An Executable Mechanised Formalisation of an Adaptive State Counting Algorithm

Robert Sachtleben

An Executable Mechanised Formalisation of an Adaptive State Counting Algorithm

Robert Sachtleben[1]

Department of Mathematics and Computer Science, University of Bremen, Bremen, Germany
rob_sac@uni-bremen.de

Abstract. This paper demonstrates the applicability of state-of-the-art proof assistant tools to establish completeness properties of a test strategy and the correctness of its associated test generation algorithms, as well as to generate trustworthy executable code for these algorithms. To this end, a variation of an established test strategy is considered, which generates adaptive test cases based on a reference model represented as a possibly nondeterministic finite state machine (FSM). These test cases are sufficient to check whether the reduction conformance relation holds between the reference model and an implementation whose behaviour can also be represented by an FSM. Both the mechanical verification of this test strategy and the generation of a provably correct implementation are performed using the well-known Isabelle/HOL proof assistant.

Keywords: Complete Test Methods · Finite State Machines · Isabelle/HOL · Mechanised Proofs· Proof Assistants · Reduction

1 Introduction

Objectives In this paper, we present a mechanised proof for a variation of the complete test strategy originally published by Petrenko and Yevtushenko in [21] and provide a formalisation of selected algorithms described by the same authors in [19] to calculate concrete test suites. We generate executable code from this formalisation to provide a trustworthy implementation of the test strategy.

The formalised strategy facilitates verifying the reduction conformance relation between two finite state machines (FSMs), of which the first serves as a reference model representing a specified behaviour, whereas the second represents the true behaviour of the system under test (SUT). Both FSMs may be nondeterministic but are assumed to be completely specified and observable. Additionally assuming an upper bound on the number of states contained in the (unknown) FSM representing the behaviour of SUT, the strategy generates finite test suites guaranteeing complete fault coverage using a state counting method. These test suites are adaptive and thus the number of tests applied to the SUT during testing depends on its observed behaviour. In many situations, the generated test suites are therefore significantly smaller than those generated using

the well-known "brute force" strategy based on product FSMs[1], which requires $O(|\Sigma_I|^{mn})$ test cases[2] for the same effect.

Motivation We advocate an approach to systematic testing that includes formal proofs of fault coverage capabilities of test strategies, so that no doubt with respect to their test strength remains. This process entails making explicit any underlying hypotheses, such as the specification of fault domains. Since complete test strategies are of considerable importance for the verification of safety-critical systems, the correctness of fault coverage claims for a given strategy is crucial from the perspective of system certification. Thus, we further advocate the use of proof assistants to check the proofs, as we believe that due to the large number of different test strategies and the often intricate nature of their corresponding proofs it cannot be expected that each proof is manually checked by many members of the testing community. Our previous work [22] supports this view, as it uncovered an ambiguity in the textual description of a test strategy, which could lead to incomplete implementations. In [22] we also voiced our hope that mechanised proofs could be used as artefacts presented to certification authorities as a means of showing that an applied test strategy provides the fault detection capabilities claimed for it. We now additionally believe that the use of provably correct code generated from formalisations could reduce the effort required to provide convincing arguments for tool qualification of tools employing such code.

Provably correct implementations of the test strategy formalised in this paper can be employed as trustworthy components in a large variety of testing applications. These include the generation of test suites for testing whether an SUT behaves only in ways allowed in a specification that can be represented as a (possibly nondeterministic) completely specified FSM, where nondeterminism indicates that the SUT may omit some reactions. Such specifications include communication protocols that contain optional behaviour. Additionally, many interesting conformance relations can be reduced to testing for this reduction conformance relation between completely specified FSMs, including quasi-equivalence and quasi-reduction for incomplete specification FSMs (see [10]) and reduction for reactive I/O transition systems (see [12]). This in turn allows for trustworthy implementations to be employed in the process of testing, for example, embedded systems against specifications given in SysML (see [11]).

Main Contributions To our best knowledge, this is the first time that a mechanised proof for the complete reduction test strategy elaborated in [21] is presented. Moreover, trustworthy executable code is generated from a mechanically verified formalisation of an algorithm that realises the strategy, incorporating selected algorithms from [19]. This provably correct implementation is used as the trustworthy core of a set of testing tools.

[1] This strategy has been described, for example, in the lecture notes [17, Section 4.5].

[2] $|\Sigma_I|$ is the size of the input alphabet, n the number of states in the reference model, and m an upper bound for the number of states in the SUT model.

Related Work The first complete state counting approach to reduction testing has been published in [20], specialising on the case of deterministic implementations being tested against nondeterministic reference models for language inclusion. Later, adaptive state counting has been proposed as an optimisation in [19]. The restriction to deterministic implementations has then been dropped in later works considering a more general formulation of the problem, admitting both nondeterministic reference models and implementations. This has been studied in [9] and [21], the latter being the article the present paper is based on.

To our best knowledge, applying proof assistants to testing has first been advocated in [4]. Using Isabelle/HOL at its core, an integrated testing framework has been developed by the same authors and described in [5]. This framework allows for elaboration of test strategies (called *test theorems* in [5]), fault coverage proof, test case and test data generation in the same tool. Several cases of mechanised proofs establishing the completeness of testing theories are elaborated by the authors, not including the strategy analysed in the present paper.

Our approach to model-based testing (MBT) contrasts to that advocated in [5] in that we favour the use of specialised tools for strategy elaboration (Isabelle/HOL), modelling (FSM and SysML modelling tools), and test case and test data generation (RT-Tester [16] with SMT solver [18]). We base this preference on the possibility of performing SMT solving internally, without explicit interactions with the users, which requires less specialised expertise than the interactive handling of proof assistants. In this we agree with [1].

In [22] we presented a mechanised proof for the strategy described in [9], but in contrast to the present paper, this effort did not yet focus on providing a formalisation from which provably correct executable code could be generated.

Finally, tool qualification for model-based testing tools is discussed for example in [3], where the replaying of test executions against the specification model is introduced as a measure to uncover potential faults in the untrusted test case generation. Our approach obviates the need for many such measures by employing trustworthy code to generate test suites.

Reference to Online Resources The Isabelle/HOL session containing all the formalisations and proofs elaborated in this paper and a set of command line tools which employ the executable code generated from this session to generate test suites and apply them to SUTs, together with corresponding documentation and a short evaluation of the tools applied to a set of randomly generated FSMs, are publicly available on `https://bitbucket.org/RobertSachtleben/an-executable-formalisation-of-an-adaptive-state-counting`.

Overview Section 2 provides a short overview of the test strategy presented in [21]. Next, Section 3 describes our formalisation of this strategy in Isabelle and outlines the strategy of our mechanical completeness proof. Thereafter, Section 4 describes the generation of a trustworthy implementation, its integration into a set of test tools, and experiments performed using this implementation. Finally, we provide conclusions in Section 5.

2 Overview of the Formalised Test Strategy

The test strategy described in [21] and formalised mechanically in this paper is employed to check whether an SUT, whose behaviour is assumed to correspond to some unknown finite state machine M', conforms to a specification, given as a finite state machine M, with respect to the reduction conformance relation. We first introduce the constructions used within this strategy as detailed in [21].

Finite State Machines A *finite state machine* $M = (Q, q_0, \Sigma_I, \Sigma_O, h)$ is 5-tuple consisting of a finite set Q of states, an *initial state* $q_0 \in Q$, finite sets Σ_I and Σ_O constituting the input and output alphabet, respectively, and a transition relation $h \subseteq Q \times \Sigma_I \times \Sigma_O \times Q$ where $(q_1, x, y, q_2) \in h$ if and only if there exists in M a transition from q_1 to q_2 for input x that produces output y. We define the size of M, denoted $|M|$, by the number $|Q|$ of states it contains. Finally, the *language* $\mathcal{L}(M, q)$ denotes the set of all sequences $\bar{x}/\bar{y} \in (\Sigma_I \times \Sigma_O)^*$ of input-output (IO) pairs such that M can react to $\bar{x}$ applied to q with outputs $\bar{y}$. The language of M itself, denoted $\mathcal{L}(M)$, is the language of its initial state. We write $\bar{x}\bar{x}'/\bar{y}\bar{y}'$ for the concatenation of IO-sequences $\bar{x}/\bar{y}$ and $\bar{x}'/\bar{y}'$.

Primarily based on its transition relation, further properties of an FSM M can be distinguished: M is *deterministic* if for any state q in M and input x at most one transition exists. M is *observable* if for each contained state q, input x and output y there is at most one state q' that is reached from q through a transition with input/output x/y, i.e. there is at most one state q' in Q such that $(q, x, y, q') \in h$. That is, the target state reached from some state with some input can be uniquely determined using the observed output. This property also extends to IO-sequences, as the state reached by an IO-sequence $\bar{x}/\bar{y} \in \mathcal{L}(M, q)$ applied to state q is again uniquely determined. Next, M is *completely specified* if a transition exists from each contained state q for each contained input x. That is, any input $x \in \Sigma_I$ applied to a state $q \in Q$ must produce some output. Furthermore, M is *acyclic* if $\mathcal{L}(M)$ is finite, and a state q is a *deadlock state* if no transition from q exists. Also, M is *single-input* if for each state q all transitions from q share the same input component. Finally, an FSM $M' = (Q', q_0, \Sigma_I', \Sigma_O', h')$ is a *submachine* of M if $Q' \subseteq Q$ and $h' \subseteq h$ hold.

In the remainder of this paper we assume every FSM to be both observable and completely specified, which is no restriction, as there exist techniques to complete any FSM (see [9]) and to transform it into a language-equivalent observable minimised machine [14]. We do not, however, require any FSM to be deterministic. We furthermore assume that FSM M', representing the behaviour of the SUT, uses the same inputs as the specification M and satisfies $|M'| \leq m$.

The *product machine* (intersection) of FSMs $M_1 = (S, s_0, \Sigma_I, \Sigma_O, h_1)$ and $M_2 = (T, t_0, \Sigma_I, \Sigma_O, h_2)$ is an FSM $PM = (S \times T, (s_0, t_0), \Sigma_I, \Sigma_O, h)$, which generates the language intersection $\mathcal{L}(PM) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$ by the following construction of h: $((s, t), x, y, (s', t')) \in h \iff (s, x, y, s') \in h_1 \wedge (t, x, y, t') \in h_2$.

Finally, a state q_1 of an FSM M_1 is a *reduction* of some state q_2 of an FSM M_2 if and only if $\mathcal{L}(M_1, q_1) \subseteq \mathcal{L}(M_2, q_2)$ holds, and M_1 is a reduction of M_2 if and only if the initial state of M_1 is a reduction of the initial state of M_2.

State Preambles and Definitely Reachable States A *state preamble* P of $M = (Q, q_0, \Sigma_I, \Sigma_O, h)$ for some $q \in Q$ is an acyclic single-input submachine of M with q being its single deadlock state such that for every state $q' \neq q$ in P all transitions of M from q' with the single input defined for q' in P are also contained in P. That is, a state preamble P of M for q can be seen as a strategy for reaching q in every completely specified submachine of M and thus, for any sequence $\bar{x}/\bar{y} \in \mathcal{L}(P) \cap \mathcal{L}(M')$ that reaches q in P, the state reached by $\bar{x}/\bar{y}$ in the SUT representation M' must be a reduction of q for M' to be a reduction of M. A state q of M is called *definitely reachable* if and only some state preamble of M for q exists. We say that a state of M' is *reached* by a preamble P if it is reached by some $\bar{x}/\bar{y}$ that reaches q in P. We call such $\bar{x}/\bar{y}$ *preamble sequences*.

Note here that the initial state q_0 of M is definitely reachable by the preamble $P_0 = (\{q_0\}, q_0, \Sigma_I, \Sigma_O, \emptyset)$. This implies that if a set PS of pairs (q, P) of states of M and corresponding preambles contains (q_0, P_0) and also $\mathcal{L}(M') \not\subseteq \mathcal{L}(M)$ holds, then a minimal length IO-sequence $\bar{x}_f/\bar{y}_f$ and additionally a sequence $\bar{x}_p/\bar{y}_p$ that reaches a deadlock state in a preamble in PS must exist such that $\bar{x}_p\bar{x}_f/\bar{y}_p\bar{y}_f \in \mathcal{L}(M') \setminus \mathcal{L}(M)$ holds, since $\mathcal{L}(P_0)$ contains only the empty sequence ϵ. In the following, we will refer to sequences such as $\bar{x}_f/\bar{y}_f$ in short as *minimal sequences to a failure extending* $\bar{x}_p/\bar{y}_p$ if the set PS is obvious from the context.

In the test strategy, for each definitely reachable state q of M a preamble P for q is used to identify states in the SUT representation that must conform to q, which include all states in M' reached by sequences that reach q in P.

State Separators A *state separator* S of M for states q_1 and q_2 is an acyclic single-input FSM with two reachable deadlock states d_1 and d_2 such that (1) for any $\bar{x}/\bar{y} \in \mathcal{L}(S)$ it holds that if $\bar{x}/\bar{y}$ reaches d_1 then $\bar{x}/\bar{y} \in \mathcal{L}(M, q_1) \setminus \mathcal{L}(M, q_2)$ holds whereas if $\bar{x}/\bar{y}$ reaches d_2 then $\bar{x}/\bar{y} \in \mathcal{L}(M, q_2) \setminus \mathcal{L}(M, q_1)$ holds, and (2) for any sequence $\bar{x}/\bar{y} \in \mathcal{L}(S)$ reaching some non-deadlock state q in S with a single defined input x, S contains a transition from q with output y for each output y produced by any of the states reached in M by via $\bar{x}/\bar{y}$ from q_1 or q_2 to x. That is, S provides a strategy of reliably distinguishing q_1 and q_2 in all complete submachines of M. States q_1 and q_2 of M a called *r-distinguishable* if and only if a state separator of M for them exists. In the following, test separators S for states q_1 and q_2 are used to check whether certain states in M' behave like only one of q_1 or q_2, or neither of them.

Adaptively Testing of the Reduction Conformance Relation The SUT conforms to the specification FSM M with respect to the reduction conformance relation if and only if the FSM M', which is assumed to represent the behaviour of the SUT, is a reduction of M. That is, the SUT conforms to M if and only if every behaviour of it is also admissible in M. As the languages of completely specified FSMs with nonempty input alphabets are infinite, it is not possible to check this property by simply enumerating $\mathcal{L}(M)$ and $\mathcal{L}(M')$ and therefore test suites must be applicable in a finite amount of time. In the case of nondeterministic FSMs this furthermore requires some fairness assumption that all reactions

to some input can be observed within a finite number of applications of this input. In this paper, we employ the *complete testing assumption* (see [9]) and thus assume that there exists some upper bound k on the number of applications required to observe all reactions to a given sequence of inputs.

Furthermore, an SUT might exhibit only a proper subset of the behaviours of M, which enables the use of adaptivity in testing by controlling the application of inputs based on the observed behaviour of the SUT. An *adaptive test case* (ATC) A for FSM M is an acyclic single-input FSM that is output-complete for M (for every non-deadlock state there exists a transition for every output of M) and which may contain a designated deadlock state *fail*. FSM M' in state q' *passes* A if and only if there exists no sequence in $\mathcal{L}(A) \cap \mathcal{L}(M', q')$ that reaches *fail* in A. *Applying* A to M' thus reduces to calculating $\mathcal{L}(A) \cap \mathcal{L}(M', q')$ and checking the states reached in A, which is feasible as $\mathcal{L}(A)$ is finite.

For example, a state separator S of M for q_1 and q_1 can be transformed into an ATC I_{q_1} that checks whether the state it is applied to behaves like q_1 and not like q_2 by replacing d_2 with *fail* and adding to every non-deadlock state q a transition to *fail* for every output of M that is not produced by q in S.

Finally, adaptive test cases can be *concatenated* at their deadlock states. Given ATCs A_1 and A_2 and a deadlock state $d \neq \mathit{fail}$ of A_1, $A_1 @_d A_2$ denotes the ATC created by replacing d in A_1 with the initial state of A_2 and inserting all transitions and other states of A_2 into A_1. For simplicity we assume here that the state sets of ATCs are pairwise disjoint, ensuring that the @ operator is associative and well-behaved when concatenating multiple ATCs.

2.1 Overview of the Formalised Adaptive State Counting Algorithm

The adaptive state counting algorithm re-verified in this paper creates a test suite for M and an assumed upper bound m on the number of states in M' as a set of adaptive test cases. This is performed by first calculating a preamble for each definitely reachable state of M and a state separator for every pair of r-distinguishable states in M, followed by a process of extending sequences from the definitely reachable states until they are too long to be proper prefixes of any minimal sequence to a failure, resulting in so-called *traversal sets*. The termination criterion for this extension is based on the number of r-distinguishable states of M encountered during the application of a sequence, which can constitute a lower bound on the number of states in any non-conforming FSM M'. Finally, a test suite is constructed by concatenating preambles, extended sequences and state separators in accordance with this termination criterion.

Traversal Sets Let RD denote the set of all maximal sets of pairwise r-distinguishable states of Q. Given a set $R \in RD$ let R_{dr} denote the subset of R containing only definitely reachable states. Note that every state of M is contained in some such R and that R_{dr} may be empty.

For every definitely reachable state q of M the set $N^m(q)$ of *m-traversal sequences* is then constructed starting from the empty sequence by extending

input sequences until they satisfy the following rule: An input sequence $\bar{x}$ is not extended further if for all $\bar{x}/\bar{y} \in \mathcal{L}(M, q)$ there exists some $R \in RD$ such that $\bar{x}/\bar{y}$ applied to q visits states from R at least $(m - |R_{dr}| + 1)$ times, where a state is *visited* if it is reached by any nonempty prefix of $\bar{x}/\bar{y}$ applied to q.

From $N^m(q)$ the *traversal set* $T^m(q)$ is constructed as the set containing for every $\bar{x} \in N^m(q)$ all $\bar{x}/\bar{y} \in \mathcal{L}(M, q)$ such there exists some $R \in RD$ whose states are visited exactly $(m - |R_{dr}| + 1)$ times along $\bar{x}/\bar{y}$ applied to q, while this does not hold for any proper prefix of $\bar{x}/\bar{y}$. We call R a *terminating* set of $\bar{x}/\bar{y}$.

Finally, from each $\bar{x} \in N^m(q)$ an acyclic observable FSM $M_{q,\bar{x}}$ with language $\mathcal{L}(M_{q,\bar{x}}) = \{\bar{x}'/\bar{y}' \in T^m(q) \mid \bar{x}' \text{ is a prefix of } \bar{x}\}$ is created such that any two sequences in $M_{q,\bar{x}}$ reach the same state in $M_{q,\bar{x}}$ only if they reach the same state if applied to q in M. This FSM can then be further simplified as described in [21]. The adaptive test case $TC(q, \bar{x})$ denotes the output completion of $M_{q,\bar{x}}$.

Test Suite Generation Given a preamble P_q for every definitely reachable state q of M and for each state q in M a set ID_q containing for each q' in M that is r-distinguishable from q an adaptive test case I_q of a state separator S of M for q and q', a test suite TS is created from an empty set as follows: For each definitely reachable state q of M, each $\bar{x} \in N^m(q)$, each $TC(q, \bar{x})$ and each set $\{\bar{x}_1/\bar{y}_1, \ldots, \bar{x}_k/\bar{y}_k\}$ of all sequences of length $|\bar{x}|$ in $\mathcal{L}(TC(q, \bar{x}))$ that do not reach *fail*, a set of adaptive test cases T of the following form is added to TS, where q_i denotes the state reached by applying $\bar{x}_i/\bar{y}_i$ to q, t_i denotes the deadlock state reached by applying $\bar{x}_i/\bar{y}_i$ to the initial state of $TC(q, \bar{x})$, $TC(P_q)$ is the adaptive test case created from the output completion of P_q, and each $I_{q_i} \in ID_{q_i}$ is only concatenated onto one ATC for each $\bar{x}_i/\bar{y}_i$:

$$T := TC(P_q) @_q TC(q, \bar{x}) @_{t_1} I_{q_1} @_{t_2} I_{q_2} \ldots @_{t_k} I_{q_k}$$

Applying test suite TS to an SUT thus essentially consists of applying the traversal sets after the preambles and applying after each such sequence a set of ATCs created from state separators that r-distinguish the state reached by the sequence from all states it is r-distinguishable from.

For the formalisation and implementation described in the following sections we have chosen a different representation of the test suite and its test cases, but the underlying strategy remains unchanged. We furthermore integrate the optimisation already described by Petrenko and Yevtushenko in [19] that it is not always necessary to distinguish the target of some $\bar{x}/\bar{y} \in T^m(q)$ from all states it is r-distinguishable from, as it is sufficient to r-distinguish the target from other states in some $R \in RD$ only if R is used in the termination criterion of some sequence of which $\bar{x}/\bar{y}$ is a prefix (see the use of $Id(s', R_\beta)$ in [19, Algorithm 2]).

3 The Mechanised Proof

Isabelle/HOL Isabelle is a generic proof assistant featuring an extensive implementation of higher-order logic (Isabelle/HOL). We chosen this logic as the base

for our formalisation, as it is highly expressive and already contains many useful definitions and theorems. For an introduction to Isabelle see Nipkow et al. [15]. The Isabelle core libraries are further extended by the *Archive of Formal Proofs* (see `www.isa-afp.org`). The Isar (*Intelligible Semi-Automated Reasoning*) proof language offered in Isabelle distributions allows for proofs to be written in a human-readable style [23]. Our previous paper [22] contains an exemplary Isar proof on FSM properties. Isabelle is also able to automatically generate executable code from many formalisations written in it, which we use in Section 4 to generate a trustworthy implementation of the formalised test strategy. The correctness of this translation process is proven in [8], which also describes how the code generator can serve to facilitate program and data refinement.

Data Structures In our Isabelle/HOL formalisation we define FSMs as records (type `fsm-impl`) which we then restrict to well-formed FSMs (type `fsm`), where an FSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h)$ is well-formed if its component sets are finite and additionally $q_0 \in Q$ and $h \subseteq Q \times \Sigma_I \times \Sigma_O \times Q$ hold.

record (*'state*, *'input*, *'output*) *fsm-impl* =
  initial :: *'state*
  states :: *'state set*
  inputs :: *'input set*
  outputs :: *'output set*
  transitions :: (*'state* × *'input* × *'output* × *'state*) *set*

typedef (*'state*, *'input*, *'output*) *fsm* =
  { *M* :: (*'state*, *'input*, *'output*) *fsm-impl* . *well-formed-fsm M* }

This follows the classical definition of FSMs more closely than the definition in our previous work [22] and also avoids cluttering proofs with well-formedness assumptions on the employed FSMs. Consider, for example, the following very natural definition of completely specified FSMs in our Isabelle formalisation[3]:

completely-specified M = (∀ *q*∈*states M*.∀ *x*∈*inputs M*.∃ *q' y*.(*q*,*x*,*y*,*q'*)∈*transitions M*)

We represent state preambles and separators simply as FSMs, but we represent test suites as values of a new datatype such that a test suite $TS = (PS, tps, rds, seps)$ consists of (1) a set PS of pairs (q, P_q) for each definitely reachable state q, (2) a map tps from each definitely reachable state to the traversal sequences starting from it, (3) a map rds assigning to each pair $(q, \bar{x}/\bar{y})$, consisting of a definitely reachable state and a traversal sequence starting from it, all states that the target of $\bar{x}/\bar{y}$ needs to be r-distinguished from, and (4) a map $seps$ assigning to each pair of r-distinguishable states (q_1, q_2) at least one state separator S with corresponding deadlock states d_1 and d_2.

This representation also requires a new formulation of the pass relation. We say that M' passes test suite $TS = (PS, tps, rds, seps)$ for M, denoted

[3] Function application in Isabelle is performed in a functional programming style without braces. For example, `f(x,y,z)` is written in Isabelle as `f x y z`.

`passes_test_suite` M TS M' in the Isabelle formalisation, if (1) M' passes $TC(P_q)$ for every $(q, P_q) \in PS$, (2) for every $(q, P_q) \in PS$ and $\bar{x}/\bar{y} \in tps(q)$, every reaction of M' in a state reached by P_q to any prefix of $\bar{x}$ is also prefix of some sequence in $tps(q)$, and (3) for every $(q, P_q) \in PS$, $\bar{x}/\bar{y} \in tps(q)$, $q' \in rds(q, \bar{x}/\bar{y})$ and $(S, d_1, d_2) \in seps(q, q')$, M' in a state reached by P_q followed by $\bar{x}/\bar{y}$ passes the adaptive test case I_q created from S. Note that this different representation and formulation does not affect the result of applying a test suite TS against the SUT: M' passes TS represented as a set of ATCs if and only if `passes_test_suite` M TS M' holds.

3.1 Proof Strategy

The first main goal of our formalisation is to prove that the formalised strategy is complete. That is, an SUT should pass a generated test suite if and only if it is a reduction of its specification. We split this proof into two parts by first formulating a sufficient condition such that any test suite satisfying it is complete, and then proving that the test suites generated by the formalised strategy do satisfy the condition. We will refer to this condition as the *completeness predicate* and describe it and its formulation in Isabelle in Subsection 3.2.

An overview of the steps performed in the overall proof is given in Figure 1: Let TS be a test suite generated by the formalised strategy. By construction (outlined in Subsection 2.1, implemented in Section 4), TS satisfies the completeness predicate and is also finite. All further proofs then only rely on these properties and the assumptions on the structure of M' (observable, completely specified, same inputs as M, $|M'| \leq m$). M' can fail a test suite satisfying the completeness predicate only if a behaviour of M' is observed that is not admissible in M, and hence TS is *sound*: if M' is a reduction of M, then it passes TS. The main effort in establishing completeness thereafter lies in proving TS to be *exhaustive*: if M' is not a reduction of M, then M' must not pass TS. This is realised via a proof by contradiction detailed in Subsection 3.2. Finally, completeness follows from soundness and exhaustiveness and still holds if test suites are reduced to finite prefix-free sets of IO-sequences (see Section 4).

The formalisation in Isabelle is split into several *theory files* covering separate aspects of the proof: First, files `FSM_Impl.thy`, `FSM.thy` and `Product_FSM.thy` provide basic definitions and properties for finite state machines. Next, the files `State_Preamble.thy`, `State_Separator.thy` and `Traversal_Set.thy` respectively introduce the three main components of the test suite and algorithms for their computation. These are then combined within `Test_Suite.thy`, which defines the completeness property and shows that each satisfying test suite is complete. Finally, an algorithm for calculating complete test suites is provided in `Test_Suite_Calculation.thy` and refined in further theory files. Overall, the formalisation effort comprises 18 theory files containing a total of 621 lemmata, which can be verified within 201 seconds using Isabelle2020 on a Ryzen 5 3600 CPU and Ubuntu 18.04. Most of these lemmata prove general properties on FSMs and related algorithms and are not restricted to any specific test strategy, allowing their reuse in future formalisation projects employing FSMs.

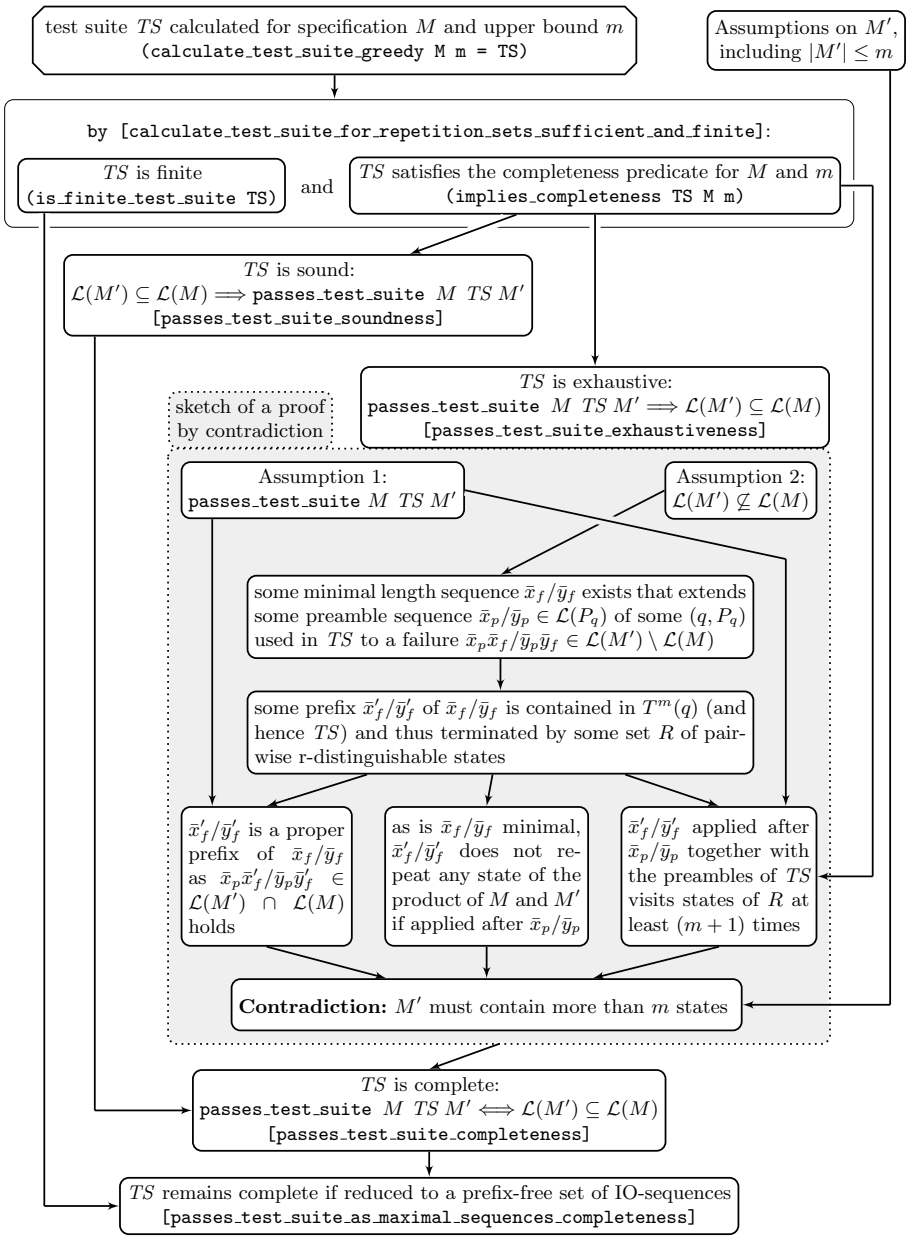

Fig. 1: Overview of the overall proof strategy employed in the mechanised formalisation, with a focus on the proof of exhaustiveness of generated test suites. Names in square brackets indicate lemmata in the Isabelle formalisation. Predicate `implies_completeness` is a shortened version of the completeness predicate described in Section 3.2 for a greedy strategy of calculating RD'.

3.2 A Completeness Predicate for Reduction Testing

The test suite generation strategy outlined in Subsection 2.1 uses the set RD of all maximal sets of pairwise r-distinguishable states of M in its termination criterion for the traversal sets. As enumerating all those sets can be reduced to the problem of listing all maximum cliques of a graph in which two states are connected if they are r-distinguishable, this can prove computationally expensive for larger FSMs. To facilitate algorithms that use only a proper subset of RD, we have formulated the completeness predicate with respect to a test suite $TS = (PS, tps, rds, seps)$, M, the assumed bound $m \geq |M'|$, and a subset RD' of RD. The predicate is given in the Isabelle formalisation as the function `implies_completeness_for_repetition_sets` and is satisfied if the following properties hold: (1) PS contains (q_0, P_0) and thus the trivial empty preamble, (2) for every $(q, P) \in PS$, P is a preamble of q and $tps(q)$ contains the traversal set $T^m(q)$, (3) all $(S, d_1, d_2) \in tps(q_1, q_2)$ are state separators of q_1 and q_2 with corresponding deadlock states, (4) every state q of M is contained in some set in RD', (5) if two traversal sequences $\bar{x}_1/\bar{y}_1 \neq \bar{x}_2/\bar{y}_2$ in $tps(q)$ are each prefix of a sequence in $T^m(q)$ terminated by some $R \in RD'$ and both sequences applied to q reach distinct states in $q_1, q_2 \in R$, then $q_2 \in rds(q, \bar{x}_1/\bar{y}_1)$ and $q_1 \in rds(q, \bar{x}_2/\bar{y}_2)$, (6) if a traversal sequence $\bar{x}/\bar{y} \in tps(q)$ is prefix of a sequence in $T^m(q)$ terminated by some $R \in RD'$ and reaches some $q_1 \in R$ if applied to q, then for any $q_2 \in R_{dr} \setminus \{q_1\}$ it holds that $q_2 \in rds(q, \bar{x}/\bar{y})$ and $q_1 \in rds(q_2, \epsilon)$, and (7) for each $R \in RD'$, if any traversal sequence is terminated by R, then for each pair of distinct states $q_1, q_2 \in R_{dr}$ it holds that $q_2 \in rds(q_1, \epsilon)$ and $q_1 \in rds(q_2, \epsilon)$.

To summarise, TS and RD' are complete for reduction testing with respect to M and m if they are produced by a strategy such as that described in Subsection 2.1, without specifying many restrictions on the actual implementation of such a strategy. Properties (1) to (4) require the test suite to consist of actual preambles, state separators and traversal sets, whereas properties (5) to (7) describe where ATCs need to be applied in order to distinguish states. Note here that these latter three properties structurally closely resemble conditions 1. to 3. in Theorem 1 of [6], which describes the H-method for equivalence testing of FSMs.

Finally observe that the soundness of any test suite TS satisfying the predicate follows directly from the reformulation of the pass relation, as satisfaction requires that application of TS entails only the application of preambles followed by corresponding traversal sets followed by corresponding state separators, and thus only of IO-sequences in $\mathcal{L}(M)$, for which no reduction of M may fail.

Completeness of Test Suites Satisfying the Completeness Predicate Let test suite TS and subset RD' of RD satisfy the completeness predicate with respect to M and an assumed upper bound $m \geq |M'|$. Then the exhaustiveness of TS follows from a classical state counting argument by establishing a lower bound greater than m on the number of states contained in any M' that is not a reduction of M but that passes TS, see for example [9] and [22]. We have used this approach in a proof by contradiction: if we assume that M' passes TS while also $\mathcal{L}(M') \not\subseteq \mathcal{L}(M)$ holds, then some minimal sequence to a failure $\bar{x}_f/\bar{y}_f$

extending some preamble sequence $\bar{x}_p/\bar{y}_p$ of some (q, P_q) in the preambles of TS must exist, which is not applied in TS. Hence, some proper prefix $\bar{x}'_f/\bar{y}'_f$ of $\bar{x}_f/\bar{y}_f$ must be contained in $T^m(q)$ and terminated by some $R \in RD'$. Now $\bar{x}_p\bar{x}'_f/\bar{y}_p\bar{y}'_f$ cannot repeat any state of the product of M and M', as this would allow for a sequence to a failure shorter than $\bar{x}_f/\bar{y}_f$. Furthermore, by construction of the traversal sequences, $\bar{x}'_f/\bar{y}'_f$ applied after $\bar{x}_p/\bar{y}_p$ together with the preambles in TS must visit states of R at least $(m+1)$ times in M. These last two properties, in conjunction with M' passing all applied state separators of conditions (5) to (7) of the completeness predicate, require M' to contain at least $(m+1)$ distinct states, which provides a contradiction to the assumed upper bound $|M'| \leq m$. Completeness of TS follows from exhaustiveness and the previously established soundness. That is, M' passes TS if and only if M' is a reduction of M.

4 A Trustworthy Implementation

Complete Algorithms The calculation of a test suite satisfying the completeness predicate can be separated into five major steps: (1) the calculation of definitely reachable states of M and corresponding preambles, (2) the calculation of r-distinguishable state pairs in M and corresponding state separators, (3) the calculation of a subset RD' of RD, (4) the calculation of traversal sets from each definitely reachable state, also storing the corresponding terminating $R \in RD'$, and (5) the combination of these results into a test suite by calculating for each definitely reachable state and each traversal sequence from this state the states it must be r-distinguished from. Figure 2 depicts these steps and their corresponding functions in the Isabelle formalisation within the box for function `generate_test_suite_greedy`. This function implements the test strategy described above using a simple greedy algorithm to calculate a set RD'.

Step (1) of the above calculation is realised in the formalisation using function `calculate_state_preamble_from_input_choices` to construct a state preamble for a given state q if it exists, based directly on Algorithm 1 of [19]. It creates a preamble by starting from q and analysing *backward reachability*, iteratively adding not yet backwardly reached nodes q' and inputs x such that all transitions from q' for x in M reach nodes added in previous iterations, including q. If this process adds the initial state of M, then the selected states and corresponding inputs induce a valid preamble for q. State separators are calculated in a similar way to establish step (2), starting from the deadlock states of a partial separator, again as described in [19]. Based on these results, we have formalised two possible implementations for step (3): first by naively enumerating all elements of RD and second by a greedy algorithm that calculates for each state q a set $R_q \in RD$ by starting from $R_q := \{q\}$ and iteratively adding states that are r-distinguishable from all states currently in R_q until R_q is maximal. Next, we realise step (4) by performing a straightforward enumeration of paths from each definitely reachable state until the corresponding traversal sequence can be terminated by some $R \in RD'$. Finally, the implementation of step (5) follows above description.

Test Suites as Sets of IO-Sequences To facilitate easier integration with other tools and storage of calculated test suites, we additionally provide algorithms to reduce test suites to finite prefix-free sets of IO-sequences, which might be stored by data structures as simple as lists. The previously introduced Function `generate_test_suite_greedy` performs this transformation as a final step by extracting from the calculated test suite all IO-sequences $\bar{x}_p\bar{x}_t\bar{x}_r/\bar{y}_p\bar{y}_t\bar{y}_r$ such that $\bar{x}_p/\bar{y}_p$ is a sequence reaching q in some preamble P_q, (q, P_q) is contained in TS, $\bar{x}_t/\bar{y}_t$ is a traversal sequence in $T^m(q)$ and $\bar{x}_r/\bar{y}_r$ is a sequence in some state separator applied thereafter that does not reach *fail*. The resulting finite set of IO-sequences is then simplified by removing all sequences that are proper prefixes of other contained sequences.

Let TS be the calculated test suite represented as defined in Section 3. Then, by construction and depicted as the last step of Figure 1, the reduction of TS to a finite set (or list) TS_L of IO-sequences satisfies the following property:

$$\begin{aligned}\mathcal{L}(M') \subseteq \mathcal{L}(M) &\Leftrightarrow \texttt{passes\_test\_suite}\ M\ TS\ M' \\ &\Leftrightarrow \forall \bar{x}x\bar{x}'/\bar{y}y\bar{y}' \in TS_L : \forall \bar{x}x/\bar{y}y' \in \mathcal{L}(M') : \exists \bar{x}x\bar{x}''/\bar{y}y'\bar{y}'' \in TS_L\end{aligned}$$

That is, an SUT representation M' passes TS if and only if for any $\bar{x}x/\bar{y}y$ that is prefix of any sequence in TS_L the SUT reacts to x applied after $\bar{x}/\bar{y}$ only with outputs y' such that $\bar{x}x/\bar{y}y'$ is also prefix of some sequence in TS_L. Note here that M is not referenced in the bottom right side of the bi-implications. Test suite TS_L can thus be applied to an SUT in practice by *applying* to the latter each contained IO-sequence (*test case*) at least k times (to satisfy the complete testing assumption). A test case $x_1 \dots x_n/y_1 \dots y_n$ is *applied* by iteratively applying each x_i (beginning at x_1) and observing the corresponding response y_i' of the SUT, continuing only if $y_i = y_i'$. The SUT *passes* the application of $x_1 \dots x_n/y_1 \dots y_n$ if and only if the observed SUT response $x_1 \dots x_j/y_1' \dots y_j'$ is prefix of some sequence in TS_L. Finally, by the above property TS_L must be a subset of $\mathcal{L}(M)$ and hence the SUT passes a test case if and only if the observed response of the SUT to the test case is be admissible in M, as $\mathcal{L}(M)$ is prefix closed.

Refinement and Extensibility Isabelle provides a powerful refinement mechanism to generate more efficient code from definitions, which we have employed in theory file `Test_Suite_Calculation_Refined.thy` to refine both data structures and algorithms. First, we have used the Containers framework [13] to represent sets using data structures such as red-black-trees wherever possible, which improves on Isabelles default set implementation as lists. Furthermore, we provide several *code equations* that allow the code generator to replace function definitions by (provably equivalent) definitions that are more efficient, for example by extracting common subexpressions to avoid repeated evaluation of identical expressions. These refinements have allowed us to use simple definitions that are easy to use within the proofs, while still being able to use more efficient definitions in code generation. They are also extensible as it is possible to add further code equations and overwrite existing ones.

New variations or optimisations of the test strategy described in this paper can furthermore easily be proven complete by establishing that they satisfy the completeness predicate, as we have decoupled implementation details from the proof of completeness via this criterion. To prove complete, for example, an alternative test strategy resulting from replacing the currently used algorithm for calculating state separators by one of the algorithms presented in [7], it is sufficient to prove in Isabelle that this new algorithm generates valid state separators and to provide for the new test strategy a lemma analogous to lemma `calculate_test_suite_for_repetition_sets_sufficient_and_finite`, which proves satisfaction of the completeness predicate.

Generated Test Tools In theory file `Test_Generator_Export_Code.thy` we generate Haskell code for function `generate_test_suite_greedy`, which we then use to implement a comprehensive test tool set able to test an SUT given as a function in C against a specification FSM. This tool set is available at the provided online resources (see Section 1) and consists of two parts: a test suite generator which uses the code generated from Isabelle to generate a test suite for a given specification FSM, and a test harness which applies each test case in a given test suite against an SUT, using the above definition of applying test cases, to calculate a verdict on whether the SUT conforms to the specification. This verdict is `PASS` if and only if all test cases pass. Figure 2 depicts the workflow.

The two tools each contain fewer than 200 lines of code in addition to the generated code and serve only as interfaces between the specification FSM, test suites and the SUT. They can thus be verified manually with little effort. Together with the trustworthiness of the provably correct generated code (see [8]) facilitating the test suite generator, this has important implications on tool qualification efforts of test tools that employ the above tools: First, the fault detection capabilities of each generated test suite are established by mechanised proofs within our formalisation, providing evidence that it is complete for the fault domain of SUTs whose behaviour can be represented by an FSM M' that is completely specified, observable, contains at most m states, and has the same nonempty set of inputs as M. Verification of these proofs themselves reduces to verifying the used definitions, as any proof is verified automatically by the small trustworthy inference kernel of Isabelle, even when using sophisticated proof methods (see [2]). Furthermore, as generated test suites are a proper subsets of $\mathcal{L}(M)$, no test case application following the previously described procedure can introduce undetected SUT failures by reporting some test case as passed even though the observed response of the SUT to it is not admissible in the specification. Thus, by verification of the test case application mechanism in the test harness, the first of two hazards introduced by model-based test tools as identified in [3] can be mitigated. Mitigation of the second hazard identified there, undetected coverage failures due to test executions failing to meet test case specific pre-conditions, can for our fault domain be reduced to verifying that the test harness resets the SUT to an initial state before applying each test case, as this is the only pre-condition we require. Since the integration of the SUT into

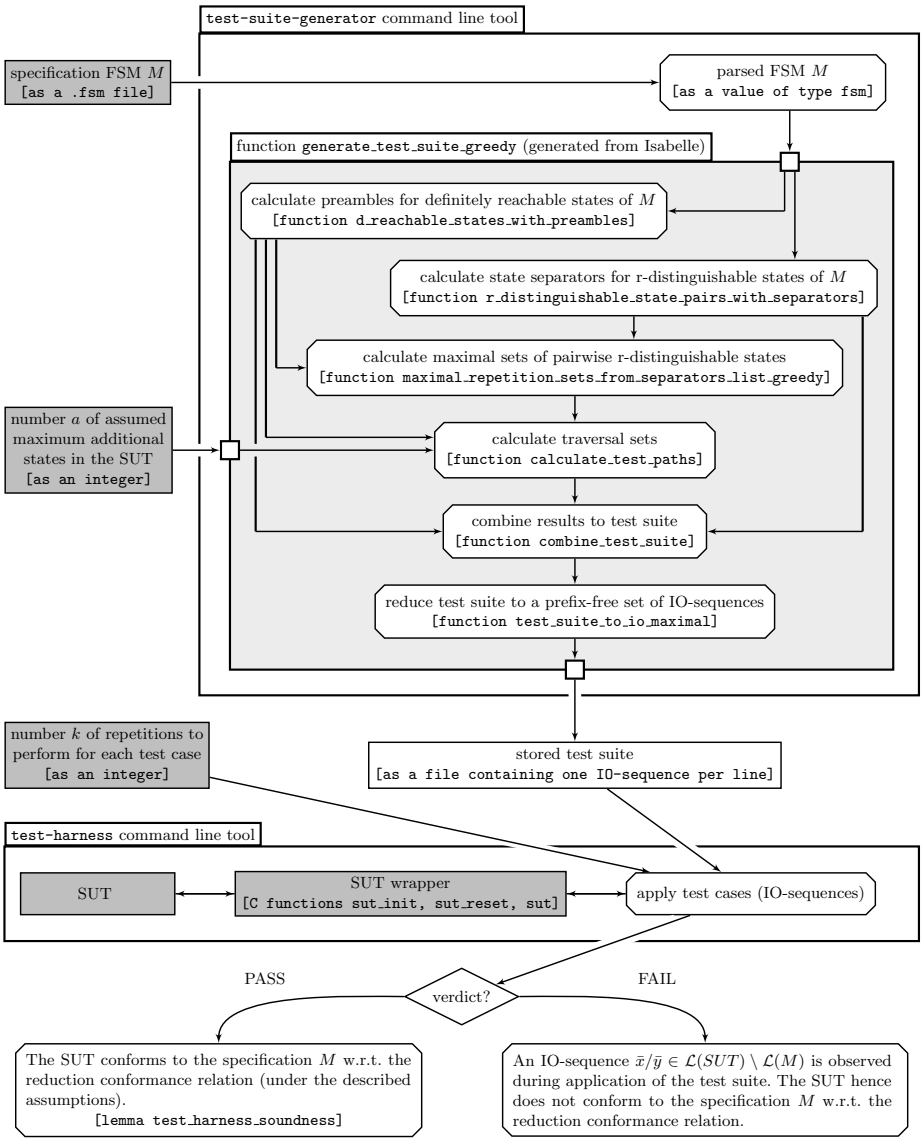

Fig. 2: Overview of the workflow of using the test suite generator and test harness command line tools to test an SUT. Dark grey shadings indicate inputs and SUT integration provided by the user, while light grey shadings indicate the trustworthy code generated from the Isabelle formalisation.

the test harness is highly dependent on the SUT, we have not included the test harness in the Isabelle formalisation, but we believe that the very small provided implementation can easily be verified for a given SUT. Such a verification of the test harness tool and the integration of the generated code into the test suite generator tool then obviates measures such as replay of observed behaviours against the specification (see [3]) arising from untrusted test case generators.

Statistical Experiments We have applied the generated implementation on a synthetic data set containing randomly generated FSMs of varying size and extend of nondeterminism, where we define the *degree of nondeterminism* of an observable FSM $M = (Q, q_0, \Sigma_I, \Sigma_O, h)$, denoted $d_{nondet}(M)$, as the ratio between the number of transitions such that other transition with the same source and input exist, and the overall number $|h|$ of transitions in M:

$$d_{nondet}(M) = |\{(q, x, y, q') \in h \mid \exists y', q'' : (q, x, y', q'') \in h \wedge y \neq y'\}| \ / \ |h|$$

The synthetic data set contains for each configuration (s, d) with $4 \leq s \leq 20$ and $d \in \{0.1, 0.2, 0.3, 0.4\}$ a collection of 1000 randomly generated FSMs M with 6 inputs and 4 outputs such that $|M| = s$ and M has been generated for a target value of $d_{nondet}(M) = d$ using the *fsmlib-cpp* library[4], an open source project containing many fundamental algorithms for processing FSMs. We have restricted the range of values for d to comparatively small values, as we believe that specifications of safety-critical systems often exhibit nondeterminism only to a very limited extend.

Figure 3 shows the average size (number of test cases) of test suites calculated for this data set, using an upper bound $m = |M|$, and the average time required for each calculation as measured on a Ryzen 5 3600 CPU calculating test suites for 8 FSMs in parallel. These results indicate that, for a fixed input alphabet, the average size of generated test suites correlates with the extend of nondeterminism in the specification. This follows in particular from the definition of traversal sets, since a higher degree of nondeterminism indicates, on average, a higher number of outgoing transitions for each state and thus an increase in the number of distinct IO sequences originating at each state. Note also that the calculated test suites are significantly smaller than those created using the "brute force" strategy of enumerating all input sequences of length $|M| \cdot m$, resulting in test suites of size at least $|\Sigma_I|^{m \cdot |M|}$. We conjecture that the execution time of the generated implementation can be drastically reduced by further refinement and the employment of more sophisticated algorithms in particular for the calculation of state separators.

The data set, instructions on how to employ the generated implementation to calculate test suites, and detailed results for each FSM contained in the data set are available as part of the online resources referenced in Section 1.

[4] Publicly available for download at `https://github.com/agbs-uni-bremen/fsmlib-cpp`. Random FSMs for a given degree of nondeterminism have been constructed using method `createRandomFsm` of class `Fsm`.

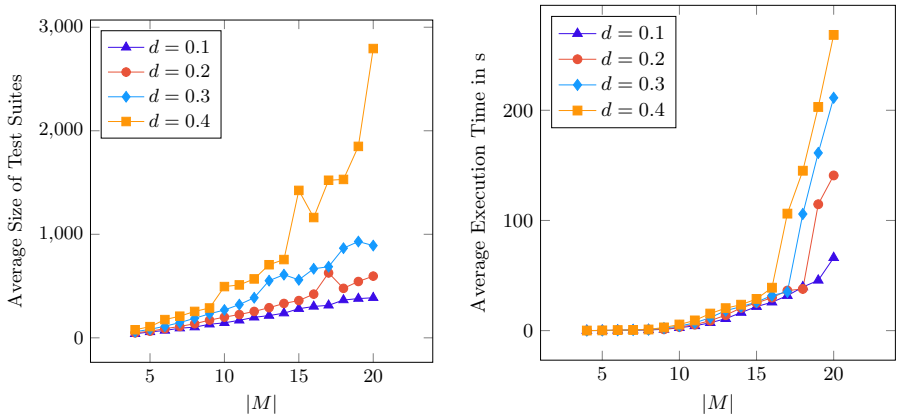

Fig. 3: Average size (left) and calculation time in s (right) of test suites for randomly generated, completely specified, observable and minimised specification FSMs M for $m = |M|$, depending on $|M|$ and target value d for $d_{nondet}(M)$.

5 Conclusions and Future Work

We have provided the first comprehensive mechanised proof of the test strategy elaborated by Petrenko and Yevtushenko in [21] and established the correctness of an implementation of this strategy. As a second main contribution, we have used the mechanised formalisation to generate from it trustworthy executable code and embedded it into a set of test tools that facilitate the calculation and application of test suites. Further investigations are required to quantitatively compare this generated implementation against hand-crafted implementations of the formalised test strategy. The theories and proofs of the formalisation, as well as the implementation of the test strategy itself, have been developed using the Isabelle/HOL tool, indicating the suitability of the latter to perform such undertakings with acceptable effort.

We advocate the use of mechanised proofs of the completeness of test strategies and the use of provably correct automatically generated implementations, because such strategies and guarantees of their fault detection capabilities are of considerable value in model-based testing of safety-critical systems, where undiscovered flaws in test strategies or their implementations might lead to insufficient actual test strength and therefore to the possibility of fatal errors in the system under test being left undiscovered. The use of provably correct automatically generated implementations can furthermore mitigate hazards introduced by model-based test tools and obviate measures against untrusted test case generators, simplifying tool qualification efforts of test tools that employ such implementations.

Acknowledgements I would like to thank my doctoral thesis supervisor Jan Peleska for several helpful discussions.

References

1. Bjørner, N.: Z3 and SMT in industrial R&D. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E.P. (eds.) Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10951, pp. 675–678. Springer (2018). https://doi.org/10.1007/978-3-319-95582-7_44, https://doi.org/10.1007/978-3-319-95582-7_44
2. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic proof and disproof in isabelle/hol. In: International Symposium on Frontiers of Combining Systems. pp. 12–27. Springer (2011)
3. Brauer, J., Peleska, J., Schulze, U.: Efficient and trustworthy tool qualification for model-based testing tools. In: Nielsen, B., Weise, C. (eds.) Testing Software and Systems. Proceedings of the 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 2012. pp. 8–23. No. 7641 in Lecture Notes in Computer Science, Springer, Heidelberg Dordrecht London New York (2012)
4. Brucker, A.D., Wolff, B.: Interactive testing with HOL-TestGen. In: Grieskamp, W., Weise, C. (eds.) Formal Approaches to Software Testing, 5th International Workshop, FATES 2005, Edinburgh, UK, July 11, 2005, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3997, pp. 87–102. Springer (2005). https://doi.org/10.1007/11759744_7, https://doi.org/10.1007/11759744_7
5. Brucker, A.D., Wolff, B.: On theorem prover-based testing. Formal Asp. Comput. **25**(5), 683–721 (2013). https://doi.org/10.1007/s00165-012-0222-y, https://doi.org/10.1007/s00165-012-0222-y
6. Dorofeeva, R., El-Fakih, K., Yevtushenko, N.: An improved conformance testing method. In: Wang, F. (ed.) Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3731, pp. 204–218. Springer (2005). https://doi.org/10.1007/11562436_16, https://doi.org/10.1007/11562436_16
7. El-Fakih, K., Yevtushenko, N., Saleh, A.: Incremental and heuristic approaches for deriving adaptive distinguishing test cases for nondeterministic finite-state machines. Comput. J. **62**(5), 757–768 (2019). https://doi.org/10.1093/comjnl/bxy086, https://doi.org/10.1093/comjnl/bxy086
8. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: International Symposium on Functional and Logic Programming. pp. 103–117. Springer (2010)
9. Hierons, R.M.: Testing from a nondeterministic finite state machine using adaptive state counting. IEEE Trans. Computers **53**(10), 1330–1342 (2004). https://doi.org/10.1109/TC.2004.85, http://doi.ieeecomputersociety.org/10.1109/TC.2004.85
10. Hierons, R.M.: FSM quasi-equivalence testing via reduction and observing absences. Sci. Comput. Program. **177**, 1–18 (2019). https://doi.org/10.1016/j.scico.2019.03.004, https://doi.org/10.1016/j.scico.2019.03.004
11. Huang, W., Peleska, J.: Complete model-based equivalence class testing. Software Tools for Technology Transfer **18**(3), 265–283 (2016). https://doi.org/10.1007/s10009-014-0356-8, http://dx.doi.org/10.1007/s10009-014-0356-8

12. Huang, W.l., Peleska, J.: Complete model-based equivalence class testing for nondeterministic systems. Formal Aspects of Computing **29**(2), 335–364 (2017). https://doi.org/10.1007/s00165-016-0402-2, http://dx.doi.org/10.1007/s00165-016-0402-2
13. Lochbihler, A.: Light-weight containers for isabelle: efficient, extensible, nestable. In: International Conference on Interactive Theorem Proving. pp. 116–132. Springer (2013)
14. Luo, G., von Bochmann, G., Petrenko, A.: Test selection based on communicating nondeterministic finite-state machines using a generalized wp-method. IEEE Trans. Software Eng. **20**(2), 149–162 (1994). https://doi.org/10.1109/32.265636, http://doi.ieeecomputersociety.org/10.1109/32.265636
15. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9, https://doi.org/10.1007/3-540-45949-9
16. Peleska, J., Brauer, J., Huang, W.: Model-based testing for avionic systems proven benefits and further challenges. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 11247, pp. 82–103. Springer (2018). https://doi.org/10.1007/978-3-030-03427-6_11, https://doi.org/10.1007/978-3-030-03427-6_11
17. Peleska, J., Huang, W.l.: Test Automation - Foundations and Applications of Model-based Testing. University of Bremen (January 2019), lecture notes, available under http://www.informatik.uni-bremen.de/agbs/jp/papers/test-automation-huang-peleska.pdf
18. Peleska, J., Vorobev, E., Lapschies, F.: Automated test case generation with SMT-solving and abstract interpretation. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) Nasa Formal Methods, Third International Symposium, NFM 2011. LNCS, vol. 6617, pp. 298–312. Springer, Pasadena, CA, USA (April 2011)
19. Petrenko, A., Yevtushenko, N.: Adaptive testing of deterministic implementations specified by nondeterministic FSMs. In: Testing Software and Systems. pp. 162–178. No. 7019 in Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2011)
20. Petrenko, A., Yevtushenko, N., Bochmann, G.V.: Testing deterministic implementations from nondeterministic FSM specifications. In: Testing of Communicating Systems, IFIP TC6 9th International Workshop on Testing of Communicating Systems. pp. 125–141. Chapman and Hall (1996)
21. Petrenko, A., Yevtushenko, N.: Adaptive testing of nondeterministic systems with FSM. In: 15th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2014, Miami Beach, FL, USA, January 9-11, 2014. pp. 224–228. IEEE Computer Society (2014). https://doi.org/10.1109/HASE.2014.39, http://dx.doi.org/10.1109/HASE.2014.39
22. Sachtleben, R., Hierons, R.M., Huang, W.l., Peleska, J.: A mechanised proof of an adaptive state counting algorithm. In: Gaston, C., Kosmatov, N., Le Gall, P. (eds.) Testing Software and Systems. pp. 176–193. Springer International Publishing, Cham (2019)
23. Wenzel, M.: Isabelle, Isar - a versatile environment for human readable formal proof documents. Ph.D. thesis, Technical University Munich, Germany (2002), http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.pdf