOL-TestGen with an application to firewall testing. In *Tests and Proofs*, LNCS 4454, p. 149–168, Springer, 2007.

[3] A. D. Brucker and B. Wolff. HOL-TestGen: An interactive test-case generation framework. In FASE , LNCS 5503, p. 417–420. Springer, 2009.

[4] A. D. Brucker and B. Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 2012.

[5] A. D. Brucker, A. Feliachi, Y. Nemouchi, and B. Wolff. Test program generation for a microprocessor. In *TAP*, LNCS 7942, p. 76–95, 2013.

[6] H. P. de Leòn, S. Haar, and D. Longuet. Conformance relations for labeled event structures. In *TAP*, LNCS 7305, p. 83–98, Springer 2012.

[7] Euro-Mils. http://www.euromils.eu/

[8] A. Feliachi, M. Gaudel, M. Wenzel, and B. Wolff. The circus testing theory revisited in Isabelle/HOL. In *ICFEM*, LNCS 8144, p. 131–147, Springer 2013.

[9] A. Gill. *Introduction to the theory of finite-state machines.* McGraw-Hill, 1962.

[10] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of microkernel-based systems. In *SOSP*, 1997.

[11] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an OS kernel. In *SOSP*, p. 207–220. 2009.

[12] Liedtke. on μ-kernel construction. *SOSP*, 29(5):237–250, 1995.

[13] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[14] Common criteria for information technology security evaluation. http://www.commoncriteriaportal.org/.

[15] M. Musuvathi, S. Qadeer, and T. Ball. Chess: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, Microsoft Research, 2007.

[16] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*, LNCS 2283. Springer, 2002.

[17] W. J. Shan Lu and Y. Zhou. A study of interleaving coverage criteria. *ESEC-FSE companion*, p. 533–536, 2007.

[18] SYSGO. Pikeos. http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/.

[19] SYSGO. *PikeOS Fundamentals.* SYSGO, 2013.

[20] SYSGO. *PikeOS Kernel.* SYSGO, 2013.

[21] J. Tretmans. Model based testing with labelled transition systems. LNCS 4949, p. 1–38, 2008.

[22] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

[23] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.