Automated Transition Coverage in Behavioural Conformance Testing

Lina Marsso, Radu Mateescu, Wendelin Serwe

To cite this version:

Lina Marsso, Radu Mateescu, Wendelin Serwe. Automated Transition Coverage in Behavioural Conformance Testing. ICTSS 2020 - 32nd IFIP International Conference on Testing Software and Systems, Dec 2020, Napoli, Italy. pp.219-235, 10.1007/978-3-030-64881-7_14 . hal-03038050

HAL Id: hal-03038050

https://hal.inria.fr/hal-03038050

Submitted on 3 Dec 2020

Automated Transition Coverage in Behavioural Conformance Testing

Lina Marsso, Radu Mateescu, and Wendelin Serwe

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

Abstract. In the setting of ioco-based conformance testing with test purposes, we propose an automatic approach to generate a test plan (set of test purposes) with its associated test suite (set of test cases) covering all transitions of the IOLTS model of the system. The approach can also be applied to improve an existing test plan, by both, completing the coverage and eliminating redundancies. Implementing our approach on top of the CADP toolbox, we report on experiments with several examples of concurrent systems and discuss possible variants and heuristics to fine-tune the overall performance of the approach, as well as the quality of the computed test plan.

1 Introduction

MBT (Model-Based Testing) [41,39] encompasses the range of methods that exploit a model of the SUT (System Under Test) to automate testing. MBT enables to keep tests in close correspondence with the SUT's requirements and reduces the cost of the test activity, at the price of developing a model of the SUT. Conformance testing is a form of black-box MBT seeking to establish that an SUT behaves according to a model, which serves as an oracle. We make the common hypothesis that the behaviour of both, the model and the SUT, can be represented as an IOLTS (Input-Output Labelled Transition System) [22], which is a convenient semantic representation for high-level formal languages.

A popular conformance relation for IOLTSs is *ioco* [38], which served as basis for various testing approaches, such as on-line testing, as implemented in the JTorX tool [1], or on-the-fly test case generation guided by test purposes, as implemented in the TGV [22] and TESTOR [27] tools. The former approach has the advantage of being fully automatic (the tester executes the SUT and the model in a co-simulation manner), whereas the latter approach using test purposes allows the tester to build a *test plan*, i.e., set of test purposes at a similar abstraction level as the system requirements. In an approach based on *ioco* and test purposes, the test plan must be transformed into a *test suite*, i.e., a set of concrete, deterministic test cases to be executed on an SUT. Each test purpose directs the test case extraction and enables to handle large models by ignoring those parts of the model irrelevant to the considered test purpose. In both approaches, the tester is confronted with the questions of when to stop

[*] Institute of Engineering Univ. Grenoble Alpes

the testing process (either by terminating the co-simulation, or by devising no more test purposes), and how thoroughly the SUT has been tested. These well-known questions in the testing domain are classically addressed using *coverage criteria* [42] measuring the degree to which the internal structure of an SUT was exercised during the testing process.

Suitable coverage criteria for LTSs (and thus IOLTSs) were proposed in [37], which quantify in an increasingly stronger way how much of the LTS structure is explored during the testing process: (a) *all labels* (covering each label of the LTS, given that a same label may occur on several transitions in the LTS); (b) *all states* (covering each state of the LTS); (c) *all transitions* (covering each transition in the LTS); (d) *all proper paths* (covering each finite sequence of the LTS without repeated states); (e) *all paths* (covering each sequence of the LTS). Since criteria (d) and (e) are too costly in practice, and sometimes impossible to achieve with a finite testing process, such as criterion (e) on an LTS containing cycles, we focus on criterion (c), referred to as *transition coverage* in the sequel.

In this paper, we propose an approach to automatically generate a set of test purposes with their corresponding CTGs (Complete Test Graphs), each of which contains all necessary information to drive a (conformant) SUT towards the corresponding test purpose (if possible). This approach is iterative: in each iteration, a new test purpose is derived from a counterexample illustrating a not yet covered transition of the model. It is also possible to start from an existing, non-trivial (i.e., not empty) test plan, completing it to cover all transitions, as well as detecting redundant test purposes that do not increase the coverage. Because a CTG is not necessarily controllable (e.g., there might be a non-deterministic choice between inputs to be sent to the SUT), we further automatically extract from each CTG a deterministic test suite covering all transitions of the CTG. The union of all such generated test suites thus ensures transition coverage of the IOLTS model. We implemented the approach on top of TESTOR[1] and the CADP toolbox[2] [13], and we experimented it on several distributed systems.

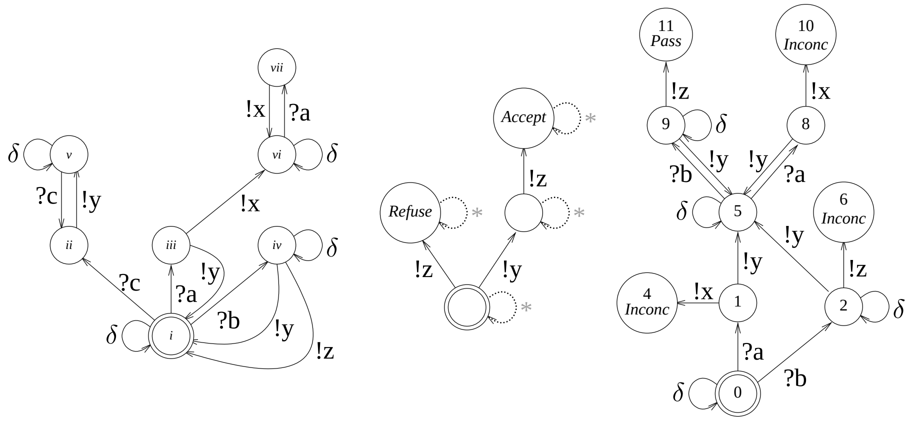
The remainder of the paper is organised as follows. Section 2 recalls the essential notions of the underlying theory. Section 3 describes the main algorithms and illustrates them on a running example. Section 4 presents an experimental evaluation of our approach, declined along several variants. Section 5 compares our approach to related work. Finally, Section 6 gives some concluding remarks.

2 Background

Conformance testing establishes that the behaviour of an SUT corresponds to the behaviour of a model (M) modulo a conformance relation. Behaviours are represented as IOLTSs (Input-Output Labelled Transition Systems) [22]. An IOLTS (S, A, T, s_0) comprises a set of states S, a set of labels (or actions) A, a transition relation $T \subseteq S \times A \times S$, and an initial state $s_0 \in S$. The label set is partitioned in $A = A_\mathrm{I} \cup A_\mathrm{O} \cup \{\tau\}$, where $A_\mathrm{I}, A_\mathrm{O}$ are the input and output labels,

[1] `http://convecs.inria.fr/software/testor`

[2] `http://cadp.inria.fr`



(a) model (suspension automaton) (b) test purpose (c) complete test graph

Fig. 1. Model, test purpose, and corresponding complete test graph [22]

and τ is the internal (invisible) label. For value-passing systems, labels are of the form $G\ v_1 \ldots v_n$, where G is the name of a gate (communication link) and $v_1, ..., v_n$ are data values; for dataless systems, labels are simply gate names. A transition $(s_1, a, s_2) \in T$ (also noted $s_1 \xrightarrow{a} s_2$) indicates a move labelled by a between states s_1 and s_2. Input (resp. output) labels are noted $?a$ (resp. $!a$). We assume the following *testing hypothesis*: an SUT can be modelled in the same way as the formal model, i.e., in our case as an IOLTS. Input (resp. output) labels of the SUT are controllable (resp. observable) by the environment (tester).

The tester observes the execution traces of the SUT and also detects *quiescent* states, i.e., deadlocks (states without successors), outputlocks (states without outgoing output labels), or livelocks (states on τ-cycles). The quiescence present in an IOLTS $M = (S^M, A^M, T^M, s_0^M)$ is modelled by a *suspension automaton* $\Delta(M)$, an IOLTS obtained from M by adding δ-loops on the quiescent states, where δ is a special output label. In the sequel, we use the same running example as [22], whose suspension automaton $\Delta(M)$ is shown in Fig. 1(a).

An SUT conforms to the model M modulo the *ioco* relation [38] if after executing each trace of $\Delta(M)$, the suspension automaton $\Delta(\text{SUT})$ exhibits only those outputs and quiescences that are allowed by M. Since two sequences with the same observable labels (and quiescence) cannot be distinguished, the suspension automaton $\Delta(M)$ must be determinised before generating tests (this is the case for $\Delta(M)$ in Fig. 1(a), which is deterministic and minimised). A noteworthy feature of *ioco* is that an SUT with *fewer* (output) transitions than the model may be considered conformant: if the model contains a choice between several outputs, an SUT is free to implement only one of them. For such an SUT, no test suite can thus achieve transition coverage of the model, because the missing transition will never be covered by executing the test cases on the SUT.

We follow here the test generation technique of the TGV tool [22], which uses *test purposes* to guide the selection of test cases. A test purpose is a deterministic and complete IOLTS TP $= (S^{\mathrm{TP}}, A^{\mathrm{TP}}, T^{\mathrm{TP}}, s_0^{\mathrm{TP}})$, with the same labels as the model $A^{\mathrm{TP}} = A^M$, and equipped with two sets of trap states $\mathit{Accept}^{\mathrm{TP}}$ and $\mathit{Refuse}^{\mathrm{TP}}$, which are used to select desired behaviours and to cut the exploration of M, respectively. The test purpose shown in Fig. 1(b) specifies a desired behaviour consisting of a label !y followed by !z and leading to an *Accept* state (the occurrence of !z before a !y is forbidden by a *Refuse* state). A transition $s \overset{*}{\rightarrow} s'$ matches every outgoing transition of s with a label other than those of its neighbours. Test purposes are used to mark the *Accept* and *Refuse* states in the IOLTS of the model M, by computing the synchronous product SP $= M \times$ TP. To keep only the visible behaviours and quiescence, SP is suspended and determinised, leading to $\mathrm{SP}_{vis} = det(\Delta(\mathrm{SP}))$.

A *test case* is an IOLTS TC $= (S^{\mathrm{TC}}, A^{\mathrm{TC}}, T^{\mathrm{TC}}, s_0^{\mathrm{TC}})$ equipped with three sets of trap states $\mathit{Pass} \cup \mathit{Fail} \cup \mathit{Inconc} \subseteq S^{\mathrm{TC}}$ denoting verdicts. The labels of TC are partitioned into $A_{\mathrm{I}}^{\mathrm{TC}}$ and $A_{\mathrm{O}}^{\mathrm{TC}}$ subsets. 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 test 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 labels 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* (inconclusive) indicates that correct behaviour has been observed but TP cannot be fulfilled. Frequently, the *Fail* state is omitted: from any state of the TC, observing an unexpected output of the SUT leads to *Fail*.

In general, several test cases can be produced from a given model and test purpose. The union of these test cases forms the CTG (Complete Test Graph) [22], which is an IOLTS similar to a TC, but possibly not controllable. Formally, a CTG is the subgraph of SP_{vis} induced by the states L2A (Lead to Accept) from which an *Accept* state is reachable, decorated with *Pass* and *Inconc* verdicts. Figure 1(c) shows the CTG corresponding to M and TP, which is not controllable (e.g., in state s_5 two inputs ?a and ?b are possible).[3] *Pass* verdicts correspond to *Accept* states (e.g., s_{11}). *Inconc* verdicts correspond either to *Refuse* states (e.g., s_6) or to states from which no *Accept* state is reachable (e.g., s_4 or s_{10}). *Fail* verdicts, not displayed on the figure, are produced for states in which the SUT can exhibit an output label or a quiescence not specified in the CTG (e.g., for label !z or quiescence in state s_1). Note that some loops of M can be unrolled in the CTG: for instance, the transitions $(s_{iii}, !\mathrm{y}, s_i)$ and $(s_{iv}, !\mathrm{y}, s_i)$ going back to the initial state of M correspond in the CTG to the transitions $(s_1, !\mathrm{y}, s_5)$ and $(s_2, !\mathrm{y}, s_5)$, respectively.

Our coverage approach relies on CTG generation and on several automata manipulation features provided by the tools of CADP [13], documented on the

[3] To avoid confusion, we follow in this paper the convention of [27] and consider that the inputs and outputs of a CTG or TC are the same as those of the model.

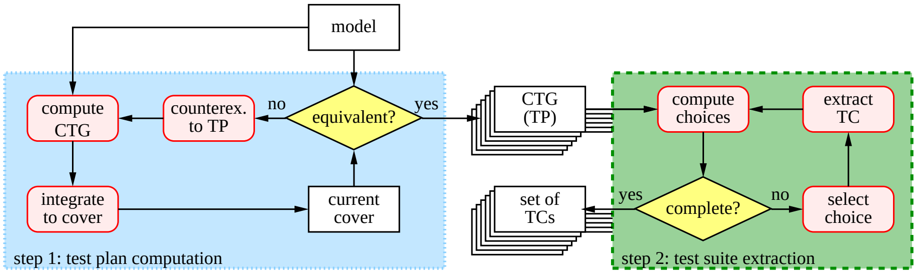


Fig. 2. Overview of the proposed approach

CADP web page. CTGs are generated using TESTOR [27], a new conformance test case generator that is also able to produce controllable TCs on the fly. The minimal deterministic automata of the CTGs and of the model M are obtained by applying weak trace reduction using the REDUCTOR tool, followed by strong bisimulation minimisation using the BCG_MIN tool. The coverage of M by a CTG is estimated by computing the semi-composition [24] between M and the CTG using the PROJECTOR tool, which produces a subgraph of M subsuming the regular language denoted by the CTG. Finally, M is compared with a CTG modulo strong bisimulation using the BISIMULATOR equivalence checker, which produces counterexamples (distinguishing sequences) used to generate new TPs during the coverage process. The whole coverage approach was automated using SVL [11] scripts.

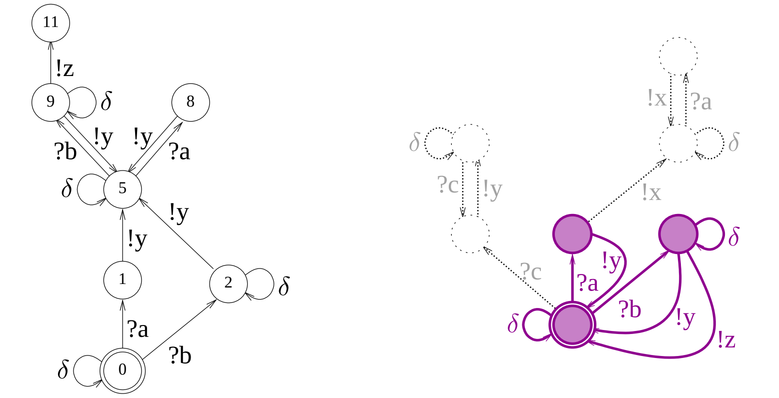
3 Computation of a Test Suite Covering All Transitions

The starting point of our approach is a *finite* IOLTS model M. To ensure meaningful measurements about coverage, we first compute the (minimised) suspension automaton $\varDelta(M)$. This preliminary step determinises M, removes all internal transitions and unreachable states, and marks quiescent states (with δ-loops). For readability, in the remainder of this section we use “model”, rather than “suspension automaton” or “$\varDelta(M)$”.

Our approach, shown in Fig. 2, consists of two steps: first, we cover the model with a set of CTGs, for each of which we then extract all contained TCs. Exploiting the fact that the extraction of a CTG is completely determined by the model and the corresponding TP, our approach not only yields a test suite, but also a test plan as a set of TPs.

3.1 Covering the Model with Complete Test Graphs

We measure coverage by checking strong bisimilarity of the model with the combination of the (so far) generated CTGs; in the sequel, we refer to the latter as *cover*. This naturally leads to an iterative approach in the style of CEGAR



(a) before semi-composition (b) after semi-composition

Fig. 3. Cover corresponding to the CTG (combined with the empty initial cover)

(Counter-Example Guided Abstraction Refinement) [8]: as long as the current cover is not equivalent to the model, use the counterexample generated by the equivalence checker as a new TP.

Starting with the trivial, empty cover (i.e., an empty deadlocking IOLTS, containing one single state and no transitions), our approach relies on two operations, briefly described below, that are executed at each iteration: the computation of the cover obtained by adding a new CTG and the transformation of a counterexample into a TP. At the end of this section, we present two optimisations to speed up the approach by starting with a different initial cover.

Adding a CTG to the Cover. Computing the new cover obtained by adding a CTG involves three steps. In a first step, we remove all verdict transitions from the CTG. This can be achieved by simply hiding the verdict transitions. A variant of this first step removes also all transitions leading to an inconclusive verdict, considering that such transitions do not contribute to the coverage[4], balancing thus a lower with a more significant coverage. This variant has been implemented by a simple traversal of the CTG.

In a second step, we combine the CTG with hidden verdicts (as obtained by the previous step) with the cover computed so far by the overall loop. We construct the union of both IOLTSs, considering all but their initial states as different, and minimise the resulting IOLTS for weak trace equivalence [6]. Because both the cover and the CTG are included in the model, the resulting IOLTS is also included in the model modulo trace equivalence, but it might not be a subgraph of the model (in the sense of a graph homomorphism), as illustrated by Fig. 3(a). For instance, in the model in Fig. 1(a), the output !y leads from state

[4] When repeatedly executing a TC until a *Pass* or *Fail* verdict is reached, there is no guarantee that a transition leading to an *Inconc* verdict has been executed.

s_{iii} back to the initial state s_i, whereas in Fig. 3(a) the corresponding transition leads from state s_1 to s_5 (which is different from the initial state s_0).

In a third step, to ensure that the resulting cover for the next iteration is a subgraph of the model, we compute the semi-composition [15,16,24,12] of the model and the IOLTS obtained after the two previous steps—by [12, Proposition 3], the result of a semi-composition is a subgraph of the left operand. Also, the resulting cover contains all visible traces of the CTG (by definition of semi-composition and because by construction all traces of a CTG are included in the model). Notice that this third step reverts any unrolling of loops induced by the synchronous product of the model with the TP. Figure 3(b) shows the cover computed using these three steps to integrate the CTG shown in Fig. 1(c) with the empty initial cover.

Transforming a Counterexample into a TP. Because the model and the cover are both deterministic, a counterexample witnessing their non-bisimilarity is necessarily a (distinguishing) sequence leading to a transition missing in the cover. Also, using a breadth-first search algorithm in the equivalence checker yields a counterexample of minimal length; this is interesting to obtain a TP as simple as possible, and to avoid reexploring parts of the model already covered.

This sequence is then transformed into a TP by simply declaring its final state as an *Accept* state. A slightly more involved transformation adds to all non-accepting states a "*"-labelled transition to a *Refuse* state, enforcing that all TCs aim to execute the precise counterexample sequence. This yields a larger TP, but in general also a smaller associated CTG, containing fewer TCs. Another variant of TP generation, suitable for value-passing systems, applies *data abstraction* by replacing the data values with wildcards in the labels of the counterexample, generalising it for a larger coverage.

The iterative approach generates a series of TPs increasing in length. This series of TPs might contain some redundancy, if TP_i is the prefix of TP_j (with $i < j$) generated later in the process. Fortunately, TP_i can be safely ignored in this case, because all transitions covered by its corresponding CTG_i are also covered by CTG_j corresponding to TP_j. Because checking for each new TP whether one of the previous TPs is a prefix has a cost quadratic in the number of TPs, it is preferable, after all TPs have been generated, to scan them only once in reverse order, and to discard those the corresponding CTG of which do not increase the coverage, as this has a cost linear in the number of TPs.

Optimisations Starting with a Non-trivial Cover. Because our approach generates TPs of increasing length, it might be advantageous to start with a non-empty cover. We consider two orthogonal possibilities for initialising the cover.

A first idea is to start with the construction of a set of TPs covering all labels of the model. To this end, one can iteratively choose a label not yet covered, construct a simple TP aiming to reach this label, and integrate the corresponding CTG into the cover as described before. Instead of considering all labels, one can also apply this idea only to all *output* (respectively, *input*) labels. By construction, this fully automatic technique generates no redundant TPs.

Another idea is to initialise the cover using a set of TPs provided by the user. Indeed, the designer of a model knows in general a set of properties the model should satisfy. More often than not, these properties have already been formalised and verified on the model—this is often the main reason for developing the model. In general, these properties can be transformed into a *test plan*, in our case, a set of TPs. Using these TPs for the first iterations has the advantage of pinpointing untested parts of the model and giving a feedback of the test plan quality in terms of coverage. In particular, if in iteration $i + 1$, the new cover is strongly bisimilar to the cover computed in iteration i, the i-th CTG (and by transitivity, the i-th TP) can be considered redundant. Although this redundancy information might depend on the order the test plan is processed, this feedback is still valuable to streamline the test plan.

When combining these ideas, inserting the cover for all labels *after* taking into account the user-provided TPs (if any) and *before* the iterative completion of the cover is the most reasonable scheme, because it increases the likelihood of starting the generic loop with an already large cover.

3.2 Extracting all Test Cases Contained in a Complete Test Graph

A CTG contains all information to drive a (conformant) SUT towards the *Accept* states of the corresponding TP. In general however, a CTG is not controllable; extracting a TC from a CTG consists in choosing a solution for all controllability conflicts. Such a TC contains thus all information to drive a (conformant) SUT towards an *Accept* state of the TP for a particular sequence of inputs provided to the SUT. Hence, a single TC is not sufficient, because other possible input sequences are discarded.

To generate a test suite covering a CTG, we propose to ensure that for each controllability conflict, all possibilities to solve this conflict are taken into account by at least one TC of the test suite.

Because solving controllability conflicts as in [22, Section 4.5] is tailored to the efficient extraction of a single TC, it also guarantees that the extracted TC is a subgraph of the CTG. This is achieved by enforcing the constraint that for each state of the CTG, a single solution to the controllability conflict is adopted. However, due to this additional constraint, a part of the CTG might not be taken into account when extracting TCs, potentially leaving some parts (transitions and even states) of the model outside the scope of the test suite.

For instance, for the running example, the approach of [22] allows extracting two TCs, namely those shown in Fig. 4(a) and (b). However, these two TCs ignore the state s_8 of the CTG (see Fig. 1(c)), because the controllability conflict in state s_5 (a choice between the inputs ?a and ?b) is allowed to be solved only in a single way, and solving it by always choosing the input ?a (leading to state s_8 in Fig. 1(c)) would make the *Pass* verdict unreachable, if respecting the constraint that a TC has to be a subgraph of the CTG.

To extract a test suite taking into account all the information contained in a CTG, it might thus be necessary to unroll loops, duplicating some states. This can be automated by an iterative approach keeping track of the solutions to the

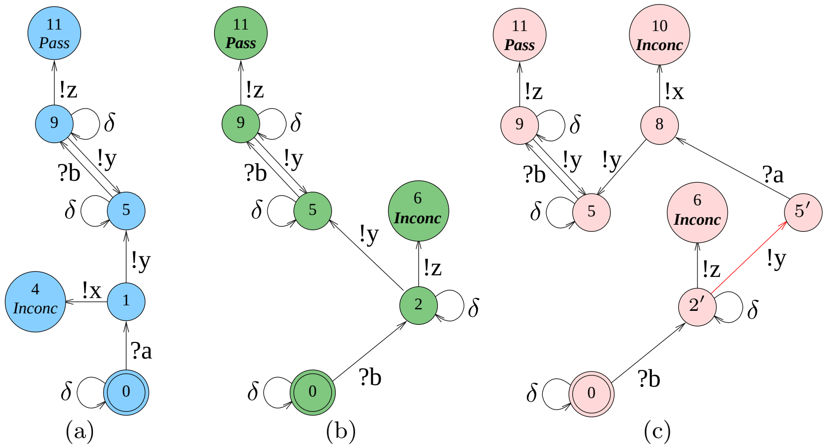


Fig. 4. Three test cases for the CTG in Fig. 1(c)

controllability conflicts of a CTG. While there is a not yet considered solution, i.e., a transition $s_1 \xrightarrow{a} s_2$ of the CTG not yet included in any TC, (1) apply the TC extraction algorithm [27] starting from the target state s_2 and (2) prefix the obtained TC by a sequence to s_1 followed by the transition $s_1 \xrightarrow{a} s_2$, duplicating the states of the CTG—states s'_2 and s'_5 in Fig. 4(c)—and completing the TC by appropriately handling outputs not present in the prefixed sequence, by applying the TC extraction algorithm—this adds the transition $s'_2 \xrightarrow{!z} s_6$ to the *Inconc* verdict in Fig. 4(c). A more detailed discussion of this technique and possible heuristics is unfortunately out of the space limitations for the present paper.

4 Experimental Evaluation

We evaluated our approach using the following 16 models of communication protocols and distributed systems[5]: ABP (demo 1) and ABP-data (demo 2) are dataless and data-aware variants of the Alternating Bit Protocol with controllable failures of the communication links; AAP and AAP-big (demo 33) are configurations of an asynchronous agreement protocol [2,33] with 1 and 10 rounds; BRP-basic, BRP, and BRP-big (demo 16) are variants of the Bounded Retransmission Protocol [28] with controllable message-loss, larger messages, and more retries; CAR-LNT is a purely asynchronous variant of a simple autonomous car [26]; CCP (demo 28) is a multi-processor cache-coherency protocol; CFS (demo 25) is a Cluster File System [31]; CIM (demo 34) is a computer-integrated manufacturing architecture [29]; DES and DES-basic (demo 38) are variants of the Data Encryption Standard [36] with visible subkeys [27, Section 3.4] and less iterations; TLS (demo 6) is the Transport Layer Security 2.3 handshake

[5] The CADP demo examples are available on `https://cadp.inria.fr/demos`.

protocol [5]; SMS (demo 40) is a stock management system [7]; and TOY is the running example given in [22].

The size of the 16 corresponding suspension automata ranges from 7 states and 15 transitions (TOY) upto 71 196 states and 377 448 transitions (CFS). The number of gates present on transition labels ranges from three (AAP, BRP and CCP) to twelve (AAP-big). All models except ABP and TOY are value-passing, with up to 90 labels (CAR-LNT) and 30 input labels (CFS) containing data values. Note that the number of input labels directly influences the number of possible choices when extracting TCs. Four models (AAP, AAP-big, SMS, and TLS) contain a deadlock, and several ones contain cycles.

Because our approach is for a large part implemented as an SVL [11] script, we could easily experiment with variants, along two orthogonal axes. On the one hand, we used various initialisations of the test plan: none (*completion*), constructing first a test plan to cover all labels (*all-labels*), and starting from a user-provided test plan (*manual*). On the other hand, when transforming a counterexample into a TP, we discarded (or kept) the data values, applying data abstraction (or not).

For evaluating the efficiency of our coverage approach, we compared the different settings based on the following metrics: (i) number of TCs and CTGs generated; (ii) total number of transitions of the test suite; (iii) the total runtime and peak memory usage; (iv) number of redundant TPs; and (v) number of transitions leading to an *Inconc* state. The number of transitions of the test suite and the number of transitions leading to an *Inconc* influence the cost of actually using the test suite, because the transitions of each TC might be executed during testing, and each *Inconc* verdict requires another execution (hopefully conclusive) of the TC.

4.1 Experiments with Automatically Building a Test Plan

Table 1 summarises the results of our experiments with fully automatic settings, i.e., without user-provided test plans. Columns entitled "trans" report the total number of transitions of all TPs, CTGs, and TCs. Columns entitled "red." report the number of redundant TPs (and thus CTGs). Columns entitled "inconc." report the total number of transitions leading to *Inconc* states in all TCs. Missing lines correspond to experiments taking more than 62 hours. In the case the extraction of a test suite did not finish in 62 hours, the execution time corresponds to the computation of a test plan. These results were obtained using the petitprince cluster of the Grid5000 testbed. None of the experiments required more than 65 MB of RAM.

Comparing the rows "completion — extraction" and "all-labels — completion — extraction", we observe that starting with the construction of a test plan covering all labels speeds up the approach and yields an overall smaller test plan and test suite. The speedup (of up to six times) is partially due to fewer redundant CTGs.

Comparing abstract and concrete TPs, we observe that for dataless examples (ABP and TOY), there is as expected no difference between abstract and

Table 1. Fully automatic test plan computation and test suite extraction

setting	Example	concrete TPs								abstract TPs							
		TP	CTG			TC			time	TP	CTG			TC			time
		trans	nb	trans	redun.	nb	trans	inconc.	(s)	trans	nb	trans	redun.	nb	trans	inconc.	(s)
completion — extraction	AAP	1,186	111	1,530	255	111	1,682	610	4,324	114	9	1,029	14	105	4,085	1,285	1,031
	AAP-big	66,148	1,848	127,431	7,860	1,848	1,306,679	88,711	108,947	4,756	119	104,660	435	1,819	876,200	373,044	18,246
	ABP	336	31	595	15	31	395	117	524	336	31	595	15	31	395	117	524
	ABP-data	144	22	142	6	22	162	10	363	40	6	110	2	26	282	18	281
	BRP-basic	136	16	168	12	16	179	33	323	58	6	156	4	21	357	76	230
	BRP	3,122	156	3,133	280	156	3,389	1,089	4,934	1,844	88	3,471	141	170	5,209	1,824	5,539
	BRP-big	203,186	5,122	201,791	12,643	5,122	108,700	3,777	252,529	39,976	936	206,034	2,201	5,289	172,710	134,386	73,326
	CAR-LNT	427,526	3,412	426,730	11,440	3,412	410,908	80,596	186,523	119,526	926	163,736	4,370	3,091	542,277	87,309	77,981
	CCP	22,258	1,249	19,250	1,145	1,249	21,660	4,223	30,802	622	29	11,543	88	1,590	50,856	13,291	11,876
	CIM	23,398	757	15,904	1,970	757	17,148	1,004	4,825	804	21	11,345	93	784	32,591	1,829	6,987
	DES	12,502	243	9,491	263	243	9,953	2,139	6,376	11,014	215	8,750	263	215	8,793	1,879	5,968
	DES-basic	2,364	142	2,292	89	142	2,534	422	3,182	1,884	108	2,071	89	108	2,045	352	2,705
	SMS	936	37	852	64	37	852	186	1,187	908	35	844	55	35	844	179	1,065
	TLS	1,004	61	5,471	518	61	5,471	4,908	6,115	332	19	3,679	40	65	6,123	5,411	1,034
	TOY	32	4	30	7	4	30	3	117	32	4	30	7	4	30	3	117
all-labels — completion — extraction	AAP	970	83	1,596	151	109	2,814	1,212	3,024	104	9	1,131	8	105	4,565	1,122	949
	AAP-big	34,792	848	81,973	2,302	1,427	2,578,585	345,603	37,175	4,196	100	104,693	169	1,861	545,830	452,238	15,557
	ABP	228	21	552	5	34	509	60	373	228	21	552	5	34	509	60	373
	ABP-data	2	1	36	0	6	77	0	56	2	1	36	0	6	77	0	56
	BRP-basic	2	1	52	1	7	190	0	62	54	6	155	3	21	354	72	217
	BRP	2	1	520	1	67	5,457	0	420	1,838	88	3,496	133	172	5,257	1,824	3,280
	BRP-big	2	1	19,722	1	1,673	135,429	0	11,412	39,970	936	206,203	2,177	5,340	322,324	134,506	72,635
	CAR-LNT	10	5	55,113	0	5,045	40,878,934	713,797	31,521	4	2	28,742	0	2,838	27,859,872	364,392	19,361
	CCP	2,828	155	5,734	78	1,144	134,524	522	9,747	562	25	11,423	66	1,688	57,403	13,024	12,181
	CFS	2,520	66	382,943	51	> 36,000 TCs			1,869	2	1	379,471	1	> 36,000 TCs			38
	CIM	734	19	3,235	55	485	93,555	18	4,249	116	4	4,236	3	562	48,897	94	4,128
	DES	2	1	516	0	155	30,674	0	1,033	2	1	516	1	146	32,468	0	981
	DES-basic	2	1	246	0	73	941	0	532	2	1	246	1	71	4,370	0	524
	SMS	2	1	132	0	25	2,599	75	177	2	1	132	0	25	2,599	75	170
	TLS	4	2	1,067	10	61	1,977	1,580	480	4	2	1,067	0	61	1,860	440	404
	TOY	28	4	32	3	4	33	2	76	28	4	32	3	4	33	2	76

concrete TPs. For most examples, generating abstract TPs improves the overall quality of the test plan. For all examples except the variants of BRP, we observe less TPs and a smaller total number of transitions in the test plan. An explanation is that a CTG generated for a concrete TP is in general controllable, yielding a test plan as large as the test suite. Using abstract TPs also builds the test plan faster, mostly because fewer redundant CTGs are generated. According to our experiments, the benefits of this data abstraction are particularly visible when not starting with an initial test plan covering all labels.

However, for BRP examples, abstract TPs are worse than concrete ones, because the same gate is used at the beginning of the protocol with some particular data values and at the end of the protocol with completely different values. As discarding data values annihilates this *control*-related distinction, the resulting abstraction is too coarse and degrades the performance. Splitting these labels of BRP using two different gates removes the ambiguity and leads to similar performance improvements as for the other examples.

As a summary for fully automatic settings, we found that, in general, starting with a test plan covering all labels and using abstract TPs (highlighted block in Table 1) is the best option. In particular, this setting produces a smaller test plan, because the abstract TPs generate fewer redundancies. However, a smaller test plan does not always yield a smaller test suite, because a more generic TP yields more TCs, in particular without an initial test plan covering all labels (see for instance CCP and CIM). These larger test suites might also contain more transitions to *Inconc* states.

4.2 Experiments with Completing a User-Provided Test Plan

We also applied our approach to complement an existing test plan for the selected models. The TPs making up the test plan either existed as such in the case study (ABP, CAR-LNT, DES, DES-basic, TLS, and TOY), or were created by transforming temporal logic properties verified in the case study. The latter step is not trivial for safety properties, which formalise bad things that (should) never happen: using such a property directly as a TP would yield a trivial CTG always resulting in an *Inconc* verdict. Thus, the property has to be transformed into a TP expressing the expected correct behaviour.

Our results when starting from a user-provided test plan are summarised in Table 2, which adds three columns to those of Table 1: the "user" column gives the number of user-provided TPs; the "ign." column gives the number of user-provided TPs that are flagged as redundant later-on; and the "gen." column gives the number of non-redundant automatically generated TPs. The "nb" column gives the number of CTGs corresponding to all non-redundant TPs (both user-provided and automatically generated). Since most user-provided TPs contain concrete, semantically meaningful data values, we did not apply data abstraction on the provided test plan. However, we applied data abstraction for all TPs automatically generated, because (for all examples but TLS) we observed that completion with abstract TPs resulted in a smaller test plan (measured in the

Table 2. Test suite generation starting from a user-provided test plan

Example	user-provided — all-labels — completion — extraction										
	TP				CTG			TC			time
	user	ign.	gen.	sum	nb	sum	redun.	nb	sum	inconc.	(s)
AAP	2	1	8	103	9	1,131	4	107	4,583	1,122	777
AAP-big	2	0	99	4,198	101	104,975	166	1,887	546,095	452,238	15,716
ABP	3	2	0	3	1	524	2	53	2,696	0	356
ABP-data	6	4	0	15	2	92	4	40	546	0	193
BRP-basic	3	1	0	5	1	76	2	18	573	0	134
BRP	4	3	0	5	1	575	3	105	5,578	0	666
BRP-big	4	3	0	5	1	19,930	3	2,602	154,197	120	17,429
CAR-LNT	2	0	0	4	2	28,742	0	2,838	27,859,872	364,392	17,138
CCP	1	0	24	563	25	11,423	64	1,688	57,403	13,024	10,741
CIM	6	5	0	29	1	4,392	5	794	200,562	0	5,559
DES	1	0	0	6	1	1,845	0	485	354,766	0	3,209
DES-basic	1	0	0	6	1	854	0	259	32,254	0	1,663
SMS	2	1	1	4	2	263	1	50	5,174	200	356
TLS	3	3	18	302	18	3,915	23	63	6,096	5,359	824
TOY	1	1	4	28	4	32	3	4	33	2	76

number of TPs and their total number of transitions). We omitted CFS from Table 2, because the generation of a test plan timed out after 62 hours.

Except for CCP, SMS, TLS, and TOY, the CTGs generated starting from the provided test plan cover all the labels of the model. The running example TOY comes with only one TP to illustrate test case extraction. CCP is an academic example, for which we crafted a single, rather abstract TP. For SMS and TLS, the provided test plan even covers almost all transitions. We can note many redundancies in some provided test plans (ABP, ABP-data, all variants of BRP, CIM, SMS, and TLS), which could indicate that the properties focus on a specific part of the model. Interestingly, for CFS, extending a user-provided test plan timed out, contrary to starting with a test plan covering all labels. It is worth noting that the user-provided test plan covers all labels, but only 348,327 of the 377,448 transitions in the suspension automaton for CFS—the iterative extension yields more than 8,716 TPs.

Some of the provided TPs target a particular behaviour, sometimes in a very precise manner. Therefore, the corresponding test suite contains already a large number of transitions to an *Inconc*, and this number only increases by completing the test plan. An exception is CAR-LNT, where the two provided TPs specify the two only ways to reach the final state.

5 Related Work

Although various coverage criteria have been proposed [30] and intensively studied [40] for software testing, only a few approaches target behavioural coverage of LTSs. In the following, we focus on the most closely related approaches.

The diagnostic generation features of model checkers have been used to generate test suites. For EFSMs (Extended Finite-State Machines), model checking a coverage criterion (specified as a temporal logic formula in a CTL fragment) produces as witnesses finite sequences. Two kinds of structural coverage criteria [21], based on control and data flow, are well-suited for deterministic, sequential EFSMs. Our TPs have a similar abstraction level as the CTL formulas in [21], but with the advantage of producing CTGs encompassing sets of test cases rather than individual witness sequences.

Conformance testing for concurrent systems has been studied from several perspectives, with various notions of behavioural coverage. In [10], a concurrent system is modelled as an IOPN (Input/Output Petri Net), whose semantics is its unfolding into an IOLES (Input/Output Labelled Event Structure), equipped with *co-ioco*, a generalisation of the *ioco* conformance relation to concurrent systems. TCs are generated from finite IOLESs for two coverage criteria: all paths of length n and unfolding all cycles k times, respectively. In our approach, we opted for a counterexample-driven behavioural coverage, that can be further refined towards partially covering paths of length n (by abstracting some of the concrete labels in the counterexample TPs).

Generalising a classical FSM test generation method to the IOLTS setting, *ioco*-based complete test suites can be generated from deterministic IOLTSs [9]. Completeness is achieved by generating test suites for mutant IOLTSs (aka fault domains) modelling potential faults in the SUT, so that an SUT with the faults will be detected. The considered coverage criteria ensure that each state and transition of the SUT corresponds to some state or transition of the model. Most of the classical FSM-based testing approaches [25] aim at detecting faults in the SUT, whereas conformance testing for concurrent systems aims at establishing that an SUT fulfils the requirements expressed by the formal model [17]. Therefore, we focused primarily on behavioural coverage instead of fault coverage. Experiments have shown that conformance testing has at least the same fault detection capabilities as manual testing of an SUT, and significantly better error detection capabilities in the system requirements [32].

TestComposer [23] (combining the TVEDA tool [18] with TGV [22]) and Autolink [35] were two industrial tools for the automatic test generation for SDL specifications. Similar to our approach, both first generate a test plan (a set of TPs), from which in a second step a set of test cases is extracted [34]. TestComposer relies for this second step on TGV. A notable difference to our approach is that both TestComposer and Autolink aim at structural coverage of the SDL specification, whereas we focus on (all transition) coverage of the underlying semantic model. Consequently, the techniques to construct TPs of TestComposer and Autolink differ from ours: besides manual exploration, both tools rely on specialised simulation and state space exploration algorithms, whereas we use a generic equivalence checker. Also, to the best of our knowledge, none of these tools aim at systematic generation of *all* test cases for a given TP.

Another approach for coverage-driven conformance testing was proposed in the AGEDIS project [19], by combining the test generation features of TGV and the usage of *test directives*. A test directive may specify constraints on data and also data-driven coverage criteria (e.g., cover all transitions whose source and target states satisfy specific data constraints) as in the GOTCHA tool [3]. Similarly, Uppaal-Cover uses observer automata used to express coverage criteria of test cases generated from FSM specifications [20,4]. Observer automata may specify structural criteria, dataflow criteria, and semantic coverage, but are deterministic, i.e., only a unique response and target state of the controller can be anticipated for a given state and input. Our TESTOR tool enables to describe data-handling TPs directly in the high-level language LNT [14] using multiway rendezvous. In this way, one can specify a data-driven coverage criterion by using a data-handling TP, generating the corresponding CTG, and applying the second step of our approach to extract all TCs contained in the CTG.

6 Conclusion

We proposed an automatic approach to generate a conformance test suite covering all transitions of an IOLTS M, which models the behaviour of a concurrent system. The approach proceeds in two steps. In a first step, we generate a test plan, i.e., a set of test purposes whose corresponding CTGs (complete test graphs) cover all transitions of M. This step can start either from scratch, or from a set of test purposes provided by the user. In the latter case, the approach completes the test plan to cover all transitions of M, and can also spot redundancies in the user-provided test purposes. In a second step, we produce a test suite from the test plan by extracting, from each CTG, a set of test cases covering all transitions of the CTG. We implemented this approach on top of the CADP toolbox and the TESTOR tool, and experimented it on 16 case studies. We studied possible variants and heuristics to fine-tune both the overall performance of the extraction and the quality of the resulting test plan or test suite.

Concerning future improvements, it is possible to speed up the approach by extracting test cases from different CTGs in parallel. To obtain a smaller test plan, one could also study more sophisticated heuristics when checking whether a CTG improves the overall cover, for instance by considering different, more informed orders of handling the CTGs. Finally, the built-in data abstraction (discarding the data values on the labels of counterexample sequences) could be generalised, for instance by allowing the user to specify a set of labels that should always be kept concrete.

Acknowledgments. This work was partly supported by project ArchitectECA2030 that has been accepted for funding within the Electronic Components and Systems for European Leadership Joint Undertaking in collaboration with the European Union's H2020 Framework Programme (H2020/2014-2020) and National Authorities, under grant agreement No. 877539. 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`). We are also grateful to Frédéric Lang for inspiring discussions.

References

1. A. Belinfante. JTorX: A Tool for On-line Model-driven Test Derivation and Execution. *Proc. of TACAS'10*, LNCS 6015, pp. 266–270, 2010.
2. M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. *Proc. of PODC*, 1983.
3. M. Benjamin, D. Geist, A. Hartman, G. Mas, R. Smeets, and Y. Wolfsthal. A Study in Coverage-Driven Test Generation. *Proc. of DAC'99*, pp. 970–975, 1999.
4. J. Blom, A. Hessel, B. Jonsson, and P. Pettersson. Specifying and generating test cases using observer automata. *Proc. of FATES'04*, LNCS 3395, pp. 125–139, 2004.
5. J. Bozic, L. Marsso, R. Mateescu, and F. Wotawa. A formal TLS handshake model in LNT. *Proc. of MARS'18*, EPTCS 268, 2018.
6. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *J. ACM*, 31(3):560–599, 1984.
7. A. Chirichiello and G. Salaün. Encoding abstract descriptions into executable web services: Towards a formal development. *Proc. of WI'05*, pp. 457–463, 2005.
8. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. *Proc. of CAV*, LNCS 1855, pp. 154–169, 2000.
9. A. da Silva Simão and A. Petrenko. Generating Complete and Finite Test Suite for ioco: Is It Possible? *Proc. of MBT 2014*, pp. 56–70, 2014.
10. H. P. de León, S. Haar, and D. Longuet. Model-based Testing for Concurrent Systems: Unfolding-based Test Selection. *STTT*, 18(3):305–318, 2016.
11. H. Garavel and F. Lang. SVL: a Scripting Language for Compositional Verification. *Proc. of FORTE'01*, pp. 377–392, 2001.
12. H. Garavel, F. Lang, and R. Mateescu. Compositional Verification of Asynchronous Concurrent Systems Using CADP. *Acta Informatica*, 52(4):337–392, 2015.
13. 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.
14. H. Garavel, F. Lang, and W. Serwe. From LOTOS to LNT. *ModelEd, TestEd, TrustEd*, LNCS 10500, pp. 3–26, 2017.
15. S. Graf and B. Steffen. Compositional Minimization of Finite State Systems. *Proc. of CAV'90*, LNCS 531, pp. 186–196, 1990.
16. S. Graf, B. Steffen, and G. Lüttgen. Compositional Minimization of Finite State Systems Using Interface Specifications. *Formal Aspects of Computing*, 8(5):607–616, Sept. 1996.
17. R. Groz, O. Charles, and J. Renévot. Relating Conformance Test Coverage to Formal Specifications. *Proc. of FORTE*, pp. 195–210, 1996.
18. R. Groz and N. Risser. Eight Years of Experience in Test Generation from FDTs using TVEDA. In *Proc. of FORTE X*, pp. 465–480, 1997.
19. A. Hartman and K. Nagin. The AGEDIS Tools for Model Based Testing. *Proc. of ISSTA'04*, pp. 129–132, 2004.
20. A. Hessel and P. Pettersson. Model-Based Testing of a WAP Gateway: An Industrial Case-Study. In *Proc. of FMICS*, LNCS 4346, pp. 116–131, 2006.
21. H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A Temporal Logic Based Theory of Test Coverage and Generation. *Proc. of TACAS'02*, LNCS 2280, pp. 327–341, 2002.
22. 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.

23. A. Kerbrat, T. Jéron, and R. Groz. Automated Test Generation from SDL Specifications. *Proc. of SDL'99*, pp. 135–152, 1999.
24. J.-P. Krimm and L. Mounier. Compositional State Space Generation from LOTOS Programs. *Proc. of TACAS'97*, LNCS 1217, 1997.
25. D. Lee and M. Yannakakis. Optimization Problems from Feature Testing of Communication Protocols. *Proc. of ICNP*, pp. 66–75, 1996.
26. L. Marsso, R. Mateescu, I. Parissis, and W. Serwe. Asynchronous testing of synchronous components in GALS systems. *Proc. of IFM'19*, LNCS 11918, pp. 360–378. Springer, Dec. 2019.
27. L. Marsso, R. Mateescu, and W. Serwe. TESTOR: A Modular Tool for On-the-Fly Conformance Test Case Generation. *Proc. of TACAS'18*, LNCS 10806, pp. 211–228, 2018.
28. R. Mateescu. Formal description and analysis of a bounded retransmission protocol. *Proc. of COST 247*, pp. 98–113, University of Maribor, 1996.
29. S. Mauw. *Process Algebra as a Tool for the Specification and Verification of CIM-architectures*, *Cambridge Tracts in Theoretical Computer Science* 17, pp. 53–80, 1990.
30. G. J. Myers. *The art of software testing (2. ed.).* Wiley, 2004.
31. C. Pecheur. Advanced Modelling and Verification Techniques Applied to a Cluster File System. *Proc. of ASE'99*, 1999.
32. A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One Evaluation of Model-based Testing and its Automation. *Proc. of ICSE'2005*, pp. 392–401, 2005.
33. M. O. Rabin. Randomized byzantine generals. *Proc. of the IEEE Symposium on Foundations of Computer Science*, pp. 403–409, 1983.
34. M. Schmitt, Ebner, and J. Grabowski. Test Generation with Autolink and Test-Composer. *Proc. of SAM'2000*, SDL Forum 2000.
35. M. Schmitt, J. Grabowski, D. Hogrefe, and B. Koch. Autolink—A Tool for the Automatic and Semi-Automatic Test Generation. *Formale Beschreibungstechniken für verteilte Systeme*, *GMD-Studien* 315, pp. 333–341, 1997.
36. W. Serwe. Formal Specification and Verification of Fully Asynchronous Implementations of the Data Encryption Standard. *Proc. of MARS*, EPTCS 196, 2015.
37. R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE TSE*, 18(3):206–215, 1992.
38. J. Tretmans. Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. *Computer networks and ISDN systems*, 29(1):49–79, Dec. 1996.
39. M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-based Testing Approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.
40. Q. Yang, J. J. Li, and D. M. Weiss. A survey of coverage-based testing tools. *Computer Journal*, 52(5):589–597, 2009.
41. J. Zander, I. Schieferdecker, and P. J. Mosterman, editors. *Model-Based Testing for Embedded Systems.* Computational Analysis, Synthesis, & Design Dynamic Systems, 2011.
42. H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.