POWVER

Technical Report 2017-07

Title: **Model-Based Testing for Asynchronous Systems**

Authors: Alexander Graf-Brill, Holger Hermanns

Report Number: 2017-07

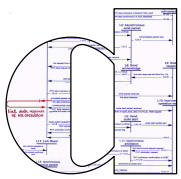
ERC Project: Power to the People. Verified.

ERC Project ID: 695614

Funded Under: H2020-EU.1.1. – EXCELLENT SCIENCE

Host Institution: Universität des Saarlandes, Dependable Systems and Software
Saarland Informatics Campus

Published In: FMICS-AVoCS 2017



POWER TO THE PEOPLE.
VERIFIED.
erc

Model-Based Testing for Asynchronous Systems

Alexander Graf-Brill and Holger Hermanns

Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Abstract Model-based testing is a prominent validation technique, integrating well with other formal approaches to verification, such as model checking. Automated test derivation and execution approaches often struggle with asynchrony in communication between the implementation under test (IUT) and tester, a phenomenon present in most networked systems. Earlier attacks on this problem came with different restrictions on the specification model side. This paper presents a new and effective approach to model-based testing under asynchrony. By waiving the need to guess the possible output state of the IUT, we reduce the computational effort of the test generation algorithm while preserving soundness and conceptual completeness of the testing procedures. In addition, no restrictions on the specification model need to be imposed. We define a suitable conformance relation and we report on empirical results obtained from an industrial case study from the domain of electric mobility.

1 Introduction

Model-based testing is a validation technique where, based on a formal specification of a system, a suitable set of experiments (test suite) is generated in an automated manner and executed on the implementation of that system, so as to assert some notion of conformance between the implementation and its specification. In model-based testing it is common to use variants of input-output transitions systems (IOTS) as formal models to capture the system behaviour on the specification side. In IOTS, transitions between states have structured action labels: the name of a performed action and an identifier of its type, i.e. input (stimuli) to the implementation or output (response) of the implementation. By automated inspection of the possible inputs and outputs in the current states of a given specification model, a model-based testing tool can either provide one of these inputs to or records an output from the implementation under test (IUT). It then updates its knowledge of the current state in the specification model. Whenever an unexpected output of the IUT occurs, i.e. an output which is not considered possible according to the current state(s) of the specification model, the IUT is refused with a verdict "fail". Testing is usually employed for finding problems in an IUT, instead of for verifying the absence of any problems. Nevertheless it is theoretically appealing to discuss the size a complete test suite needs to have in order to be usable for such a verification. Finiteness of such a complete test suite however requires finite and acyclic behaviour, which is rarely the case for embedded systems, the class of systems we look at.

While the specification can be provided as a formal model, this is not naturally given for the IUT, which is most often a real physical object, or a piece of code. To enable a formal relation between the specification and the IUT, the so-called *testing hypothesis* or *test assumption*, is usually put in place, assuming the existence of an equivalent formal model of the IUT. It is common to use IOTS for both, the model of the specification and the IUT, as we do in the sequel.

The most prominent conformance relation in use is *input-output conformance* (**ioco**) [24]. It is defined for systems interacting *synchronously* with their environment, and especially with the model-based testing tool. Here "synchronously" means that each input to the IUT instantaneously leads to a state transition in the IUT, and each output of the IUT can be instantaneously processed by its environment. Model-based testing for synchronous communication has been extensively studied for decades [22,23,2,10,19,17,9,8,21,18], spanning varying conformance relations and modelling formalisms. IOTS may be nondeterministic in the sense that a state has several outgoing transitions with the same label, so as to support abstraction or implementation freedom wrt. certain system aspects.

In contrast to synchronous testing, where the exact state of an IUT on the specification side is known modulo non-determinism, this does not hold if testing systems communicating asynchronously, especially if being tested via one or more asynchronous channels. Rooted in possible message delays, it is then no longer guaranteed that inputs provided to and outputs received from an IUT are being processed in the order they appear to the tester.

Asynchronous communication can appear in different flavours, since buffering and delaying of messages may happen in various ways, depending on the characteristics of the channels connecting the two sides. Channels may only delay inputs wrt. outputs, or the other way around, they may allow arbitrary re-ordering of messages, for instance if separate channels for different inputs or outputs are in place. However, the most commonly assumed communication scenario is that of bidirectional FIFO (first-in-first-out) communication, effectuated by two independent FIFO channels, one for inputs, one for outputs.

The problem of asynchronous testing has received attention since the inception of model-based testing [22]. A conceptually pioneering approach [27,26] considers a so-called queue operator, which adds infinite queues for inputs and outputs, so as to model the entirety of the possible asynchrony in interaction between tester and IUT. Modelling these queues explicitly however is challenging because of their infinite size. Indeed, it is left unanswered how the presented theory could be implemented without the need for restrictions on the model to be taken into account. Additionally, the queue context may induce that the test case generation algorithm [27] produces irrelevant test cases. This is because the queue context is always ready to receive any input action, which includes inputs which are impossible according to the specification at the current state (and states reachable by a sequence of output actions of the system), thereby inspecting executions which are irrelevant for testing conformance.

A conceptually different approach [20] proposes to divide the tester into an input test process and an output test process, both operating with finite buffer. This approach comes with appropriate implementation relations and test derivation procedures which however require a fault model for the tester architecture, and focusses on input-enabled specifications, i.e. systems where in every state, every input action is enabled, and without output cycles. Under these assumptions, completeness relative to the fault model can be achieved by a finite suite. Subsequent work [16] considers a single interaction sequence derived from a specification to generate asynchronous test cases. By applying the delay operator [1], outputs of the system are shifted along the sequence to emulate asynchrony. This enables relaxations of several of the restrictions on the specification model imposed before. Test case generation is incomplete but driven by coverage criteria w.r.t. the specification model. The need for repeated delay operator constructions is costly, and the proposed algorithm is only applied to offline test generation.

Another approach [28] considers IUTs which are internal-choice IOTS. Internal-choice IOTS do only have inputs enabled in quiescent states, i.e. in states which do not possess output transitions. With this assumption in place for both, IUT and specification, asynchronous testing and synchronous testing are equivalent and standard test case generation algorithms can be used. If the specification is not an internal-choice IOTS, the methodology becomes incomplete.

Asynchronous test case generation from test purposes is considered in [7] for specifications and IUTs obeying certain restrictions. A test purpose describes a set of interaction sequences which are to be investigated at the IUT. By incorporating the asynchronous behaviour directly at the finite test purpose the approach ensures finiteness of the test suite. A comparison of the complexity of different asynchronous testing approaches can be found in [14], together with an overview of several implementation relations for testing through asynchronous channels [15].

All the approaches discussed above either impose restrictions on the specification, or sacrifice expressiveness of the generated test suite, or work with potentially unboundedly growing representations. In this paper we propose a methodology for model-based testing of asynchronous system which does not impose restrictions on the specification model, while preserving soundness and completeness. The method we are going to present is rooted in the theory of the delay operator, but derives the test cases directly from an IOTS using a single input queue, and executes them. The approach is effective and computationally affordable, and can be applied to generate a test suite offline, or to construct a test case online, i.e. incrementally during test execution. We thereby construct on-the-fly the asynchronous transition system of the specification, based on its input queue behaviour only. Our methodology is driven by the practical needs arising in the context of the EnergyBus specification [6] which aims at establishing a common basis for the interchange and interoperation of electric devices in the context of energy management systems (EMS).

2 Synchronous Input-Output Conformance Testing

The basis for model-based testing is a precise specification of the IUT which unambiguously describes what an implementation may do and what it may not do.

Input-output transition systems. A common semantic model to describe the behaviour of a system are labeled transition systems (LTS). In the presence of inputs and outputs, a suitable variation is provided by *Input-Output Transition Systems* (IOTS).

Definition 1. *An* input-output transition system *is a 5-tuple* $\langle Q, L_?, L_!, T, q_0 \rangle$ *where*

- Q *is a countable, non-empty set of* states*;*
- $L_?$ *and* $L_!$ *are disjoint countable sets* ($L_? \cap L_! = \emptyset$) *of* input labels *and* output labels*, respectively;*
- $T \subseteq Q \times (L \cup \{\tau\}) \times Q$*, with* $\tau \notin L$*, is the* transition relation*, where* $L = L_? \cup L_!$*;*
- q_0 *is the* initial state*.*

The class of input-output transition systems with inputs in $L_?$ *and outputs in* $L_!$ *is denoted by* $\mathcal{IOTS}(L_?, L_!)$*.*

As usual, τ represents an unobservable internal action of the system. We write $q \xrightarrow{\mu} q'$ if there is a transition labelled μ from state q to state q', i.e., $(q, \mu, q') \in T$. The composition of transitions $q_1 \xrightarrow{\mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_{n-1}} q_n$ expresses that the system, when in state q_1, may end in state q_n, after performing the sequence of actions $\mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_{n-1}$, i.e. $\exists (q_i, \mu_i, q_{i+1}) \in T, i \leq n-1$. Due to non-determinism, it may be the case, that after performing the same sequence, the system may end in another state (or multiple such states): $q_1 \xrightarrow{\mu_1 \cdot \mu_2 \cdot \ldots \cdot \mu_{n-1}} q'_n$ with $q_n \neq q'_n$.

Traces and derived notions. Usually an IOTS can represent the entire behaviour of a system, including concrete interactions between system and environment. One such behaviour is represented by a so-called *trace*, of which we are only interested in its observable part, obtained by abstracting from internal actions of the system. Let $p = \langle Q, L_?, L_!, T, q_0 \rangle$ be an IOTS with $q, q' \in Q, L = L_? \cup L_!, a, a_i \in L$, and $\sigma \in L^*$. We write $q \overset{\epsilon}{\Rightarrow} q'$ to express that $q = q'$ or $q \xrightarrow{\tau \cdot \ldots \cdot \tau} q'$. $q \overset{a}{\Rightarrow} q'$ denotes the fact that $\exists q_1, q_2 \in Q : q \overset{\epsilon}{\Rightarrow} q_1 \xrightarrow{a} q_2 \overset{\epsilon}{\Rightarrow} q'$. This can be extended for a sequence of actions $q \xRightarrow{a_1 \cdot \ldots \cdot a_n} q'$ s.t. $\exists q_0, \ldots, q_n \in Q : q = q_0 \overset{a_1}{\Rightarrow} q_1 \overset{a_2}{\Rightarrow} \ldots \overset{a_n}{\Rightarrow} q_n = q'$. $q \overset{\sigma}{\Rightarrow}$ and $q \overset{\sigma}{\not\Rightarrow}$ are then defined as $\exists q' : q \overset{\sigma}{\Rightarrow} q'$ and $\nexists q' : q \overset{\sigma}{\Rightarrow} q'$, respectively.

Furthermore, $init(q)$ denotes the set of available transitions in a state q, i.e., $\{\mu \in L \cup \{\tau\} \mid q \xrightarrow{\mu}\}$. The set of traces starting in state q is then defined as $traces(q) =_{\text{def}} \{\sigma \in L^* \mid q \overset{\sigma}{\Rightarrow}\}$. For a given trace σ, the set of reachable states is given by the definition $q \,\mathbf{after}\, \sigma =_{\text{def}} \{q' \mid q \overset{\sigma}{\Rightarrow} q'\}$. The extension for starting in a set of states Q' is $Q' \,\mathbf{after}\, \sigma =_{\text{def}} \bigcup \{q \,\mathbf{after}\, \sigma \mid q \in Q'\}$. With $der(q)$ we denote the set of all reachable states from q, i.e., $\{q' \mid \exists \sigma \in L^* : q \overset{\sigma}{\Rightarrow} q'\}$.

Definition 2. *Let $p = \langle Q, L_?, L_!, T, q_0 \rangle$ be an IOTS with $q, q_1, q_2 \in Q$, $a \in L_?$, and $\sigma \in L^*$.*

- *q is* input-enabled*, iff $\forall a \in L_?.q \stackrel{a}{\Rightarrow}$.*
- *q is* input-progressive*, iff $\nexists \sigma \in L_!^+ : q \stackrel{\sigma}{\Rightarrow} q \wedge \nexists q_1, q_2 : q \stackrel{\epsilon}{\Rightarrow} q_1 \stackrel{\tau}{\rightarrow} q_2 \stackrel{\epsilon}{\Rightarrow} q$*
- *q is* fully-specified*, iff $L_? \subseteq init(q) \vee init(q) \cap L_? = \emptyset$*

An IOTS p is input-enabled, or input-progressive, or fully-specified if and only if all its reachable states are input-enabled, or input-progressive, or fully-specified, respectively. It is common practice to work with specifications modelled as IOTS without further restrictions while IUTs are often assumed to be represented as input-enabled IOTS.

Input-output conformance and quiescence. A specific conformance relation, input-out conformance (**ioco**) [24] dominates theoretical as well as practical work on model-based testing. It relates implementations with specifications with respect to the possible output behaviour observed after executing traces of the specification. In **ioco**, the output behaviour includes a designated output *quiescence*, abbreviated with the special label δ. *Quiescence* represents the situation when there is no output to observe at all. A state q is said to be quiescent, denoted by $\delta(q)$, iff $init(q) \cap L_! = \emptyset$, whereby $\delta \notin (L \cup \{\tau\})$. In this case we add the transition $q \stackrel{\delta}{\rightarrow} q$ for technical convenience. The set of possible outputs of a state q is then defined as $out(q) =_{\text{def}} \{a \in L_! \mid q \stackrel{a}{\rightarrow}\} \cup \{\delta \mid \delta(q)\}$, and this is lifted to sets of states P by $out(P) =_{\text{def}} \bigcup\{out(q) \mid q \in P\}$. Since quiescence is now interpreted as an additional observable output, we extend the definition for traces to *suspension traces*.

Definition 3. *Let $p = \langle Q, L_?, L_!, T, q_0 \rangle \in \mathcal{IOTS}(L_?, L_!)$. The* suspension traces *of p are given by $Straces(p) =_{\text{def}} \{\sigma \in (L \cup \{\delta\})^* \mid q_0 \stackrel{\sigma}{\Rightarrow}\}$.*

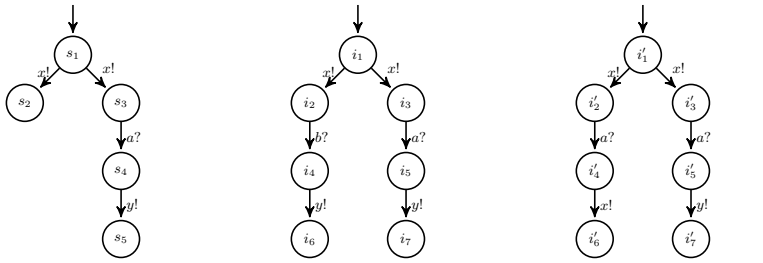
The definition of **ioco** then looks as follows:

Definition 4. *Given a set of input labels $L_?$ and a set of output labels $L_!$, the relation $\mathbf{ioco} \subseteq \mathcal{IOTS}(L_?, L_!) \times \mathcal{IOTS}(L_?, L_!)$ is defined for a specification s and an input-enabled implementation i as*

$$i \,\mathbf{ioco}\, s \Leftrightarrow_{\text{def}} \forall \sigma \in Straces(s) : out(i \,\mathbf{after}\, \sigma) \subseteq out(s \,\mathbf{after}\, \sigma).$$

Underspecification. Since **ioco** is defined for specifications without further restrictions and only takes suspension traces of the specification into account, the behaviour of an implementation after a trace not considered according to the specification is irrelevant for the relation. Figure 1 displays three IOTS (for readability we omitted the δ transitions as well as self-loops needed to ensure input-enabledness). The trace $x!b?$ is not in $Straces(s)$, i.e. it is *underspecified* w.r.t. s. So, any implementation of s is allowed to behave as it desires after that trace, and therefore $i \,\mathbf{ioco}\, s$. In contrast, the trace $x!a?$ is in $Straces(s)$ and the allowed outputs after $x!a?$ are $\{y!\}$. Therefore, $i' \,\mathbf{ioco}\, s$ does not hold. However, $s_1 \,\mathbf{after}\, x! = \{s_2, s_3\}$ and $a?$ is not specified in state s_2. Thus, one could argue

that the trace $x!a?$ actually constitutes a variant of underspecification, as well. This reasoning leads to the definition of **uioco** [24] which actually excludes such traces from consideration, and hence i' **uioco** s.



(a) specification s (b) implementation i (c) implementation i'

Figure 1: Variants of underspecification

Test generation and execution. Based on the definition of **ioco**, test cases are generated and executed in interaction with the IUT. A test case $t = \langle Q_t, L_?, L_!, T_t, v, t_0 \rangle$ is an extension of IOTS s.t. $\langle Q_t, L_?, L_!, T_t, t_0 \rangle \in \mathcal{IOTS}(L_?, L_!)$. Q_t is a set of states of Q, i.e. $Q_t \subseteq \mathcal{P}(Q)$ and $T_t \subseteq Q_t \times (L_? \cup L_! \cup \{\theta\}) \times Q_t$, where $\theta \neq \tau \neq \delta$ and $\theta \notin (L_? \cup L_!)$ is a special label synchronising with δ to detect quiescence. The function $v \in Q_t \times V$ is the *verdict label function* which assigns to each state of the test case a verdict in the set $V = \{\text{none}, \textbf{pass}, \textbf{fail}\}$. A test case is then generated as follows: The initial state of a test case consists of the τ-closure of the initial state of the specification, i.e. the set of all states which are reachable by a sequence of τ transitions. Then, one of the following three options is chosen nondeterministically. Either the current state is marked in the verdict label function as **pass** and test case generation is stopped; or an input action which is enabled in one of the current states of the specification is chosen and a transition for this action is added to the test. The successor state than consists of all valid successor states for the chosen (weak) input action. In addition, to be prepared to perform any output action of the IUT which might interrupt the input, for all outputs in $L_!$ a transition is added to all corresponding successor states of the specification. If the output is not foreseen by the specification, the successor state is a new state labeled with **fail**. For all valid successor states the test case generation algorithm is called recursively. The third option is to wait for an output of the system. For all outputs in $L_!$ and quiescence a transition is added to all corresponding successor states of the specification. Again, if the output is not foreseen by the specification, the successor state is a new state labeled with **fail** and for all valid successor states the test case generation algorithm is called recursively. States which are neither labeled with **pass** nor **fail** are marked with "none" in the verdict label function.

An execution of a test case is then the parallel composition of the test case and the IUT. A *test run* is than any trace of the parallel composition which ends in a state which is labeled with **pass** or **fail**. An IUT then passes a test case if and only if all possible test runs lead to states labeled with **pass**. It fails the test case otherwise. By assuming some kind of fairness, an IUT will reveal sooner or later all its nondeterministic behaviour when executed with a test case.

3 Asynchronous Input-Output Conformance Testing

The traditional synchronous testing theory is not applicable when testing communication is asynchronous [27]. The implementation relations used for synchronous testing are not testable in an asynchronous context, and test cases derived from specifications to be used for synchronous testing do reject correct implementations when tested asynchronously. Therefore, the asynchronous communication behaviour needs to be directly taken into account within the conformance relation and the test case generation.

Queue operator. One approach to include the asynchronous communication behaviour of a system applies the so-called queue operator [26]. This takes an IOTS s and yields an IOTS s' which behaves like s in the context of an input queue and an output queue, both with infinite capacity. The behaviour of $s \in \mathcal{IOTS}(L_?, L_!)$ in a queue context $[_{\sigma_!} \ll s \ll_{\sigma_?}]$ (abbreviated by $Q(s)$), where $\sigma_! \in L_!^*$ and $\sigma_? \in L_?^*$ represent the input and output queue state as words of arbitrary length over inputs, respectively outputs. It is derived by applying the following axioms and inference rules:

$$A1 \quad [_{\sigma_!} \ll s \ll_{\sigma_?}] \xrightarrow{a} [_{\sigma_!} \ll s \ll_{\sigma_? \cdot a}] \quad a \in L_? \qquad A2 \quad [_{x \cdot \sigma_!} \ll s \ll_{\sigma_?}] \xrightarrow{x} [_{\sigma_!} \ll s \ll_{\sigma_?}] \quad x \in L_!$$

$$I1 \quad \frac{s \xrightarrow{\tau} s'}{[_{\sigma_!} \ll s \ll_{\sigma_?}] \xrightarrow{\tau} [_{\sigma_!} \ll s' \ll_{\sigma_?}]} \qquad I2 \quad \frac{s \xrightarrow{a} s'}{[_{\sigma_!} \ll s \ll_{a \cdot \sigma_?}] \xrightarrow{\tau} [_{\sigma_!} \ll s' \ll_{\sigma_?}]} \quad a \in L_?$$

$$I3 \quad \frac{s \xrightarrow{x} s'}{[_{\sigma_!} \ll s \ll_{\sigma_?}] \xrightarrow{\tau} [_{\sigma_! \cdot x} \ll s' \ll_{\sigma_?}]} \quad x \in L_!$$

Obviously, the resulting state space of $Q(s)$ is infinite. Looking at the output queue, this infinity problem materialises for systems having at least one output action on a cycle, i.e. $\exists \sigma_1, \sigma_2 \in L^*, x \in L_!, q, q_1, q_2 \in der(s) : q \stackrel{\sigma_1}{\Longrightarrow} q_1 \xrightarrow{x} q_2 \stackrel{\sigma_2}{\Longrightarrow} q$. The state space however remains finite at any finite depth of the testing process, unless the system contains output loops. In the latter case, the weak trace construction in the testing theory leads to an immediate explosion, rooted in an infinite branching. This however can be prevented by putting restrictions on the specification, namely input-progressiveness. The input queue, in turn, is always ready to receive an input, thus, growing to unbounded size. In addition, the input capability of $Q(s)$ is in no sense related to the actual structure of the underlying system s. Thus, providing input actions which are not specified in s

at the current state, may lead to the execution of underspecified traces w.r.t. s, which are being irrelevant for testing conformance. When only considering input-enabled or fully-specified specifications, this discrepancy is obviously not there. Therefore, using the queue operator as basis for asynchronous testing seems to be rather inconvenient.

Delay operator. A conceptually different way of including asynchronous communication is the *delay* operator [1]. Instead of being directly applied to an IOTS, the delay operator works on the traces of a system. For a set of action sequences, e.g. traces, $E \subseteq L^*$ and a subset $L' \subseteq L$, the operator $delay[L'] : 2^{L^*} \to 2^{L^*}$ gives the smallest superset of E s.t. for $\sigma_1, \sigma_2 \in L^*$, any $a \in L \setminus L'$ and $a_1 \in L'$:

$$\sigma_1 a_1 a \sigma_2 \in delay[L'](E) \Rightarrow \sigma_1 a a_1 \sigma_2 \in delay[L'](E).$$

Given a set of traces E and a set of actions L', $delay[L'](E)$ calculates a set of traces where actions in L' are shifted towards the end of a trace in E while keeping the relative order of actions in $L \setminus L'$. For an IOTS $p = \langle Q, L_?, L_!, T, q_0 \rangle$, the observable traces in a queue context can then be defined as $traces(Q(p)) = pref(delay[L_!](traces(p)))$, where $pref(U)$ is the prefix closure of a set of traces U. On the other hand, when a trace σ has been observed, p can have executed any of the traces in $delay[L_?](\sigma L_!^*) \cap traces(p)$.

Since the delay operator directly operates on traces of a system, genuine underspecified traces are excluded. However, due to delayed input actions, it is still possibly that an execution is steered away from specified traces, which has to be dealt with in the test case generation algorithm [16]. Again, this problem does not arise when only considering input-enabled or fully-specified specifications. If assuming input progressive specifications (as in [16]) the test algorithm can be made to assign verdicts in quiescent states of an IUT only. But this assumption is otherwise not needed for generating test cases from given traces, which are in fact, finite. Nevertheless, the test generation algorithm is only suitable for offline test case generation due to the need for repeated calculation of delayed traces and their intersection with traces of the system.

Our approach.

The method we are going to present is a practical approach to deriving test cases directly from an IOTS, offline or online, while theoretically being (almost) equivalent to applying the delay operator to the specification traces. Notably, we neither have to propose any restrictions on the specification, nor do we examine underspecified traces of the system, nor can our tester become trapped in an immediate growth of the state space due to infinite branching. At the same time, the approach is effective and computationally affordable.

Input queue context. The starting point of our approach is the construction of the *input queue context* of a system s which represents the asynchronous communication behaviour of s in the presence of an infinite input queue.

Definition 5. *For an IOTS* $s = \langle Q, L_?, L_!, T, q_0 \rangle$*, the* input queue context *is the smallest IOTS* $s_{\ll} = \langle Q_{\ll}, L_?, L_!, T_{\ll}, q_{0\ll} \rangle$ *where* $Q_{\ll} \subseteq (Q \times L_?^*), \sigma \in L_?^*, \mu \in L \cup \{\tau\}$ *s.t.:*

- $q_{0\ll} = (q_0, \epsilon)$ *and* $q_{0\ll} \in Q_{\ll}$
- $T_{\ll} = \{((q,\sigma), \tau, (q',\sigma)) \mid q, q' \in Q, q \xrightarrow{\tau} q'\}$
  $\cup\ \{((q,\sigma), a, (q,\sigma a)) \mid q \in Q, a \in L_?\}$
  $\cup\ \{((q,a\sigma), \tau, (q',\sigma)) \mid q, q' \in Q, q \xrightarrow{a} q'\}$
  $\cup\ \{((q,\sigma), x, (q',\sigma)) \mid q, q' \in Q, x \in L_!, q \xrightarrow{x} q'\}$
- $q \in Q_{\ll} \wedge (q, \mu, q') \in T_{\ll} \Rightarrow q' \in Q_{\ll}$

The input queue context of a system behaves exactly as the queue context derived by the queue operator, but without applying the rules A2 and I3. Interestingly, despite the fact that for a system s, $s_{\ll}$ and $Q(s)$ are not isomorphic, the observable trace behaviour of both resulting systems is actually equivalent.

Proposition 1. *Let* $s \in \mathcal{IOTS}(L_?, L_!)$

1. $traces(Q(s)) = traces(s_{\ll})$
2. $Straces(Q(s)) = Straces(s_{\ll})$

This follows from the observation already mentioned when introducing the delay operator: $traces(Q(p)) = pref(delay[L_!](traces(p)))$.

Shifting outputs. The core property exploited by our approach (already appearing above) is that the asynchronous behaviour can be modelled by only shifting one action set, i.e. outputs, w.r.t. the other action set. To establish this shift, it is actually irrelevant which set of actions is buffered. Notably, this means, that we could equally well model the same phenomena by an output queue context instead of an input queue context, but requiring input-enabled specifications. However, inputs are under full control of the tester while outputs are under the control of the IUT. So, with an output queue context we would still face the immediate explosion problem due to infinite branching in the test generation algorithm when dealing with output loops. This is not the case for the input queue context as defined above. Thus, the input queue context is computable using the standard test case generation algorithm proposed for synchronous communication.

Asynchronous transition system. In comparison with the delay operator approach, we however still have the issue with unnecessarily testing underspecified traces. In order to remedy this, we define the *asynchronous transition system* on top of the input queue context.

Definition 6. *Given an IOTS* $s = \langle Q, L_?, L_!, T, q_0 \rangle$ *and its input queue context* $s_{\ll} = \langle Q_{\ll}, L_?, L_!, T_{\ll}, q_{0\ll} \rangle$*, the* asynchronous transition system *(ATS) is the smallest IOTS* ${}_{\ll}s_{\ll} = \langle {}_{\ll}Q_{\ll}, L_?, L_!, {}_{\ll}T_{\ll}, {}_{\ll}q_{0\ll} \rangle$ *where* ${}_{\ll}Q_{\ll} \subseteq \mathcal{P}(Q \times L_?^*)$*,* $a \in L_?, x \in L_!, \sigma, \sigma_1, \sigma_2 \in L^*, \mu \in L \cup \{\tau, \delta\}$ *s.t.:*

- $\ll q_{0\ll} = q_{0\ll}$ **after** ϵ
- $\ll T_{\ll} = \{(\hat{q}, a, \hat{q}') \mid \exists (q, \epsilon) \in \hat{q} : q \xrightarrow{a} \wedge \forall (q', \sigma_1) \in \hat{q} : (q', \sigma_1) \overset{a}{\Rightarrow} (q'', \sigma_2) \implies (q'', \sigma_2) \in \hat{q}'\}$
  $\cup \{(\hat{q}, x, \hat{q}') \mid \exists q \in \hat{q} : q \xrightarrow{x} \wedge\ \hat{q}' = \hat{q}$ **after** $x\}$
  $\cup \{(\hat{q}, \delta, \hat{q}') \mid \exists (q, \epsilon) \in \hat{q} : \delta(q)\ \wedge\ \hat{q}' = \{(q', \epsilon) \mid (q', \epsilon) \in \hat{q} \wedge \delta(q')\}\}$
- $q \in_{\ll} Q_{\ll} \wedge (q, \mu, q') \in_{\ll} T_{\ll} \Rightarrow q' \in_{\ll} Q_{\ll}$

The initial state of the ATS is the τ-closure of the initial state of the underlying input queue context. Continuing from here, the ATS is further constructed by adding transitions for the asynchronous behaviour and by eliminating nondeterminism (putting all successor states together). A state in the ATS can receive an input action, iff there is one state in the input queue context which has an empty input queue. Then, the successor state consists of all the successor states of the input queue context after the corresponding input transition. By restricting the input functionality in this way, we make sure that we always follow specified traces of the system, i.e. we are not examining genuine underspecified traces. The ATS can issue an output action, again, iff there is a state in the input queue context which enables this output action. All states reachable by this output transition form the new successor state, including states reachable by successive τ transitions inherited from the underlying system or from the opportunity of the input queue context to process inputs present in the input queue. The last part of the definition of the above transition relation deals with our interpretation of quiescence in the asynchronous communication setting. When quiescence is observed, we do not only assume that the system is in no state which can produce an output, but we also assume the input queue to be as processed as possible. Thus, we only can observe quiescence in a state of an input queue context which is quiescent in the perspective of the underlying system and whose input queue is either empty, or the next input action in the queue is blocking w.r.t. currently enabled input transitions. If the input queue is not empty, we can conclude that this state configuration represents an underspecified trace. Since it is quiescent, it can not evolve by further output and it does not have a suitable input action enabled w.r.t. the specification, thus it must be an underspecified trace. Therefore, we restrict quiescence further to only quiescent states with empty input queues.

Passing underspecified behaviour. Regarding unintended examination of underspecified traces due to the asynchronous communication, there is one situation left which we did not take care of so far. When receiving an output from the system, which is not foreseen in any of the current states, this is seen as an illegal input. However, when we already drifted in an underspecified trace, the reception of such an output should lead to the verdict "pass". Such a situation is identified by inspecting the input queues of the current states. If there is no state with an empty input queue, we know that there is no trace of the specification corresponding to the current execution. Note, a valid output in such a situation will be processed further, since we could still be on a valid trace with pending inputs not received so far by the IUT. Technically speaking, we observed a trace

σ_1 s.t. $delay[L_?](\sigma_1) \cap traces(s) = \emptyset$, but their might be a sequence of output actions $\sigma_2 \in L_!^+$ s.t. $delay[L_?](\sigma_1\sigma_2) \cap traces(s) \neq \emptyset$. Taking care of this situation is done during the test case generation.

Role of ATS. Since the asynchronous transition system directly takes all nondeterminism and weak transitions in the input queue context into account, it represents an intermediate step to the *test graph* of our testing approach.

Test generation algorithm. Our test case generation algorithm is provided as Algorithm 1. Starting with an empty test case, we set the initial state to the τ-closure of the initial state of the system with empty input queues. Following the structure of the test case generation algorithm for synchronous communication, we then nondeterministically choose between ending with verdict **pass** (lines 11-14), providing an enabled input to the IUT and recursively construct the following subtree (lines 15-20), or add transitions for all outputs (including quiescence) (lines 21-52) and recursively construct the following subtree for valid outputs (lines 23-28 and 38-42). The provided algorithm is suitable for both, offline and online test case generation. For offline test case generation, it is common to only explore one subtree of valid outputs and stop with the verdict **pass** for the other output actions.

Asynchronous input-output conformance. With the test case generation algorithm in place, what is missing is the definition of the actual conformance relation we are testing for, which we call **asy**nchronous **i**nput-**o**utput **co**nformance (**asyioco**). First, we need an additional definition.

Definition 7. *For a given IOTS $s \in \mathcal{IOTS}(L_?, L_!)$ and a suspension trace $\sigma \in Straces(s)$, the set of* asynchronous trace executions *is defined as the smallest subset of $Straces(s)$ s.t. for $\sigma_1, \sigma_2 \in (L_? \cup L_!)^*$, any $x \in L_!$ with $x \neq \delta$ and $a \in L_?$:*

$$\sigma \in asyexec_s(\sigma)$$
$$\sigma_1 a x \sigma_2 \in asyexec_s(\sigma) \implies (\sigma_1 x a \sigma_2 \in Straces(s) \implies \sigma_1 x a \sigma_2 \in asyexec_s(\sigma))$$

Here we directly encode the delay operator into the definition of asynchronous trace executions to point out, that input actions can not be shifted along quiescence.

Definition 8. *Given a set of input labels $L_?$ and a set of output labels $L_!$, the relation* **asyioco** $\subseteq \mathcal{IOTS}(L_?, L_!) \times \mathcal{IOTS}(L_?, L_!)$ *is defined for a specification s and an input-enabled implementation i as:*

$$i \,\mathbf{asyioco}\, s \Leftrightarrow_{\text{def}} \forall \sigma \in Straces(s).\ out(i \,\mathbf{after}\, asyexec_s(\sigma)) \subseteq out(s \,\mathbf{after}\, asyexec_s(\sigma))$$

In words, this definition says that an IUT conforms to a specification, iff for each observable behaviour of the specification, the possible outputs of the IUT after asynchronously executing this trace w.r.t. specified traces are foreseen by the specification after all possible asynchronous executions.

Algorithm 1: Test case generation algorithm for asynchronous communicating systems through queues

```
1  Function TCG(s)
       Input  : IOTS s = ⟨Q, L?, L!, T, q0⟩
       Output: Test case t = ⟨Qt, L!, L?, Tt, v, t0⟩
2      t0 ← (q0, ϵ) after ϵ
3      Qt ← {t0}
4      Tt, v ← ∅
5      ⟨Qt, L!, L?, Tt, v, t'⟩ ← reTCG(s, ⟨Qt, L!, L?, Tt, v, t0⟩)
6      return reTCG(s, ⟨Qt, L!, L?, Tt, v, t0⟩)
7  end
8
9  Function reTCG(s, t)
       Input  : IOTS s = ⟨Q, L?, L!, T, q0⟩,
                Test case t = ⟨Qt, L!, L?, Tt, v, t0⟩
       Output: Test case t' = ⟨Qt, L!, L?, Tt, v, t0⟩
10     choice {pass, input, output} do
11         case pass do
12             v ← v ∪ {(t0, pass)}
13             return ⟨Qt, L!, L?, Tt, v, t0⟩
14         end
15         case input ∧ ∃a ∈ L?, (q, ϵ) ∈ t0. q -a-> do
16             t' ← t0 after a
17             Qt ← Qt ∪ {t'}
18             Tt ← Tt ∪ {(t0, a, t')}
19             return reTCG(s, ⟨Qt, L!, L?, Tt, v, t'⟩)
20         end
21         otherwise do
22             v ← v ∪ {(t0, none)}
23             for x ∈ L! : ∃(q, σ) ∈ t0 : q -x-> do
24                 t' ← t0 after x
25                 Qt ← Qt ∪ {t'}
26                 Tt ← Tt ∪ {(t0, x, t')}
27                 ⟨Qt, L!, L?, Tt, v, t'⟩ ← reTCG(s, ⟨Qt, L!, L?, Tt, v, t'⟩)
28             end
29             for x ∈ L! : ∄(q, σ) ∈ t0 : q -x-> do
30                 Qt ← Qt ∪ {t'}
31                 Tt ← Tt ∪ {(t0, x, t')}
32                 if ∃(q, ϵ) ∈ t0 then
33                     v ← v ∪ {(t', fail)}
34                 else
35                     v ← v ∪ {(t', pass)}
36                 end
37             end
38             if ∃(q, ϵ) ∈ t0 : δ(q) then
39                 t' ← {(q, ϵ) ∈ t0. δ(q)}
40                 Qt ← Qt ∪ {t'}
41                 Tt ← Tt ∪ {(t0, δ, t')}
42                 ⟨Qt, L!, L?, Tt, v, t'⟩ ← reTCG(s, ⟨Qt, L!, L?, Tt, v, t'⟩)
43             else if ∃(q, ϵ) ∈ t0 ∧ ∀(q', ϵ) ∈ t0 : ¬δ(q') then
44                 Qt ← Qt ∪ {t'}
45                 Tt ← Tt ∪ {(t0, δ, t')}
46                 v ← v ∪ {(t', fail)}
47             else
48                 Qt ← Qt ∪ {t'}
49                 Tt ← Tt ∪ {(t0, δ, t')}
50                 v ← v ∪ {(t', pass)}
51             end
52         end
53     end
54     return ⟨Qt, L!, L?, Tt, v, t0⟩
55 end
```

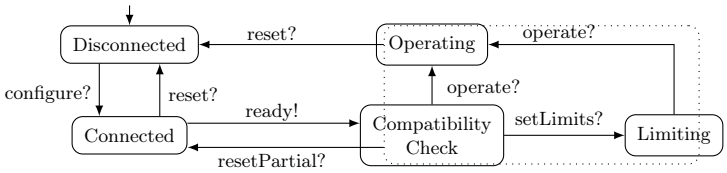


Figure 2: EnergyBus Energy Management System FSA (simplified)

Proposition 2. *Let specification s and implementation $i \in \mathcal{IOTS}(L_?, L_!)$. The following holds for input-enabled i and for s being*

1. *input-enabled:* $i \,\mathbf{asyioco}\, s \Leftrightarrow i \leq_{qcst} s \Leftrightarrow Q(i)\,\mathbf{ioco}\,Q(s)$
2. *fully-specified:* $i \,\mathbf{asyioco}\, s \Leftarrow i \leq_{qcst} s \Leftrightarrow Q(i)\,\mathbf{ioco}\,Q(s)$
3. *partially-specified:* $i \,\mathbf{asyioco}\, s \Leftarrow Q(i)\,\mathbf{ioco}\,Q(s) \wedge i \leq_{qcst} s \Leftarrow Q(i)\,\mathbf{ioco}\,Q(s)$

The definition of **asyioco** is similar to *queue-context suspension trace inclusion*($\leq_{qcst}$) [16] if restricting to fully or partially specified IOTS. As already discussed, these settings either exclude the need to handle underspecification of the specification or they exclude underspecification in its entirety. The latter can thus be considered as an asynchronous version of **uioco** [24]. In contrast, **asyioco** follows the **ioco** philosophy and only exclude traces which in any case are underspecified. Therefore, we think **asyioco** is a more natural extension of **ioco** to asynchronous communication.

We claim that the test case generation algorithm we presented is sound and complete w.r.t. **asyioco**. The latter feature is of only a theoretical nature. since completeness can only be achieved by generating an infinite amount of test cases, which can in practice not be executed in finite time.

4 EnergyBus Case Study

The EnergyBus specification [6] aims at establishing a common basis for the interchange and interoperation of electric devices in the context of energy management systems (EMS). The central and innovative role of EnergyBus is the transmission and management of electrical power: the purpose of its protocol suite is not just to transmit data, but in particular to manage the safe access to electricity and its distribution inside an EnergyBus network. Conceptually, EnergyBus extends the CANopen architecture in terms of *CANopen application profiles* endorsed by the CiA association [6]. Among these, the "Pedelec Profile 1" (PP1) is very elaborate, targeting a predominant business context, which is also at the centre of ongoing international standardisation efforts as part of IEC/IS/TC69/JPT61851-3.

Formal EnergyBus Specification. Since EnergyBus is defined as a layer on top of CANopen, EnergyBus documentation [6] as well as the CANopen documentation [4] have to be taken into account for formal modelling. Both specifications are provided as informal combinations of text, protocol flow charts, data

tables, and finite state automata (FSA). The definitions include several data structures and various services for e.g. initial configuration, data exchange, and basic communication capability control. Figure 2 presents a simplified view on a core EnergyBus control functionality. Our formal model of EnergyBus uses the Modest modelling language.

Aside from the basic control functionality, the EnergyBus protocol is all about data. To overcome the state space explosion problem, we applied several abstraction techniques to appropriate areas of our model, transferring the complexity from the Modest model to the adapter component.

Results. Already during the model construction phase, our work [11] uncovered several issues concerning the (at that time current version of the) EnergyBus specification documents. On the one hand there were gaps in the specification, preventing some parts of the services to be modelled to a reasonable extent; on the other hand there were ambiguities in some parts of the specification, possibly inducing non-interoperability. These have been reported so as to be corrected in standardisation. The actual test runs then revealed two different types of further errors. The first type were traditional implementation bugs of a non-severe nature. The second type of observed errors were intricately related to the hard- and software hierarchy of the test and IUT architecture, i.e. the CAN bus system. They can be viewed as spurious **fail** verdicts rooted in the fact that the different communication layers made the traditional model-based testing assumption of synchronous communication unsound. One of these spurious **fail** verdicts can be illustrated by means of Figure 2. An already *configure*d device can transit from state *Connected* to state *Compatibility Check* by announcing being *ready*, or it can be ordered to switch back to state *Disconnected* via a *reset*. One test execution trace we observed was *configure*?.*reset*?.*ready*!. In the synchronous testing approach this should end in a **fail** verdict, because after performing the prefix *configure*?.*reset*? the set of potential states where the IUT might be in is $\{Disconnected\}$, and *ready!* is not part of the *out* set of this state. However, the behaviour obviously represents the case where the device already switched to the state *Compatibility Check*, but the tester issued the *reset?* command before the *ready!* output arrived. In our asynchronous approach, the above prefix we would instead lead to the set $\{(Disconnected, \epsilon), (Connected, reset?), (Disconnected, configured?.reset?)\}$. And since *ready!* is in the *out* set of *Connected*, this turns *ready!* into a valid output. We can thus conclude the test with a **pass** verdict, or, more importantly, we can continue testing from the set $\{(Disconnected, \epsilon), (CompatibilityCheck, reset?)\}$.

The above asynchronicity phenomena indeed triggered the development and implementation of the asynchronous model-based testing method discussed here. New test runs with this improved methodology confirmed the already uncovered implementation bugs that have been reported and fixed. Since the spurious **fail** runs no longer appear, we have invested in a better analysis of the remaining errors. A newly identified type of error was rooted in two distinct interpretations of the EnergyBus basic device initialisation and the core EnergyBus device control leading to incompatible implementations. To pinpoint this, we

developed two different models of the specification and continued testing with the respective version. In addition, we observed that some CAN implementations take the liberty to reorder messages within responses, so that consecutive messages passed by an IUT's application to its local CAN controller may be sent out in reverse order, which made manual inspection still be needed to definitely rule out spurious **fail** verdicts.

Asynchronous Testing with Motest. The presented approach is implemented in our model-based testing tool Motest, which is part of the Modest Toolset [13]. The tool platform is based on the Modest modelling language [12] and encompasses several tools for formal modelling, simulation and verification of systems. The Modest Toolset is available at www.modestchecker.net.

Due to our tight interaction with the EnergyBus consortium we had the opportunity to apply motest to a variety of prototypes and retail devices implementing EnergyBus as soon as those became available. Lately we went a step further, by making motest together with the specification models available free-of-charge to the entirety of the EnergyBus e.V. association, so as to enable its direct use by association members as part of their in-house testing. The feedback collected is very encouraging.

5 Conclusion

This paper has discussed a novel, practical approach to model-based testing for asynchronous communicating systems. Test cases are generated directly on the model of the specification in a way that resembles the theory of the delay operator. We presented a pseudo-code algorithm together with the definition of **asyioco**, for which our algorithm produces sound and theoretically complete test suites. Our algorithm is implemented in the Motest tool as part of the Modest Toolset. As we discussed, this tool is in use for model-based conformance testing of the EnergyBus standard over CAN.

Acknowledgments. This work is supported by the ERC Advanced Grant powver (695614) and the Sino-German project CAP (GZ 1023).

References

1. S. Balemi. *Control of Discrete Event Systems: Theory and Application.* PhD thesis, Swiss Federal Inst. of Technology, Zurich, Switzerland, 1992.
2. G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.
3. M. Bijl, A. Rensink, and J. Tretmans. Action Refinement in Conformance Testing. In *Testing of Communicating Systems*, volume 3502 of *LNCS*, pages 81–96. Springer, 2005.
4. CAN in Automation Int. Users and Manufacturers Group e.V. *CiA 301 CANopen Application Layer and Comm. Profile, v. 4.2.0*, 2011.

5. CAN in Automation Int. Users and Manufacturers Group e.V. *CiA 305 Layer setting services (LSS) and protocols, v. 3.0.0*, 2013.
6. CAN in Automation Int. Users and Manufacturers Group e.V. and EnergyBus e.V. *CiA 454 Draft Standard Proposal Application profile for energy management systems – doc. series 1-14, v. 2.0.0*, 2014.
7. A. da Silva Simão and A. Petrenko. From test purposes to asynchronous test cases. In *ICST 2010 Workshops Proceedings*, pages 1–10. IEEE Computer Society, 2010.
8. R. De Nicola. Extensional equivalences for transition systems. *Acta Inf.*, 24(2):211–237, 1987.
9. R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theor. Comput. Sci.*, 34:83–133, 1984.
10. M.-C. Gaudel. Testing can be formal, too. In *TAPSOFT 1995*, LNCS 915:82–96. Springer, 1995.
11. A. Graf-Brill, H. Hermanns, and H. Garavel. A model-based certification framework for the EnergyBus standard. In *FORTE 2014*, LNCS 8461:84–99. Springer, 2014.
12. E. M. Hahn, A. Hartmanns, H. Hermanns, and J.-P. Katoen. A compositional modelling and analysis framework for stochastic hybrid systems. *Formal Methods in System Design*, 43(2):191–232, 2013.
13. A. Hartmanns and H. Hermanns. The Modest Toolset: An integrated environment for quantitative modelling and verification. In *TACAS*, LNCS 8413:593–598. Springer, 2014.
14. R. M. Hierons. The complexity of asynchronous model based testing. *Theor. Comput. Sci.*, 451:70–82, 2012.
15. R. M. Hierons. Implementation relations for testing through asynchronous channels. *Comput. J.*, 56(11):1305–1319, 2013.
16. J. Huo and A. Petrenko. On testing partially specified IOTS through lossless queues. In *TestCom 2004*, LNCS 2978:76–94. Springer, 2004.
17. C. Jard and T. Jéron. TGV: theory, principles and algorithms. *STTT*, 7(4):297–315, 2005.
18. R. Langerak. A testing theory for LOTOS using deadlock detection. In *PSTV 1989*, 87–98. North-Holland, 1989.
19. A. Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In *MOVEP 2000*, LNCS 2067:196–205. Springer, 2000.
20. A. Petrenko, N. Yevtushenko. Queued testing of transition systems with inputs and outputs. In *Proc. of FATES 2002*, 79–93., 2002.
21. I. Phillips. Refusal testing. *Theor. Comput. Sci.*, 50:241–284, 1987.
22. J. Tretmans. *A formal approach to conformance testing.* PhD thesis, University of Twente, Enschede, Netherlands, 1992.
23. J. Tretmans. Testing concurrent systems: A formal approach. In *CONCUR 1999*, LNCS:1664:46–65. Springer, 1999.
24. J. Tretmans. Model-based Testing with Labelled Transition Systems. In *Formal Methods and Testing*, LNCS:4949:1–38. Springer-Verlag, 2008.
25. J. Tretmans and E. Brinksma. TorX: Automated Model Based Testing – Côte de Resyste, 2003.
26. J. Tretmans and L. Verhaard. A queue model relating synchronous and asynchronous communication. In *PSTV 1992*, 131–145. North-Holland, 1992.
27. L. Verhaard, J. Tretmans, P. Kars, and E. Brinksma. On asynchronous testing. In *IWPTS 1992*, 55–66. North-Holland, 1992.
28. M. Weiglhofer and F. Wotawa. Asynchronous input-output conformance testing. In *COMPSAC 2009*, 154–159. IEEE Computer Society, 2009.