TESTOR: A Modular Tool for On-the-Fly Conformance Test Case Generation

Lina Marsso, Radu Mateescu, and Wendelin Serwe

Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP*, LIG, 38000 Grenoble, France

Abstract. We present TESTOR, a tool for on-the-fly conformance test case generation, guided by test purposes. Concretely, given a formal specification of a system and a test purpose, TESTOR automatically generates test cases, which assess using black box testing techniques the conformance to the specification of a system under test. In this context, a test purpose describes the goal states to be reached by the test and enables one to indicate parts of the specification that should be ignored during the testing process. Compared to the existing tool TGV, TESTOR has a more modular architecture, based on generic graph transformation components, is capable of extracting a test case completely on the fly, and enables a more flexible expression of test purposes, taking advantage of the multiway rendezvous. TESTOR has been implemented on top of the CADP verification toolbox, evaluated on three published case-studies and more than 10000 examples taken from the non-regression test suites of CADP.

1 Introduction

Model-Based Testing [7] is a validation technique taking advantage of a model of a system (both, requirements and behavior) to automate the generation of relevant test cases. This technique is suitable for complex industrial systems, such as embedded systems [45] and automotive software [35]. Using formal models for testing is required for certification of safety-critical systems [36]. Conformance testing aims at extracting from a formal model of a system a set of test cases to assess whether an actual implementation of the system under test (SUT) is conform to the model, using black-box testing techniques (i.e., without knowledge of the actual code of the SUT). This approach is particularly suited for nondeterministic concurrent systems, where the behavior of the SUT can be observed and controlled by a tester only via dedicated interfaces, named points of control and observation.

Often, the formal model is an IOLTS (Input/Output Labeled Transition System), where transitions between states of the system are labeled with an action classified as input, output, or internal (i.e., unobservable, usually denoted by τ). In this setting, the most prominent conformance relation is input-output

* Institute of Engineering Univ. Grenoble Alpes

conformance (**ioco**) [39,41]. The theory underlying **ioco** is well established, implemented in several tools [25,2,28,22,1], and still actively used, as witnessed by a series of recent case studies [10,9,20,38,27].

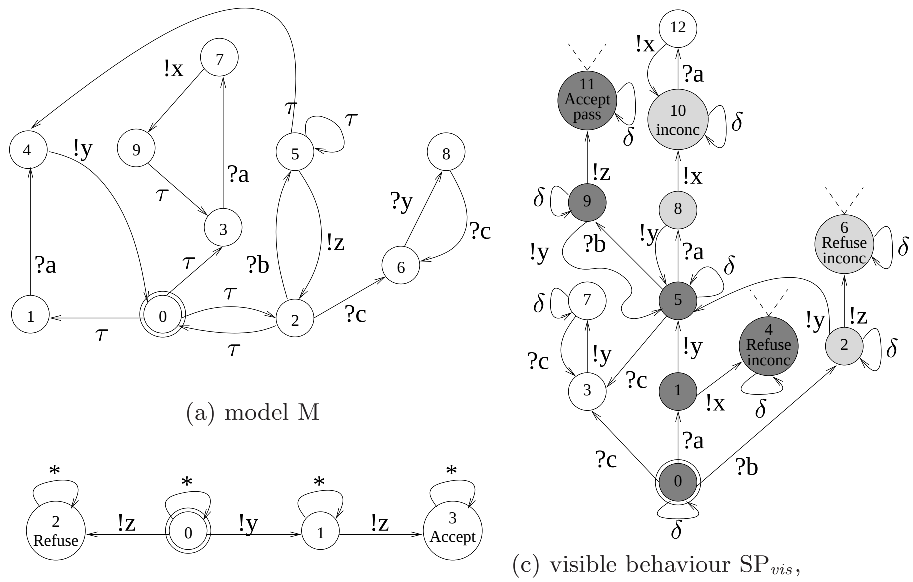
As regards asynchronous systems, i.e., systems consisting of concurrent processes with message-passing communication, there exist two different approaches to model-based conformance testing: *coverage-oriented approaches* run the test(s) to stimulate the SUT until a coverage goal has been reached, whereas *test purpose guided approaches* use test suites, each test of which terminates with a verdict (passed, failed, or inconclusive). The generation of tests from the model can be carried out *offline*, before executing them against the SUT, or *online* [28] during their execution, by combining the exploration of the model and the interaction with the SUT.

In this paper, we present TESTOR, a tool for on-the-fly conformance test case generation guided by test purposes, which, following the approach of TGV [25], characterize some state(s) of the model as accepting. The generated test cases are automata that attempt to drive a SUT towards these states. TESTOR extends the algorithms of TGV to extract test cases completely on the fly (i.e., during test case execution against the SUT), making TESTOR suitable for online testing. TESTOR is constructed following a modular architecture based on generic, recent, and optimized graph manipulation components. This also makes the description of test purposes more convenient, by replacing the specific synchronous product of TGV and taking advantage of the multiway rendezvous [23,18], a powerful primitive to express communication and synchronization among a set of distributed processes. TESTOR was built on top of the OPEN/CAESAR [15] generic environment for on-the-fly graph manipulation provided by the CADP [16] verification toolbox.

The remainder of the paper is organized as follows. Section 2 recalls the essential notions of the underlying theory. Section 3 presents the architecture, main algorithms, and implementation of TESTOR, and gives some examples. Section 4 describes various experiments to validate TESTOR and compare it to TGV. Section 5 compares TESTOR to existing test generation approaches. Finally, Section 6 gives some concluding remarks and future work directions.

2 Background: Essential Definitions of [25]

Conformance testing checks that a SUT behaves according to a formal reference model (M), which is used as an oracle. We use Input-Output Labelled Transition Systems (IOLTS) [25] to represent the behavior of the model M. We assume that the behavior of the SUT can also be represented as an IOLTS, even if it is unknown (the so-called testing hypothesis [25]). An IOLTS (Q, A, T, q_0) consists of a set of states Q, a set of actions A, a transition relation $T \subseteq Q \times A \times Q$, and an initial state $q_0 \in Q$. The set of actions is partitioned in $A = A_\mathrm{I} \cup A_\mathrm{O} \cup \{\tau\}$, where $A_\mathrm{I}, A_\mathrm{O}$ are the subsets of input and output actions, and τ is the internal (unobservable) action. A transition $(q_1, b, q_2) \in T$ (also noted $q_1 \xrightarrow{b} q_2$) indicates that the system can move from state q_1 to state q_2 by performing action b. Input



(a) model M

(b) test purpose TP

(c) visible behaviour SP_{vis},
complete test graph CTG (gray),
and a test case TC (dark gray)

Fig. 1. Example of test case selection (taken from [25])

(resp. output) actions are noted $?a$ (resp. $!a$). In the sequel, we consider the same running example as [25], whose IOLTS model M is shown on Figure 1(a).

Input actions of the SUT are controllable by the environment, whereas output actions are only observable. Testing allows one to observe the execution traces of the SUT, and also to detect *quiescence*, i.e., the presence of deadlocks (states without successors), outputlocks (states without outgoing output actions), or livelocks (cycles of internal actions). The quiescence present in an IOLTS L (either the model M or the SUT) is modeled by a *suspension automaton* $\Delta(\mathrm{L})$, an IOLTS obtained from L by adding self-loops labeled by a special output action δ on the quiescent states. The SUT conforms to the model M modulo the **ioco** relation [40] if after executing each trace of $\Delta(\mathrm{M})$, the suspension automaton $\Delta(\mathrm{SUT})$ exhibits only those outputs and quiescences that are allowed by the model. Since two sequences having the same observable actions (including quiescence) cannot be distinguished, the suspension automaton $\Delta(\mathrm{M})$ must be determinized before generating tests.

The test generation technique of TGV is based upon *test purposes*, which allow one to guide the selection of test cases. A test purpose for a model $\mathrm{M} = (Q^{\mathrm{M}}, A^{\mathrm{M}}, T^{\mathrm{M}}, q_0^{\mathrm{M}})$ is a deterministic and complete (i.e., in each state all actions are accepted) IOLTS $\mathrm{TP} = (Q^{\mathrm{TP}}, A^{\mathrm{TP}}, T^{\mathrm{TP}}, q_0^{\mathrm{TP}})$, with the same actions as the model $A^{\mathrm{TP}} = A^{\mathrm{M}}$. TP is equipped with two sets of trap states $\mathit{Accept}^{\mathrm{TP}}$ and $\mathit{Refuse}^{\mathrm{TP}}$, which are used to select desired behaviors and to cut the exploration of M, respectively. In the TP shown on Figure 1(b), the desired behavior consists of an action !y followed by !z and is specified by the accepting state q_3; notice that the occurrence of an action !z before a !y is forbidden by the refusal state q_2. In a TP, a special transition of the form $q \xrightarrow{*} q'$ is an abbreviation for the complement set of all other outgoing transitions of q. These $*$-transitions facilitate the definition of a test purpose (which has to be a complete IOLTS) by avoiding the need to explicitly enumerate all possible actions for all states. Test purposes are used to mark the accepting and refusal states in the IOLTS of the model M. In TGV, this annotation is computed by a synchronous product [25, Definition 8] SP = M × TP. Notice that SP preserves all behaviors of the model M because TP is complete and the synchronous product takes into account the special $*$-transitions. When computing SP, TGV implicitly adds a self-looping $*$-transition to each state of the TP with an incomplete set of outgoing transitions. To keep only the visible behaviors and quiescence, SP is suspended and determinized, leading to $\mathrm{SP}_{vis} = det(\Delta(\mathrm{SP}))$. Figure 1(c) shows an excerpt of SP_{vis} limited to the first accepting and refusal states reachable from $q_0^{\mathrm{SP}_{vis}}$.

A *test case* is an IOLTS $\mathrm{TC} = (Q^{\mathrm{TC}}, A^{\mathrm{TC}}, T^{\mathrm{TC}}, q_0^{\mathrm{TC}})$ equipped with three sets of trap states $\mathbf{Pass} \cup \mathbf{Fail} \cup \mathbf{Inconc} \subseteq Q^{\mathrm{TC}}$ denoting verdicts. The actions of TC are partitioned into $A_{\mathrm{I}}^{\mathrm{TC}}$ and $A_{\mathrm{O}}^{\mathrm{TC}}$ subsets[1]. A test case TC must be *controllable*, meaning that in every state, no choice is allowed between two inputs or an input and an output (i.e., the tester must either inject a single input to the SUT, or accept all the outputs of the SUT). Intuitively, a TC denotes a set of traces containing visible actions and quiescence that should be executable by the SUT to assess its conformance with the model M and a test purpose TP. From every state of the TC, a verdict must be reachable: **Pass** indicates that TP has been fulfilled, **Fail** indicates that SUT does not conform to M, and **Inconc** indicates that correct behavior has been observed but TP cannot be fulfilled. An example of TC (dark gray states) is shown on Figure 1(c). Pass verdicts correspond to accepting states (e.g., q_{11}). Inconclusive verdicts correspond either to refusal states (e.g., q_4 or q_6) or to states from which no accepting state is reachable (e.g., state q_{10}). Fail verdicts, not displayed on the figure, are reached from every state when the SUT exhibits an output action (or a quiescence) not specified in the TC (e.g., an action !z or a quiescence in state q_1).

In general, there are several test cases that can be generated from a given model and test purpose. The union of these test cases forms the Complete Test Graph (CTG), which is an IOLTS having the same characteristics as a TC except for controllability. Figure 1(c) shows the CTG (light and dark gray states) corresponding to M and TP, which is not controllable (e.g., in state q_5 the two input actions ?a and ?b are possible). Formally, a CTG is the subgraph of SP_{vis} induced by the states L2A (*lead to accept*) from which an accepting state is reachable, decorated with pass and inconclusive verdicts. A controllable TC exists iff the CTG is not empty, i.e., $q_0^{\mathrm{SP}_{vis}} \in \mathrm{L2A}$ [25].

[1] In TGV [25], the actions of test cases are symmetric w.r.t. those of the model M and the SUT, i.e., $A_{\mathrm{O}}^{\mathrm{TC}} \subseteq A_{\mathrm{I}}^{M}$ (TC emits only inputs of M) and $A_{\mathrm{I}}^{\mathrm{TC}} \subseteq A_{\mathrm{O}}^{\mathrm{SUT}} \cup \{\delta\}$ (TC captures outputs and quiescences of SUT). To avoid confusion, we consider here that inputs and outputs of TC are the same as those of M and SUT.

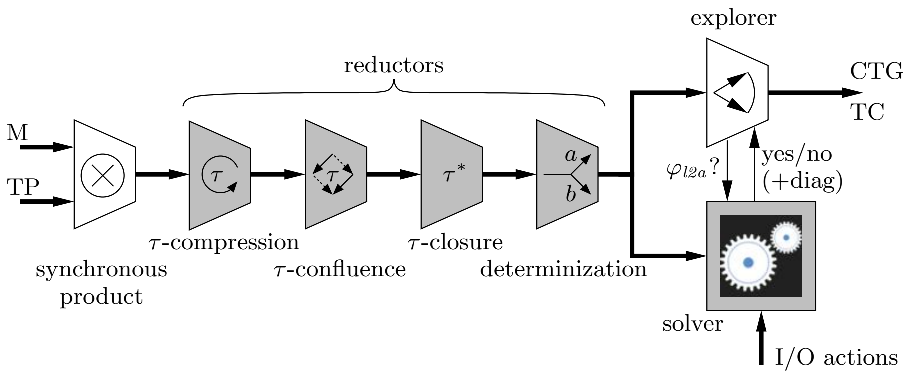


Fig. 2. Architecture of TESTOR

The execution of a TC against the SUT corresponds to a parallel composition TC || SUT with synchronization on common observable actions, verdicts being determined by the trap states reached by a maximal trace of TC || SUT, i.e., a trace leading to a verdict state. Quiescent livelock states (infinite sequences of internal actions in the SUT) are detected using timers, and lead to inconclusive verdicts. A TC may have cycles, in which case global timers are required to prevent infinite test executions.

3 TESTOR

We present the architecture and implementation of TESTOR, its on-the-fly algorithm for test-case extraction, and show several ways of specifying test purposes.

3.1 Architecture

TESTOR takes as input a formal model (M), a test purpose (TP), and a predicate specifying the input/output actions of M. Depending on the chosen options, it produces as output either a complete test graph (CTG), or a test case (TC) extracted on the fly. TESTOR has a modular component-based architecture consisting of several on-the-fly IOLTS transformation components, interconnected according to the architecture shown on Figure 2. The boxes represent transformation components and the arrows between them denote the implicit representations (*post* functions) of IOLTSs.

The first component produces the synchronous product (SP) between the model M and the test purpose TP. Following the conventions of TGV [25], the synchronous product supports $*$-transitions and implements the implicit addition of self-looping $*$-transitions. The next four reduction components progressively transform SP into $\mathrm{SP}_{vis} = det(\Delta(\mathrm{SP}))$ as follows: (i) τ-compression produces the suspension automaton $\Delta(\mathrm{SP})$ by squeezing the strongly connected components of τ-transitions and replacing them with δ-loops representing quiescence; (ii) τ-confluence eliminates redundant interleavings by giving priority to confluent τ-transitions, i.e., whose neighbor transitions (going out from the same source state) do not bring new observational behavior; (iii) τ-closure computes the transitive reflexive closure on τ-transitions; (iv) the resulting τ-free IOLTS is determinized by applying the classical subset construction. The reduction by τ-compression is necessary for τ-confluence (which operates on IOLTSs without τ-cycles) and is also useful as a preprocessing step for τ-closure (whose algorithm is simpler in the absence of τ-cycles). Although τ-confluence is optional, it may reduce drastically the size of the IOLTS prior to τ-closure, therefore acting as an accelerator for the whole test selection procedure when SP contains large diamonds of τ-transitions produced by the interleavings of independent actions [31]. The first three reductions [31] are applied only if TESTOR detects the presence of τ-transitions in SP.

The determinization produces as output the post function of the IOLTS SP_{vis}, whose states correspond to sets of states of the τ-free IOLTS produced by τ-closure. SP_{vis} is processed by the explorer component, which builds the CTG or the TC by computing the corresponding subgraph whose states are contained in L2A. The reachability of accepting states is determined on the fly by evaluating the PDL [14] formula $\varphi_{l2a} = \langle \mathsf{true}^* \rangle \mathit{accept}$ on the states visited by the explorer, where the atomic proposition *accept* denotes the accepting states. This check is done by translating the verification problem into a Boolean equation system (BES) and solving it on the fly using a BES solver component [32]. The synchronous product and the explorer are the only components newly developed, all the other ones (represented in gray on Figure 2) being already available in the libraries of the OPEN/CAESAR [15] environment of CADP.

3.2 On-the-Fly Test Selection Algorithm

We describe below the algorithm used by the explorer component to extract the CTG or a (controllable) TC from the SP_{vis} IOLTS on the fly.

Basically, the CTG is the subgraph of SP_{vis} containing all states in L2A, extended with some states denoting verdicts. The accepting states (which are by definition part of L2A) correspond to pass verdicts. For every state $q \in \mathrm{L2A}$, the output transitions $q \xrightarrow{!a} q'$ with $q' \notin \mathrm{L2A}$ lead to inconclusive verdicts, and the output transitions other than those contained in SP_{vis} lead to fail verdicts. To compute the CTG, the explorer component performs a forward traversal of SP_{vis} and keeps the states $q \in \mathrm{L2A}$, which satisfy the formula φ_{l2a}. The check $q \models \varphi_{l2a}$ is done by solving the variable X_q of the minimal fixed point BES $\{X_q = (q \models \mathit{accept}) \vee \bigvee_{q \xrightarrow{b} q'} X_{q'}\}$ denoting the interpretation of φ_{l2a} on SP_{vis}. The resolution is carried out on the fly using the algorithm for disjunctive BESs proposed in [32]. If the CTG is not empty (i.e., $q_0^{\mathrm{SP}_{vis}} \models \varphi_{l2a}$), then it contains at least one controllable TC [25].

The extraction of a TC uses a similar forward traversal as for generating the CTG, extended to ensure controllability, i.e., every state q of TC either has only one outgoing input transition $q \xrightarrow{?a} q'$ with $q' \in$ L2A, or has all output transitions $q \xrightarrow{!a} q''$ of SP_{vis} with $q'' \in$ L2A. The essential ingredient for selecting the input transitions on the fly is the diagnostic generation for BESs [30], which provides, in addition to the Boolean value of a variable, also the minimal fragment (w.r.t. inclusion) of the BES illustrating the value of that variable. For a variable X_q evaluated to true in the disjunctive BES underlying φ_{l2a}, the diagnostic (witness) is a sequence $X_q \xrightarrow{b_1} X_{q_1} \xrightarrow{b_2} \cdots \xrightarrow{b_k} X_{q_k}$ where $q_k \models accept$. This induces a sequence of transitions $q \xrightarrow{b_1} q_1 \xrightarrow{b_2} \cdots \xrightarrow{b_k} q_k$ in SP_{vis} leading to an accepting state. Since all states $q, q_1, ..., q_k$ also belong to L2A, this diagnostic sequence is naturally part of the TC under construction.

More precisely, the TC extraction algorithm works as follows. If $q_0^{\mathrm{SP}_{vis}} \models \varphi_{l2a}$, the diagnostic sequence for $q_0^{\mathrm{SP}_{vis}}$ is inserted in the TC (otherwise the algorithm stops because the CTG is empty). For the TC illustrated on Figure 1(c), this first diagnostic sequence is $q_0 \xrightarrow{\mathrm{?a}} q_1 \xrightarrow{\mathrm{!y}} q_5 \xrightarrow{\mathrm{?b}} q_9 \xrightarrow{\mathrm{!z}} q_{11}$. Then, the main loop consists in choosing an unexplored transition of the TC and processing it.

- If it is an input transition $q \xrightarrow{?a} q'$, nothing is done, since the target state $q' \in$ L2A by construction. Furthermore, the presence of this transition in the TC makes its source state q controllable. This is the case, e.g., for the transition $q_0 \xrightarrow{\mathrm{?a}} q_1$ in the TC shown on Figure 1(c).
- If it is an output transition $q \xrightarrow{!a} q'$, each of its neighboring output transitions $q \xrightarrow{!a'} q''$ is examined in turn. If the target state $q'' \notin$ L2A, the transition is inserted in TC and q'' is marked with an inconclusive verdict. This is the case, e.g., for the transition $q_1 \xrightarrow{\mathrm{!x}} q_4$ in the TC on Figure 1(c). If $q'' \in$ L2A, the transition in inserted in the TC, together with the diagnostic sequence produced for q''. This is the case, e.g., for the transition $q_9 \xrightarrow{\mathrm{!y}} q_5$ in the TC on Figure 1(c).

The insertion of a diagnostic sequence in the TC stops when it meets a state q that already belongs to the TC, since by construction the TC already contains a sequence starting at q and leading to an accepting state. This is the case, e.g., for the diagnostic sequence starting at state q_5 in the TC on Figure 1(c). In this way, the TC is built progressively by inserting the diagnostic sequences produced for each of the encountered states in L2A.

During the forward traversal of SP_{vis}, the explorer component continuously interacts with the BES solver, which in turn triggers other forward explorations of SP_{vis} to evaluate φ_{l2a}. The repeated invocations of the solver have a cumulated linear complexity in the size of the BES (and hence, the size of SP_{vis}), because the BES solver keeps its context in memory and does not recompute already solved Boolean variables [32].

3.3 Implementation

TESTOR is built upon the generic libraries of the OPEN/CAESAR [15] environment, in particular the on-the-fly reductions by τ-compression, τ-confluence and τ-closure [31], and the on-the-fly BES resolution [32]. The tool (available at `http://convecs.inria.fr/software/testor`) consists of 5022 lines of C and 1106 lines of shell script.

3.4 Examples of Different Ways to Express a Test Purpose

Consider an asynchronous implementation of the DES (Data Encryption Standard) [37]. In a nutshell, the DES is a block-cipher taking three inputs: a Boolean indicating whether encryption or decryption is requested, a 64-bit key, and a 64-bit block of data. For each triple of inputs, the DES computes the 64-bit (de)crypted data, performing sixteen iterations of the same cipher function, each iteration with a different 48-bit subkey extracted from the 64-bit key.

A natural TP for the DES is to search for a sequence corresponding to the encryption of a single data block, for instance `0x0123456789abcdef` with key `0x133457799bbcdff1`, the expected result of which is `0x85e813540f0ab405`. Using the LNT language [17,8], one would be tempted to write this TP as the process `PURPOSE1`, simply containing the desired sequence of three inputs (on gates `CRYPT`, `KEY`, and `DATA`) followed by an output (on gate `OUTPUT`):

```
process PURPOSE1 [CRYPT: CB, KEY, DATA, OUTPUT: C64, T_ACCEPT: none] is
   CRYPT (true); -- input
   KEY (C_13345779_9bbcdff1); -- input
   DATA (C_01234567_89abcdef); -- input
   OUTPUT (C_85e81354_0f0ab405); -- output
   loop T_ACCEPT end loop
end process
```

Following the conventions of TGV, we mark accepting (respectively, refusal) states by a self-loop labeled with `T_ACCEPT` (respectively, `T_REFUSE`).

However, `PURPOSE1` is not complete: e.g., initially only one action out of the possible set {`CRYPT (true)`, `CRYPT (false)`, `KEY (C_13345779_9bbcdff1)`, ...} is specified. Thus, when computing the synchronous product with the model, `PURPOSE1` is implicitly completed by self-loops labeled with "`*`" (as in the TP shown on Figure 1(b)), yielding a significantly more complex TC than expected. For instance, the implicit `*`-transition in the initial state allows the tester to perform the sequence "`CRYPT (false); CRYPT (true)`" rather than the expected first action "`CRYPT (true)`". To force the generation of a TC corresponding to the simple sequence, it is necessary to explicitly complete the TP with transitions to refusal states, as shown by the LNT process `PURPOSE2`, where gate `OTHERWISE` stands for the special label "`*`":

```
process PURPOSE2 [CRYPT: CB, KEY, DATA, OUTPUT: C64, SUBKEY: C48,
                  T_ACCEPT, T_REFUSE, OTHERWISE: none] is
   select -- refuse any rendezvous but "CRYPT (TRUE)"
      CRYPT (true)
   [] OTHERWISE; loop T_REFUSE end loop
   end select;
   select -- refuse any rendezvous but "KEY (C_13345779_9BBCDFF1)"
```

```
      KEY (C_13345779_9BBCDFF1)
   [] OTHERWISE; loop T_REFUSE end loop
   end select;
   loop L in
      select -- refuse any rendezvous but on gates DATA and SUBKEY
         DATA (C_01234567_89ABCDEF); break L
      [] SUBKEY (?any BIT48)
      [] OTHERWISE; loop T_REFUSE end loop
      end select
   end loop;
   loop -- refuse any rendezvous but on gates OUTPUT and SUBKEY
      select -- test target is reached by a rendezvous on OUTPUT
         OUTPUT (C_85E81354_0F0AB405); loop T_ACCEPT end loop
      [] SUBKEY (?any BIT48)
      [] OTHERWISE; loop T_REFUSE end loop
      end select
   end loop
end process
```

Instead of using the dedicated synchronous product, it is also possible to take advantage of the multiway rendezvous [23,18] to compositionally annotate the model, relying on the LNT operational semantics [8, Appendix B] to cut undesired branches. For instance, the same effect as the synchronous product with `PURPOSE2` can be obtained by skipping the left-most component "synchronous product" of Figure 2, i.e., feeding the τ-reduction steps with the IOLTS described by the following LNT parallel composition:

```
par CRYPT, KEY, DATA, OUTPUT in
   DES [CRYPT, KEY, DATA, OUTPUT, SUBKEY]
|| PURPOSE1 [CRYPT, KEY, DATA, OUTPUT, T_ACCEPT]
end par
```

This approach based on the multiway rendezvous even supports data handling. For instance, to observe the data (variable `D`), key (variable `K`), and whether an encryption or decryption is requested (variable `C`), and to verify the correctness of the result (in the rendezvous "`OUTPUT (DES (C, K, D))`", `DES` denotes a function implementing the DES algorithm), one has just to replace in the above parallel composition the call to `PURPOSE1` by a call to the process `PURPOSE3`:

```
process PURPOSE3 [CRYPT: CB, KEY, DATA, OUTPUT: C64, T_ACCEPT: none] is
   var C: BOOL, D, K: BIT64 in
      CRYPT (?C);
      KEY (?K);
      DATA (?D);
      OUTPUT (DES (C, K, D));
      loop T_ACCEPT end loop
   end var
end process
```

4 Experimental Evaluation

TESTOR follows TGV's implementation of the **ioco**-based testing theory [39,41], using the same IOLTS processing steps, adding only the τ-confluence reduction. For each step, TESTOR uses components developed, tested, and used in other tools for more than a decade. In this section, we focus on performance aspects and we compare TESTOR to TGV. For this purpose, we conducted several experiments with models and test purposes, both automatically generated and drawn from academic examples and realistic case studies.

For assessing the correctness of TESTOR, we checked that each TC is included in the CTG, and we compared the TCs and CTGs generated by TESTOR to those generated by TGV. The latter comparison required several additional steps, automated using shell scripts and a dedicated tool (about 300 lines of C code). First, we generated the LTS of each TP, applying appropriate renamings, because TGV expects the TP to be an explicit LTS, with accepting (resp. refusing) states marked by a self-looping transition labeled with `ACCEPT` (resp. `REFUSE`), and with the label "`*`". Then, we modified the TC and CTG generated by TESTOR so that each label includes the information whether the label is an input or output, and which verdict state (if any) is reached by the corresponding transition. Using this approach, we found that the CTGs generated by both tools were strongly bisimilar. The same does not hold for all the TCs, because the tools may ensure controllability in different ways, leading to non-bisimilar, but correct TCs.

For each pair of model and TP, we measured the runtime and peak memory usage of computing a TC or CTG (using TESTOR and TGV), excluding the fixed cost of compiling the LNT code (model and TP) and generating the executable. The experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`). Concretely, we used the `petitprince` cluster located in Luxembourg, consisting of sixteen machines, each equipped with 2 Intel Xeon E5-2630L CPUs, 32GB RAM, and running 64-bit Debian GNU/Linux 8 and CADP 2017-i. Each measurement corresponds to the average of ten executions.

4.1 Test Purposes taken from Case Studies

Table 1 summarizes the results for some selected examples. The first two have been kindly provided by Alexander Graf-Brill, and correspond to initial versions of TPs for his EnergyBus model [20]; both aim at exhibiting a particular boot sequence, the second one using `REFUSE` transitions. The next four examples have been used by STMicroelectronics to verify a cache-coherence protocol [27]. The last three correspond to the three TPs presented in Section 3.4 and check the correctness of a simplified[2] version of the asynchronous implementation of the

[2] The S-boxes are executed sequentially rather than in parallel and the gate `SUBKEY` is left visible to separate the iterations of the DES algorithm and thus significantly

Table 1. Run-time performance for selected examples

example	TESTOR test case time	TESTOR test case mem.	TESTOR CTG time	TESTOR CTG mem.	TGV test case time	TGV test case mem.	TGV CTG time	TGV CTG mem.
EnergyBus	3	81	182	181	2	137	52	858
EnergyBus (with REFUSE)	1	67	1	66	0	66	0	43
ACE UniqueDirty	45	121	346	451	75	159	3047	643
ACE SharedDirty	384	510	342	529	3821	746	3920	746
ACE SharedClean	298	415	325	523	2820	628	3474	663
ACE Data Inconsistency	24	116	580	711	24	142	6701	894
DES (`PURPOSE1`)	22109	300	>1week		>43GB		>220GB	
DES (`PURPOSE2`)	27344	332	27	86	24	6177	24	6176
DES (`PURPOSE3`)	2	74	4	100	not applicable			

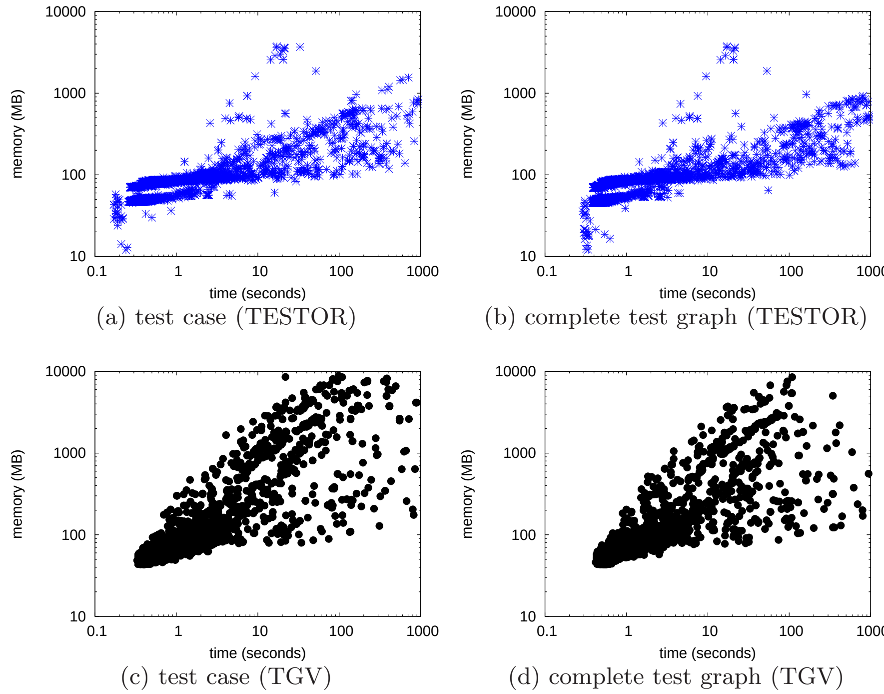
Execution time is given in seconds and memory usage in MB.

DES (Data Encryption Standard) [37]. These examples cover a large spectrum of characteristics: from no τ-transitions (ACE) to huge confluent τ-components (DES), from few visible transitions (DES) to many outgoing visible transitions (EnergyBus), and a test selection more or less guided via refusal states.

We observe that TESTOR requires less memory than TGV for all examples, but most significantly for the DES. However, although TESTOR is several orders of magnitude slower than TGV for the DES when using the synchronous product (TPs `PURPOSE1` and `PURPOSE2`), TESTOR requires only two seconds to generate a TC or CTG when using an LNT parallel composition with the TP with data handling `PURPOSE3`. This is because the LNT parallel composition, handled by the LNT compiler, enables more aggressive optimizations. Thus, using LNT parallel composition to annotate the model's accepting and refusal states is not only more convenient (thanks to the multiway rendezvous) and data aware, but also much more efficient — it is even possible to generate a TC for the original DES model (167 million states, 1.5 billion transitions) in less than 40 minutes.

For the ACE examples, TESTOR is both faster and requires less memory than TGV. This is partly due to an optimization of TESTOR, which deactivates the various reductions of τ-transitions. For a fair comparison, we also run experiments forcing the execution of these reductions. For the extraction of a TC, this increases the execution time by a factor of two and the memory requirements by a factor of three. For the computation of a CTG, this increases the memory requirements by a factor of one and a half, without modifying the execution time significantly.

reduce the size of τ-components. For the extraction of TC for `PURPOSE2` from the full version of the DES, TESTOR would run for several weeks and TGV would require more than 700 GB of RAM.



(a) test case (TESTOR)
(b) complete test graph (TESTOR)
(c) test case (TGV)
(d) complete test graph (TGV)

Fig. 3. Compared performance of TESTOR and TGV

4.2 Automatically Generated Test Purposes

To evaluate the performance, we used a collection of 9791 LTSs with up to 50 million transitions, taken from the non-regression test-base for CADP. For each LTS M of the collection, we automatically generated two TPs: one to test the reachability of an action and another to test the presence of an execution sequence. For the former TP, we sorted the actions of the LTS alphabetically, and checked the reachability of the first action, considering the second half of the action set as inputs. For the latter TP, we used the EXECUTOR tool[3] to extract a sequence of up to 1000 visible actions, which we transformed into a TP, considering all actions whose ranking is an odd number as inputs. Technically, this transformation consists in adding to each state of the sequence a self-loop labeled with τ and a $*$-transition to a refusal state.

From the generated pairs (M, TP) we eliminated those for which the automatic generation of a TP failed (for instance, due to special actions that would require particular treatment) and those for which the computation of a TC or CTG took too much time or required too much memory by either TESTOR or TGV. This led to a collection of 13,142 pairs (M, TP) for which both tools

[3] `http://cadp.inria.fr/man/executor.html`

could extract a TC. For 12,654 of them, both tools also could compute the CTG. Figure 3 displays the results for each example, using logarithmic scales for both execution time and memory requirements, to make the differences for small values more visible.

As for the case studies, we observe that TESTOR and TGV choose different tradeoffs between computation time and memory requirements. On average, TESTOR requires 0.3 times less memory and runs 1.3 (respectively 0.5) times faster to compute a TC (respectively the CTG). When considering only the 1005 pairs with more than 500,000 transitions in the LTS, the average numbers show a larger difference. On average for these larger examples, to compute a CTG, TESTOR requires 1.4 times less memory, but runs 3.5 times longer; to compute a TC, TESTOR requires 2.7 times less memory and runs 0.7 times faster.

Also, while both tools required the exclusion of examples due to excessive runtime, we excluded several examples due to insufficient memory for TGV, but not for TESTOR. Given that TCs are usually much smaller than CTGs, the on-the-fly extraction of a TC by TESTOR is generally faster and consumes less memory than the generation of the CTG. We also observed that the CTGs produced by TESTOR are sometimes smaller than (although strongly bisimilar to) those produced by TGV.

While trying to understand these results in more detail, we found examples where each tool is one or two magnitudes faster or memory-efficient than the other. Indeed, the benefits of the different reductions applied in the tools depend heavily on the characteristics of the example, most notably the sizes of the various subgraphs explored (τ-components, L2A). For instance, when the model M does not contain any τ-transition, there is no point in applying the reductions (τ-compression, τ-confluence, and τ-closure).

The modular architecture of TESTOR enabled us to easily experiment with variants of the algorithm used for solving the BES underlying φ_{l2a}. By default, when extracting a TC on the fly, we use the depth-first search (DFS) algorithm, which for disjunctive BESs stores only variables and not their dependencies (and hence only the states, and not the transitions of the model). Using the breadth-first search (BFS) algorithm of the solver produces smaller TCs, because it generates the shortest diagnostic sequences for states in L2A. However, this comes at the price of an increased execution time and memory consumption, a known phenomenon regarding BFS versus DFS algorithms [32]. Thus, one can choose between BFS or DFS resolution if the size of the TC extracted on the fly is judged more important or not than the resources required to compute it.

5 Related Work

Although model-based conformance testing has been intensively studied, there are only a few tools that use variants of the **ioco** conformance relation and that are still actively developed [4]. Other model-based tools for combinatorial and statistical testing, or white box testing are described in [43]. In the following, we compare TESTOR to the most closely related tools.

TorX [42] and JTorX [2] are online test generation tools, equipped with a set of adapters to connect the tester to the SUT. The latest versions support test purposes (TPs), but they are used differently than in TESTOR. Indeed, JTorX yields a two-dimensional verdict [3]: one dimension is the **ioco** correctness verdict (pass or fail), and the other dimension is an indication whether the test objective has been reached. This contrasts with TESTOR, which generates test cases (TCs) ensuring by construction that the execution stays inside the lead to accept states (L2A), and stopping the test execution as soon as possible with a verdict: **fail** if non-conformance has been detected, **pass** if an accepting state has been reached, or **inconclusive** if leaving L2A is unavoidable.

Uppaal is a toolbox for the analysis of timed systems, modeled as timed automata extended with data. Three test generation tools exist for Uppaal timed automata. Uppaal-Tron [28] is an online test generation tool, taking as input a specification and an environment model, used to constrain the test generation. Uppaal-Tron is also equipped with a set of adapters to derive and execute the generated tests on the SUT. Contrary to TESTOR, the TCs generated from Uppaal-Tron can be irrelevant, because the generation is not guided by TPs. Uppaal-Cover [22] generates offline a comprehensive test suite from a deterministic Uppaal model and coverage criteria specified by observer automata. Uppaal-Cover attempts to build small test suite satisfying the coverage criteria, by selecting those TCs satisfying the largest parts of the coverage criteria. In contrast to TESTOR and Uppaal-Tron, Uppaal-Cover generates offline tests. Offline generation does not face the state-space explosion, but also limits the expressiveness of the specification language (e.g, nondeterministic models are not allowed). Uppaal-Yggdrasil [26] generates offline test suites for deterministic Uppaal models, using a three-step strategy to achieve good coverage: (i) a set of reachability formulas, (ii) random execution, and (iii) structural coverage of the transitions in the model. The guidance of the test generation by a temporal logic formula is similar to the use of a TP. However, the TPs supported by TESTOR (and TGV) can express more complex properties than reachability, and enable one to control the explored part of the model (using refusal states).

On-the-fly test generation tools also exist for the synchronous dataflow language Lustre [21], e.g., Lutess [12], Lurette [24], and Gatel [29]. Contrary to TESTOR, these tools do not check the **ioco** relation, but randomly select TCs, satisfying constraints of an environment description and an oracle.

In IOLTS, actions are monolithic, which does not fit for realistic models that involve data handling. STG (Symbolic Test Generator) [11] breaks the monolithic structure of actions, enabling access to the data values, and generates tests on the fly, handling data values symbolically. This enables more user-friendly TPs and more abstract TCs, because not all possible values have to be enumerated. However, the complexity of symbolic computation is not negligible in practice. When using the LNT parallel composition, TESTOR can handle data (see example in Section 3.4) without the cost of symbolic computation, but still has to enumerate data explicitly when generating the TC. T-Uppaal [34] uses symbolic reachability analysis to generate tests on the fly and then simultaneously executes them on the SUT. The complexity of symbolic algorithms turns out to be expensive for online testing.

When executing a generated TC against a SUT, it is necessary to refine it to take into account the asynchronous communication between the SUT and the tester. Actually, the SUT accepts every input at any time, whereas the TC is deterministic, i.e., there is no choice between an input and an output. An approach for connecting a TC (randomly selected) and an asynchronous SUT was defined in [44]. A similar approach using TPs to guide the test generation was proposed in [5] and subsequently extended to timed automata [6]. Recently, this kind of connection was automated by the MOTEST tool [19].

6 Conclusion

We presented TESTOR, a new tool for on-the-fly conformance test case generation for asynchronous concurrent systems. Like the existing tool TGV, TESTOR was developed on top of the CADP toolbox [16] and brings several enhancements: online testing by generating (controllable) test cases completely on the fly; a more versatile description of test purposes using the LNT language; and a modular architecture involving generic graph manipulation components from the OPEN/CAESAR environment [15]. The modularity of TESTOR simplifies maintenance and fine-tuning of graph manipulation components, e.g., by adding or removing on-the-fly reductions, or by replacing the synchronous product. Besides the ability to perform online testing, the on-the-fly test selection algorithm sometimes makes possible the extraction of test cases even when the generation of the complete test graph (CTG) is infeasible.

The experiments we carried out on ten-thousands of benchmark examples and three industrial case studies show that TESTOR consumes less memory than TGV, which in turn is sometimes faster, for generating CTGs. We plan to experiment with state space caching techniques [33] and with other on-the-fly reductions to accelerate CTG generation in TESTOR. We also plan to investigate how to facilitate the description of test purposes, by deriving them from the action-based, branching-time temporal properties of the model (following the results of [13] in the state-based, linear-time setting) or by synthesizing them according to behavioral coverage criteria.

Acknowledgements

We are grateful to Alexander Graf-Brill and Holger Hermanns for providing us with the model and test purposes of their EnergyBus case study. We also thank Hubert Garavel for helpful remarks about the paper. This work was supported by the Région Auvergne-Rhône-Alpes within the program ARC 6.

References

1. B. K. Aichernig, F. Lorber, and S. Tiran. Integrating Model-Based Testing and Analysis Tools via Test Case Exchange. In: TASE 2012, pp. 119–126 (2012).

2. A. Belinfante. JTorX: A Tool for On-line Model-driven Test Derivation and Execution. In: TACAS 2010, LNCS, vol. 6015, pp. 266–270, Springer (2010).
3. A. Belinfante. *JTorX: Exploring Model-Based Testing*. PhD thesis, University of Twente (2014).
4. A. Belinfante, L. Frantzen, and C. Schallhart. 14 Tools for Test Case Generation. In: [7], pp. 391–438.
5. P. Bhateja. A TGV-like Approach for Asynchronous Testing. In: ISEC 2014, pp. 13:1–13:6, ACM (2014).
6. P. Bhateja. Asynchronous testing of real-time systems. In: SCSS 2017, *EPiC Series in Computing*, vol. 45, pages 42–48, EasyChair (2017).
7. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Model-Based Testing of Reactive Systems, LNCS, vol. 3472. Springer (2005).
8. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, C. McKinty, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LNT to LOTOS Translator (Version 6.7). INRIA (2017).
9. V. Chimisliu and F. Wotawa. Improving Test Case Generation from UML Statecharts by Using Control, Data and Communication Dependencies. In: QSIC 13, pp. 125–134 (2013).
10. V. Chimisliu and F. Wotawa. Using Dependency Relations to Improve Test Case Generation from UML Statecharts. In: COMPSAC, pp. 71–76, IEEE (2013).
11. D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: A symbolic test generation tool. In: TACAS 2002, LNCS, vol. 2280, pp. 470–475. Springer (2002).
12. L. du Bousquet, F. Ouabdesselam, J. Richier, and N. Zuanon. Lutess: A Specification-Driven Testing Environment for Synchronous Software. In: ICSE 1999, pp. 267–276, ACM (1999).
13. Y. Falcone, J.-C. Fernandez, T. Jéron, H. Marchand, and L. Mounier. More testable properties. STTT 14(4), 407–437 (2012).
14. M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2), 194–211 (1979).
15. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In: TACAS 1998, LNCS, vol. 1384, pp. 68–84 (1998).
16. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. STTT 15(2), 89–107 (2013).
17. H. Garavel, F. Lang, and W. Serwe. From LOTOS to LNT. In: ModelEd, TestEd, TrustEd, LNCS, vol. 10500, pp. 3–26, Springer (2017).
18. H. Garavel and W. Serwe. The Unheralded Value of the Multiway Rendezvous: Illustration with the Production Cell Benchmark. In: MARS 2017, EPTCS 244, 230–270 (2017).
19. A. Graf-Brill and H. Hermanns. Model-Based Testing for Asynchronous Systems. In: FMICS-AVoCS 2017, LNCS, vol. 10471, pp. 66–82, Springer (2017).
20. A. Graf-Brill, H. Hermanns, and H. Garavel. A Model-based Certification Framework for the EnergyBus Standard. In: FORTE 2015, LNCS, vol. 8461, pp. 84–99, Springer (2014).
21. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. Proceedings of the IEEE 79(9), 1305–1320 (1991).
22. A. Hessel and P. Pettersson. Model-Based Testing of a WAP Gateway: An Industrial Case-Study. In: FMICS/PDMC 2006, LNCS, vol. 4346, pp. 116–131 (2006).
23. C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8), 666–677 (1978).
24. E. Jahier, P. Raymond, and P. Baufreton. Case studies with Lurette V2. STTT 8(6), 517–530 (2006).

25. C. Jard and T. Jéron. Tgv: Theory, principles and algorithms – a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. STTT 7(4), 297–315 (2005).
26. J. H. Kim, K. G. Larsen, B. Nielsen, M. Mikucionis, and P. Olsen. Formal Analysis and Testing of Real-Time Automotive Systems Using UPPAAL Tools. In: FMICS 2015, LNCS, vol. 9128, pp. 47–61, Springer (2015).
27. A. Kriouile and W. Serwe. Using a Formal Model to Improve Verification of a Cache-Coherent System-on-Chip. In: TACAS 2015, LNCS, vol. 9035, pp. 708–722, Springer (2015).
28. K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing Real-time Embedded Software Using UPPAAL-TRON: An Industrial Case Study. In: EMSOFT 2005, pp. 299–306, ACM (2005).
29. B. Marre and A. Arnould. Test Sequences Generation from LUSTRE Descriptions: GATeL. In: ASE 2000, p. 229, IEEE (2000).
30. R. Mateescu. Efficient diagnostic generation for boolean equation systems. In: TACAS 2000, LNCS, vol. 1785, pp. 251–265, Springer (2000).
31. R. Mateescu. On-the-fly state space reductions for weak equivalences. In FMICS 2015, pp. 80–89, ACM (2005).
32. R. Mateescu. Caesar_solve: A generic library for on-the-fly resolution of alternation-free boolean equation systems. STTT 8(1), 37–56 (2006).
33. R. Mateescu and A. Wijs. Hierarchical Adaptive State Space Caching based on Level Sampling. In: TACAS 2009, LNCS, vol. 5505, pp. 215–229, Springer (2009).
34. M. Mikucionis, K. G. Larsen, and B. Nielsen. T-UPPAAL: online model-based testing of real-time systems. In: ASE 2004, pp. 396–397, IEEE (2004).
35. A. Mjeda. *Standard-Compliant Testing for Safety-Related Automotive Software*. PhD thesis, University of Limerick (2013).
36. Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate. Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. *IEEE Software*, 30(3), 50–57 (2013).
37. W. Serwe. Formal Specification and Verification of Fully Asynchronous Implementations of the Data Encryption Standard. In: MARS 2015, EPTCS 196, 61–147 (2015).
38. M. Sijtema, A. Belinfante, M. Stoelinga, and L. Marinelli. Experiences with Formal Engineering: Model-Based Specification, Implementation and Testing of a Software Bus at Neopost. *Science of Computer Programming*, 80(Part A), 188–209 (2014).
39. J. Tretmans. *A Formal Approach to Conformance Testing*. Twente University Press (1992).
40. J. Tretmans. Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. *Computer Networks and ISDN Systems*, 29(1), 49–79 (1996).
41. J. Tretmans. Model Based Testing with Labelled Transition Systems. In: FMT, LNCS, vol. 4949, pp. 1–38, Springer (2008).
42. J. Tretmans and H. Brinksma. TorX: Automated Model-Based Testing. In: Model-Driven Software Engineering, pp. 32–43 (2003).
43. M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5), 297–312 (2012).
44. M. Weiglhofer and F. Wotawa. Asynchronous Input-Output Conformance Testing. In: COMPSAC 2009, pp. 154–159, IEEE (2009).
45. J. Zander, I. Schieferdecker, and P. J. Mosterman, editors. *Model-Based Testing for Embedded Systems*. CRC Press (2017).