SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1593

Behaviour abstraction adequacy criteria for API call protocol testing

Hernan Czemerinski[1,*,†], Victor Braberman[1] and Sebastian Uchitel[1,2]

[1]*Departamento de Computación, FCEyN, Universidad de Buenos Aires, Buenos Aires, Argentina*
[2]*Department of Computing, Imperial College London,London, UK*

SUMMARY

Code artefacts that have non-trivial requirements with respect to the ordering in which their methods or procedures ought to be called are common and appear, for instance, in the form of API implementations and objects. Testing such code artefacts to gain confidence that they conform to their intended *protocols* is an important and challenging problem. This paper proposes conformance testing adequacy criteria based on covering an abstraction of the intended behaviour's semantics. Thus, the criteria are independent of the specification language and structure used to describe the intended protocol and the language used to implement it. As a consequence, the results may be of use to black box conformance testing approaches in general. Experimental results show that the criteria are a good predictor for fault detection for protocol conformance and for classical structural coverage criteria such as statement and branch coverage. They also show that the division of the domain derived from the criterion produces subdomains such that most of its inputs are fault revealing. Copyright © 2015 John Wiley & Sons, Ltd.

Received 8 May 2014; Revised 18 September 2015; Accepted 20 September 2015

KEY WORDS: software testing; API call protocol; adequacy criteria

1. INTRODUCTION

Despite the progress made, the automatic generation of efficient high-quality test suites is still a major challenge for many kinds of software [1–3]. This is the case for stateful components such as APIs, GUI, web software, protocol servers and clients that have non-trivial requirements with respect to the ordering in which their methods or procedures ought to be called to produce meaningful results or to access certain functionality [4]. For instance, in Java, the interface `ResultSet` is used to access or modify elements from a database. The call protocol for a `ResultSet` instance prohibits cursor movements while a new record is being inserted (i.e. no cursor movement between `moveToInsertRow` and `insertRow`) but allows closing the `ResultSet` even if insertion has not been completed. Another example of a stateful component is some web-servers: although they must be able to receive requests in any order, only by following a specific sequence of actions, it is possible to trigger portions of its functionality. Components with non-trivial call protocols and options are particularly challenging for testing approaches [2, 5]. It is known that they are hard for both random testing and related approaches [5] and dynamic symbolic execution techniques [2].

A particularly important type of testing for stateful components is performed to gain confidence that they conform to their intended *call protocols* [6]. For instance, call protocol conformance is crucial to gain assurance that client code abiding to intended usage will not fail due to making calls on code that poorly implements the intended call protocol (as the dual problem of typestate verifi-

*Correspondence to: Hernan Czemerinski, Departamento de Computación, Pabellón 1, Ciudad Universitaria, Intendente Güiraldes 2160 (C1428EGA). Ciudad Autónoma de Buenos Aires, Argentina.
†E-mail: hczemeri@dc.uba.ar

cation [7]). Thus, call protocol conformance underlies settings like model-based development [8], model-based testing [9] and test-driven development [10]. In this context, testing focus is on verifying that the code under test accepts or rejects sequences of method calls according to the intended call protocol. This is intractable as in principle it may consist in checking for a potentially infinite set of sequences of method calls that the implementation accepts or rejects, each one according to the call protocol.

White box adequacy criteria (i.e. adequacy criteria based on code) implies that the same test suite developed at design time may be adequate for some implementations but not for others. Such a situation is undesirable in, for instance, cross platform development of API's, which is becoming widespread in product family development. The question addressed in this paper is if it is possible to define adequacy criteria for testing call protocols that are independent of the structure of the code to be tested. That is, whether adequacy criteria can be defined in terms of the intended call protocol rather than its implementation.

Black box testing has addressed this problem to some extent. The vast majority of work on black box testing has studied structural strategies for defining adequacy of conformance with respect to specifications [3]. Coverage criteria are then defined either in terms of structural elements of the specification [11–15] or the executable code generated from it [16, 17]. This yields criteria and empirical studies influenced by (accidental) elements of the structure of the model or its executable code such as predicates, control or data flow elements. Authors have already warned that accidental aspects of specifications or model compilers may potentially influence effectiveness of criteria [16–21]. Moreover, being tightly coupled to a particular language, the relation between the criterion and protocol state space actually covered and, consequently, the degree to which the failure domain is explored, is not studied. This in turn hinders the generalization of the empirical results of this body of work [3] to the general problem of black box testing of call protocols.

Thus, although some sort of semantic coverage would be expected as a natural measure of testing, there is a hitherto unexplored difficulty when the behaviour of the system under test is infinite. One of the main challenges of defining effective semantic or behaviour-based coverage criteria is that call protocols are typically infinite state, and consequently, traditional black box criteria for conformance testing of call protocols are not applicable as finite state spaces are assumed [11, 12, 20, 22–25]. Various strategies for finitizing protocol state spaces have been studied [26–28]. However, there are no empirical results on their effectiveness.

The general hypothesis of this paper is that effective notions of behaviour coverage are actually feasible by defining them in terms of finite abstractions of the semantic domain that describes the intended call protocol behaviour. Two coverage criteria over a finite abstraction of deterministic infinite state behaviour call protocols are proposed together with experiments that show two things: (i) that when faults manifest themselves as unexpected exceptions or non-termination, the criteria are good predictors of fault detection in the context of call protocol conformance testing and (ii) that the criteria are good predictors of structural coverage. The practical implications of this result may be that in the context of development approaches which advocate test development before coding, generating tests according to an abstraction of the call protocol semantics of an artefact with non-trivial requirements on method call ordering would provide (i) a good criterion for detecting faults and (ii) a first and early shot at producing high code coverage test suites.

The coverage criteria are defined over enabledness-preserving abstractions (EPAs) [29]. These abstractions quotient an infinite state space into finite classes of states, which allow the same method calls. They also abstract method parameters through existential elimination. The transitions between EPA states (corresponding to action invocations that may lead the program from one state to another) are the key of the first proposed criteria: the more transitions are covered by a test suite, the higher the level of adequacy according to the criteria.

The fault detection ability of EPA transition coverage is evaluated on five industrially relevant Java classes with rich call protocols by analysing the mutant detection capability of randomly generated test suites. Questions studied are (*RQ.1*) how good EPA transition coverage is for fault detection by looking at (*RQ.1.1*) EPA transition coverage and fault detection correlation, and (*RQ.1.2*), as achieving higher coverage is related to test suite size, the impact of EPA transition coverage over fixed-size test suites. Then, a more qualitative study is performed (*RQ.2*) aiming to provide insight

DOI: 10.1002/stvr

on why the positive results for RQ.1 are obtained. The paper includes a study on the correlation between EPA transition coverage and code coverage (*RQ.2.1*) on the category partition implicitly defined by EPA transition coverage (an input i belongs to the subdomain defined by a transition t if and only if i exercises t) is likely to have dense subdomains (*RQ.2.2*) and the effectiveness of individual EPA transitions (*RQ.2.3*). It is also investigated (*RQ.3*) if the results for RQ.1 and RQ.2 are not simply an artefact of achieving action coverage. Finally, a combined criterion that involves covering EPA transitions and actions (*RQ.4*) is proposed and studied.

Subject to the threats derived from the adopted fault and failure model and the assumption of deterministic specifications and implementations (which are further discussed in Section 5), results suggest that the EPA transition criterion is a good predictor of mutant detection (RQ.1.1). Results also show that EPA transition correlations are comparable with or better than those of structural (rather than behavioural) white box criteria. They also suggest that the EPA transition coverage criterion, which is behavioural and hence independent of specification language bias, performs comparably in terms of predictability of test suite fault detection, resulting in higher values for all case studies against statement coverage and is better than branch coverage in two out of five case studies. Moreover, for fixed size, test suites with the highest behavioural adequacy are statistically better (RQ.1.2) in terms of fault detection.

Results for RQ.2 aiming at attaining a deeper understanding of the observed phenomena reinforce the results of RQ.1. Results indicate that the proposed criterion is a good predictor of structural coverage criteria (statement and branch coverage) when applied over code under test (RQ.2.1). These results indicate that a first and early shot at producing high code coverage test suites (which could be extended when code is available) can be achieved through EPA transition coverage.

Results also suggest that the domain partition implicitly derived from EPA transitions is likely to produce subdomains that are dense in failures, that is, a subdomain with high failure rate (RQ.2.2). This is close to what is known to be optimal in the context of partition testing [30].

The finer-grained study of RQ.2.3 that investigates the effectiveness of transitions (the likelihood of revealing a mutant when traversing the transition) rather than the subdomains defined by them reveals that for each fault, there is almost always one transition that is highly effective in detecting it ($> 90\%$ effectiveness), while nearly all the rest of the transitions have poor effectiveness ($< 10\%$). Furthermore, experimental results show that the effective transition is not always the same one, it depends on the fault. This is in line with previous results: as faults are a priori unknown, it makes sense to consider more adequate those tests that cover more transitions. Test suites with highest adequacy cover all transitions, and consequently, the one highly effective is exercised. However, a more significant implication is that, as uniqueness of effective subdomain per fault does not hold but does for transitions, there is an opportunity to improve on EPA transition coverage criterion.

Experimentation regarding RQ.3 aimed at investigating that promising results for EPA transition coverage are not simply because the criterion is a refinement of an action coverage criterion. Results suggest that for correlation with fault detection and structural coverage, in most cases, EPA transition coverage performs comparably or better than action coverage. Also, results indicate that subdomains defined by actions are less dense than those derived from transitions and that transitions are, by far, much more effective than actions for exposing faults. These results suggest that EPA transition coverage can provide benefits over action coverage and justify studying a combined criterion.

The last set of results (corresponding to RQ.4) shows that considering both actions and transitions for measuring the adequacy level of test suites yields to results that outperform those of actions and transitions (when considered in isolation) as a fault detection predictor.

In conclusion, this paper is a step to understanding how behavioural coverage of call protocol behaviour correlates with fault detection in the context of protocol conformance testing. Results suggest, to the extent of the external validity threats of our experiments, that the proposed criteria are good predictors for fault detection and for classical structural coverage criteria such as statement and branch coverage.

The implication being that the criterion could help improve random testing, test-driven development, test case selection and, in general, techniques for tests generation from formal specifications when applied to API code with rich call protocols. The results seem to indicate that such approaches would benefit from introducing heuristics that aim to maximize action and EPA coverage.

DOI: 10.1002/stvr

The rest of this paper is organized as follows. It begins by formally defining the addressed problem, formalizing a conformance relation (Section 2.1), the EPA transition coverage criterion (Section 2.3) and research questions (Section 2.4). Then, in Sections 3 and 4, the experimental design and results are presented, followed by related work (Section 6) and conclusions and future work (Section 7).

2. PROBLEM STATEMENT

The paper proposes adequacy criteria based on behavioural coverage. In this section, the problem is formalized by defining what is meant by an intended protocol to be provided by a code artefact, the actual protocol implemented by a code artefact and the conformance relation that is expected to hold between them. Finally, the EPA transition test adequacy criterion is defined, and research questions are formulated.

2.1. Call protocol conformance

The term *protocol conformance* has been extensively used, encompassing many different approaches related to checking if an implementation is in compliance with a specification. In this work, a more specific term is used: API call protocol conformance. As in the case of typestate verification literature [4, 31–34], the verification purpose here is interoperability preservation: an API implementation must support sequences of calls that a specification (potentially used to build or verify client code) defines as legal. As in typestate verification, our setting is restricted to deterministic specifications and to abstracting away output. Determinism in specifications is a natural assumption that provides API clients with certainties about when it is legal to call a given operation. On the other hand, assuming deterministic behaviour of failures in API implementations is a reasonable assumption in many contexts that greatly simplifies presentation and experimentation as further explained in the sequel.

Full formal treatment of programming language semantics is beyond the scope of this paper. An intuitive definition is provided, sufficient for defining rigorously the notion of call protocol conformance used in this work. The semantics of an API implementation can be defined as a call protocol labelled transition system (LTS). The states of the LTS are all configurations of the internal state of the code. If the code is a class, then configurations correspond to all structurally distinct instances of the class. If the code is an API implementation, configurations are all possible valuations on internal variables of the API. Transitions are the effect of successful invocations of specific methods with concrete parameters. A transition between states s and s' will be present if and only if the execution of the associated method – with the annotated actual parameters – on the configuration corresponding to s eventually halts, does not yield any exceptions and changes the internal state of the code to a configuration that corresponds to s'.

A specification language designed to describe the intended call protocol behaviour of a class or API to be developed can be given semantics in a similar fashion. The *intended call protocol LTS* defines which are the (potentially infinite) set of valid method invocation sequences on a code artefact (each invocation including actual parameters). An implementation is conformant if it accepts the sequences of method invocations that are legal according to the intended protocol.

Hence, this paper adopts (intended and actual) call protocol LTS as the semantic domain for implementations and specifications. The actual protocol LTS represents the real behaviour of the implementation, while the intended protocol LTS represents the intended behaviour according to some specification. Both LTS are semantic representations and independent of the programming and specification languages used.

Definition 1 (call protocol LTS)
Let $m_1, \ldots, m_n$ be method names, and $\mathcal{D}_i$ the domain of m_i. An LTS protocol for $m_1, \ldots, m_n$ is a tuple $L = \langle \Sigma, \mathbb{S}, \mathbb{S}_0, \Delta \rangle$ where

- $\mathbb{S}$ is the (possibly *infinite*) set of states.
- $\mathbb{S}_0$ is the initial state.
- $\Sigma = \bigcup_{i \leqslant n} (\{m_i\} \times \mathcal{D}_i)$ is the set of possible method invocations.

DOI: 10.1002/stvr

- $\Delta : (\mathbb{S} \times \Sigma \times \mathbb{S})$ is the transition relation that maps pairs of a state and method invocation to the corresponding resulting state. The relation must be a partial function on the first two elements of the tuple.

As said, no assumptions are made on the way the *intended protocol LTS of a code artefact is described: it is simply assumed that the semantics of such language can be defined in terms of a call protocol LTS as defined subsequently*. In fact, there are several ways an *intended protocol LTS* can be defined in practice: it could be formally given as a model in a Model-Based Testing (MBT) setting, it could be defined by a reference implementation, it could be given as a set of known valid traces, it could be mined from client applications of the code and so on. In any case, an LTS is a reasonable representation of the underlying behaviour. Note that, as mentioned, the protocol LTS is required to be deterministic.

The expression $s \rightarrow^{m_i(p)} s'$ denotes that $(s, m_i(p), s') \in \Delta, s \rightarrow^{m_i(p)}$ denotes $\exists s'.(s, m_i(p), s') \in \Delta$, and $s \not\rightarrow^{m_i(p)}$ denotes $\nexists s'.(s, m_i(p), s') \in \Delta$. These definitions are trivially extended to sequences of method invocations. Conformance between call protocol LTS is defined as an inclusion with respect to the sequences of method invocations they accept.

Definition 2 (call protocol LTS conformance)
Given an intended and an actual call protocol LTS, I and A, over the same set of methods with initial states $\mathbb{S}_{\mathbb{I}_0}$ and $\mathbb{S}_{\mathbb{A}_0}$, A is *in conformance to* I if and only if for all sequence w of method invocations – with concrete parameters –, $\mathbb{S}_{\mathbb{I}_0} \rightarrow_I^w$ then $\mathbb{S}_{\mathbb{A}_0} \rightarrow_A^w$.

The *failure model* adopted is derived from the definitions of intended and actual protocol LTS and the definition of conformance. A *failure* is then a sequence of method invocations with concrete parameters, which is part of the intended protocol but does not terminate or raise an exception when executed on the implementation.

Definition 3 (Failure)
Given an intended and an actual Protocol LTS, I and A, over the same set of methods with initial states $\mathbb{S}_{\mathbb{I}_0}$ and $\mathbb{S}_{\mathbb{A}_0}$, a sequence w of method invocations – with concrete parameters – is a failure if $\mathbb{S}_{\mathbb{I}_0} \rightarrow_I^w$ and $\mathbb{S}_{\mathbb{A}_0} \not\rightarrow_A^w$.

Figure 1(a) shows a portion of the intended call protocol LTS of a bounded stack over alphabet $\{a, b\}$. Initially the stack is empty, and thus, the only possible action in S_0 is to push an element into

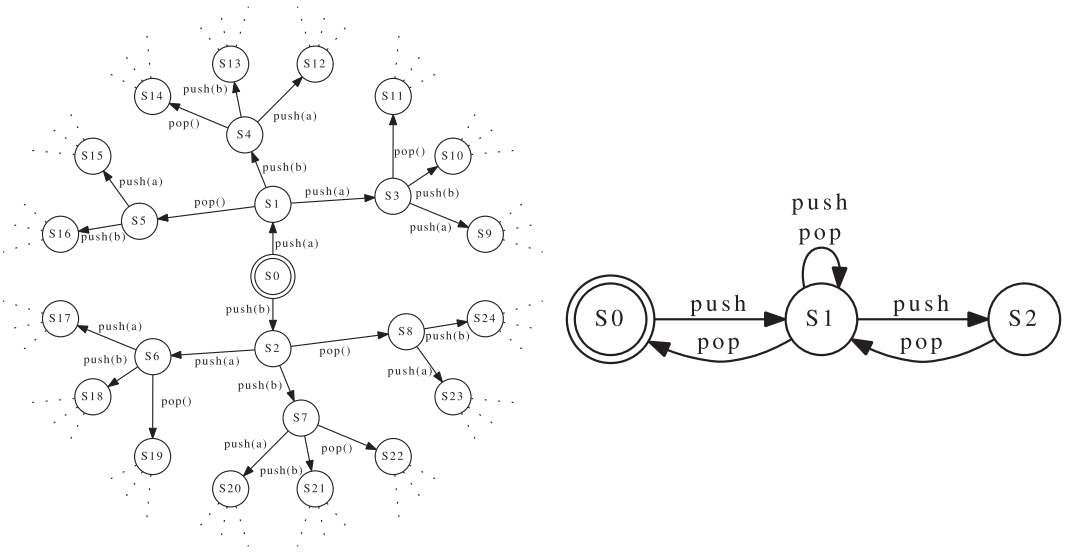


Figure 1. (a) Snippet of the intended protocol of a bounded stack and (b) its corresponding enabledness-preserving abstraction.

DOI: 10.1002/stvr

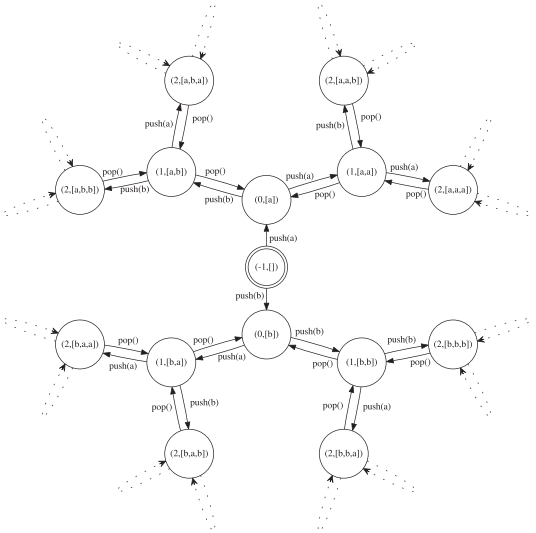
```
public class BoundedStack {
   Object[] elems;
   int top;

   public void BoundedStack() {
     elems = new Object[1000];
     top = -1;
   }

   public Object pop() {
      if (top <= 0) {
        throw new
            IllegalStateException();
      } else {
        return elems[top--];
      }
   }

   public void push(Object o) {...
}
```

(a)



(b)

Figure 2. Snippets of the (a) faulty implementation of a bounded stack and (b) its actual protocol.

it. Note that there are two outgoing transitions from S_0, each of which corresponds to a different actual parameter. Let us consider now the faulty implementation shown in Figure 2(a). It fails when trying to pop the last element of the stack: the `IllegalStateException` should be thrown when `top` is less than 0 but not when it equals 0. Figure 2(b) shows part of its actual protocol LTS. As aforementioned, it fails when `pop` is invoked on a `BoundedStack` containing only one element. Consequently, there are no transitions to go back to the initial state. Therefore, the sequence $[push(a), push(a), pop(), pop()]$ is a failure, because it is part of the intended call protocol but not of the actual call protocol. In practice, a client following the intended call protocol may fail if the implementation is non-conformant.

2.2. *Enabledness-preserving abstractions*

The aim is to define an adequacy criterion based on its protocol's behaviour. Therefore, the criterion should be independent of the specification language used to express the intended protocol and the programming language used to implement it.

Classical protocol testing approaches are defined over finite state machines [35]; however, protocol behaviour is a typically infinite state. In this paper, the proposal is to work over a *finite state abstraction* of the intended protocol. The abstraction proposed is the EPA [29]. Basically, an EPA is an LTS where labels are method names (with no concrete parameters). EPAs abstract the state space of the protocol LTS by quotienting it according to the methods that are enabled. In other words, two states of a protocol LTS are represented by the same abstract state if and only if for every method and concrete parameters enabled in one state, the same method is enabled in the other state (possibly with different concrete parameters). Consequently, an EPA state can be thought of as representing a particular subset of methods and abstracting all concrete states for which for every method in the subset, there are some concrete parameters for which calling that method on that state is valid in the protocol LTS. EPAs abstract parameters by introducing a transition between abstract states only if there exist parameter values such that a concrete state of the source abstract state can lead to a concrete state of the target abstract state (i.e. existential elimination). Figure 1(b) shows the EPA of the intended protocol LTS of a bounded stack depicted in Figure 1(a).[‡] State S_0 abstracts concrete states in which the only enabled action is $push$ (an empty stack), S_1 abstracts states where $push$

[‡]Note that the single arrow from S_1 to S_1 represents two different transitions, one for $push$ and the other for pop.

DOI: 10.1002/stvr

and *pop* are both enabled (a neither full nor empty stack), and S_2 abstracts concrete states where the only enabled action is *pop* (a full stack).

In order to formally define EPAs, the notion of states enabledness equivalence must be defined.

Definition 4 (enabledness equivalence)
Given a protocol LTS $L = \langle \Sigma = \bigcup_{i \leqslant n}(\{m_i\} \times \mathcal{D}_i),\ \mathbb{S},\ \mathbb{S}_0,\ \Delta \rangle$ over method names $m_1, \ldots, m_n$ and $\mathcal{D}_i$ as the domain of m_i and two states $s_1, s_2 \in \mathbb{S}$, s_1 and s_2 are *enabledness equivalent* states (noted $s_1 \equiv s_2$) if and only if for every $m_i \exists p \in D_i . s_1 \rightarrow^{m_i(p)} \iff \exists p' \in D_i . s_2 \rightarrow^{m_i(p')}$.

States S_1 and S_2 of the intended protocol of Figure 1(a) are enabledness equivalent, because both states enable the same set of actions (note that S_4, S_6 and many other states are equivalent to S_1 and S_2 as well). Given an LTS describing a protocol, its EPA is defined as a finite, potentially non-deterministic, state machine which groups the protocol states according to the actions that they enable.

Definition 5 (enabledness-preserving abstraction)
Given a protocol LTS $L = \langle \Sigma = \bigcup_{i \leqslant n}(\{m_i\} \times \mathcal{D}_i),\ \mathbb{S},\ \mathbb{S}_0,\ \Delta \rangle$ for a protocol over method names $m_1, \ldots, m_n$ and $\mathcal{D}_i$ as the domain of m_i, the LTS $M = \langle \bigcup_{i \leqslant n}(\{m_i\}), S, S_0, \delta \rangle$ is the EPA of L if there exists a total function $\alpha : \mathbb{S} \rightarrow S$ such that $\alpha(\mathbb{S}_0) = S_0$ and for every $s, s' \in \mathbb{S}$ and every method name $m_i, (\alpha(s), m_i, \alpha(s')) \in \delta \iff \exists p \in D_i . s \rightarrow^{m_i(p)} s'$. Furthermore, given a pair of states s_1 and s_2 on $\mathbb{S}$, it holds that $s_1 \equiv s_2 \iff \alpha(s_1) = \alpha(s_2)$.

Enabledness-preserving abstractions were used in previous work for validation of specifications [36] and programs [29]. Also, there is tool support for constructing EPAs either from a contract-based specification [36] or from source code [29].

2.3. Test coverage criterion

Naively measuring coverage of a test suite on an infinite state space would typically yield zero as a result [37]. Measuring the coverage on the entire protocol LTSs would fall in this case. Considering that they can be abstracted as EPA models, it is natural to define and investigate whether the latter would be effective as a basis for defining a coverage criterion for conformance testing. In this paper, a criterion based on transition coverage is defined and its effectiveness as test suite quality predictor is studied.

A *unit test* is defined as a sequence of method invocations with concrete parameters and a *test suite* as set of unit tests. Note that the effect of the execution of a unit test over an instance can be univocally interpreted as a path along a protocol LTS. In turn, (and because EPAs can simulate all paths of the LTS they abstract and each protocol LTS state is abstracted by only one EPA state) a path on the protocol LTS can be unequivocally simulated by a path in the EPA by applying the abstraction function α. The latter is said to be an *α-abstracted execution* of the unit test. Figure 3 shows an example of a unit test for a `BoundedStack` object and its corresponding α-abstracted execution. Note that the non-conformant implementation of Figure 2(a) throws an exception every time the α-abstracted execution of a unit test traverses the transition (S_1, pop, S_0).

The adequacy criterion over EPAs of protocol LTS, which simply consists in covering the transitions of the protocol's EPA, is defined as follows.

Definition 6 (EPA transition adequacy level)
Let L be a protocol LTS, A its EPA and TS a test suite. The adequacy level of TS with respect to

```
public void test() {
  BoundedStack bs = new BoundedStack();
  bs.push('a');
  bs.pop();
  assertEquals(0,size(bs));
}
```

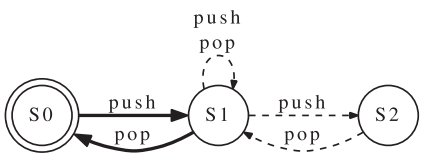


Figure 3. Sample input and its α-abstracted execution over the enabledness-preserving abstraction of the intended protocol labelled transition system.

DOI: 10.1002/stvr

A according to the EPA transition coverage criterion is defined as the percentage of transitions of A that are covered by the α-abstracted execution of the unit tests of TS.

As an example, the adequacy level of a test suite consisting only of the unit test of Figure 3 is 33, because it covers two out of six transitions. In particular, `push` is only invoked on an empty stack. A test suite with adequacy level 100 should necessarily contain in addition a unit test in which `push` is executed on a non-empty stack leading to full stack and one unit test in which `push` is executed on a non-empty stack leading to stack that is neither empty nor full.

2.4. Research questions

The research questions that are the focus of the experimentation reported in the next sections are now posed. The first question aims at establishing if the EPA transition adequacy level is a good predictor of the ability of a test suite to detect faults that manifest themselves as violations of the intended call protocol, that is, call protocol conformance failures.

RQ.1. To what extent does the EPA transition adequacy level predict the ability of test suites to detect faults in the context of call protocol conformance testing?

This question is first studied by looking at the correlation between the coverage of EPA transitions and the ability of a test suite to detect faults.

RQ.1.1. Does the EPA transition adequacy level have a high correlation with fault detection in the context of call protocol conformance testing?

Correlations are typically considered high when equal to or above 0.7. However, a comparative measure would provide a more robust answer to RQ.1.1. Consequently, the question is also answered by measuring how well branch and statement coverage performed over the code of the subjects (i.e. as white box criteria) would predict test suite fault detection ability. The importance of this analysis is twofold. First, it provides a baseline reference for correlations yielded by the proposed criterion. Second, it enables a comparison against what would be measuring structural adequacy using an ideal specification in terms of closely mimicking the structure of code under test. Choosing the code of the reference implementation as the specification language against which to compare EPA coverage is discussed in Section 5.

Because requiring coverage leads naturally to requiring larger test suites, it is standard to analyse if stronger correlations are just a consequence of size or if the criterion has an independent effect on test suites quality. More specifically, this work includes studying if picking a test suite with higher adequacy is more likely to detect more faults than picking a test suite of the same size but with lower adequacy.

RQ.1.2. Given a fixed test suite size, do test suites with higher EPA transition adequacy level perform better in terms of fault detection than those with lower EPA transition adequacy level?

RQ.2 aims at providing some insights that can explain the results of RQ.1.

RQ.2. Why does EPA transition adequacy level predict the ability of test suites to detect faults that manifest themselves as call protocol conformance failures?

Although code coverage does not always correlate to test suite effectiveness [38], programs are expected to be tested using test suites that achieve high structural coverage (otherwise, portions of the program are under-tested). Therefore, achieving high code coverage can be considered a necessary (but not sufficient) condition for test suites. Hence, a high correlation between EPA transition coverage and code coverage could partially explain the results of RQ.1.

RQ.2.1. Does EPA transition adequacy level predict the statement and branch coverage?

DOI: 10.1002/stvr

For RQ.2.1., the coverage of each test suite over a reference implementation and correlations are analysed. As before, the impact of EPA coverage on code coverage for fixed-size test suites to account for dependencies between coverage and size is analysed.

Analytical results in [30] indicate that category partition approaches that define dense subdomains (i.e. subdomains that have a high likelihood of identifying a fault) are likely to be better at identifying faults. Any coverage criterion implicitly defines a category partition; in the case of EPA transition coverage, a subdomain for a transition can be defined as all the tests that when executed cover the transition. If such implicit category partition indeed defines dense subdomains, this would contribute to explaining results in RQ.1.

RQ.2.2. Does the category partition implicitly defined by EPA transition coverage result in subdomains with high failure rates?

Although a test in the subdomain defined by a transition covers that transition, it may reveal a fault at any point of the test. Thus, RQ.2.2 provides a rather coarse-grained insight on the effectiveness of each transition in detecting faults. A finer-grained question would be to investigate the likelihood that each transition has, when exercised, of revealing a fault. If every fault has at least one transition that is highly effective in detecting it, then it is reasonable to think that the higher the coverage, the more likely the effective transition is covered.

RQ.2.3. Do faults have transitions that are highly effective in detecting them?

The third main question relates to differentiating the effect of action coverage and transition coverage. Each transition of an EPA is associated with a particular action in a many-to-one relation. For instance, in the EPA of a bounded stack shown in Figure 1(b), there are three different transitions corresponding to the action `push` and three different transitions corresponding to the action `pop`. Clearly, achieving high EPA transition adequacy requires a degree of action coverage. Thus, it is important to determine whether the observed phenomena for RQ.1 and RQ.2 are really determined by transition coverage. Therefore, in RQ.3 questions, RQ.1 and RQ.2 are revisited for action coverage, and the results are compared against those obtained for EPA transition coverage.

RQ.3. How do results for RQ.1 and RQ.2 compare when action coverage is considered instead of EPA transition coverage?

Finally, experimentation investigates if action coverage and EPA transition coverage can be combined to obtain a better criterion.

RQ.4. How do results for RQ.1 and RQ.2 compare when a combined Action-EPA transition coverage criterion is adopted?

3. EXPERIMENT DESIGN

3.1. Experiment overview

To answer the research questions proposed in Section 2.4, various types of values associated to test suites must be collected: *faults detected*, *code coverage*, *EPA transition coverage*, and *action coverage*. The experimental strategy is based on code mutations as in other works [39–41].

Fault detection involves (i) fixing both an intended protocol LTS and a conformant implementation and (ii) obtaining implementations that fail to conform to the intended protocol in diverse ways. In general, this would imply obtaining a specification of the intended protocol together with a conformant implementation. That may introduce a major threat to validity as there is no guarantee that an implementation is conformant with respect to a specification. For avoiding such risk, our strategy involves selecting an implementation as a *reference implementation* to be used as both the specification and implementation of the intended behaviour. This way, the actual protocol LTS of

DOI: 10.1002/stvr

the subject implementation is conformant by construction. Also note that the *intended protocol LTS* is in fact the actual protocol LTS of the reference implementation. This may be a typical case for regression testing when the actual protocol of the original version is intended to be preserved by the newer versions.

The reference implementation is also used as the basis for generating non-conformant or *faulty implementations*. The faulty implementations are obtained by applying mutation operators to the reference implementation. Identification of failures is carried out by executing unit tests on both the reference implementation and on the mutated implementations. When mutation is unable to execute a sequence of calls that is valid in the reference implementation, then the mutant is considered to be killed. More precisely, a failure is recorded if the mutated version throws an exception or timeouts on method call sequences that raise neither exceptions nor timeout in the reference implementation, or in other words, if a unit test is a witness to a conformance failure between the intended protocol LTS and the actual protocol LTS of the mutated implementation.

Note that semantically different mutations do not necessarily alter the actual protocol. For instance, altering the way an index is updated may (or may not) eventually lead to a state where some operation yields an exception. Between 56% and 70% of mutations (depending on the case study) produce faulty implementations. To have a representative set of flawed implementations, no mutation operator was filtered a priori. As in some other papers, mutants that were not killed by any unit test (i.e. no test sequence led to an unexpected behaviour of that mutant) were considered mutations that have the same actual protocol as the reference implementation of the class [39].

3.2. Subjects

The universe of potential subjects was restricted to one programming language to allow for a uniform experimental platform regarding mutation, test generation and infrastructure for detecting failures. The programming language chosen was Java to take advantage of existing tools and the availability of Java classes that satisfy the general criteria for subject selection: (i) code that features a rich set of restrictions on the order in which methods should be called (i.e. rich call protocols); (ii) code that is of industrial relevance; and (iii) code for which its EPAs can be obtained (Section 3.3).

The experimentation was conducted on five subject classes: `Signature`, `ListItr` and `Socket` from the Java Development Kit (JDK) 1.4 implementation; the `SMTPProcessor` class of JES mail server, a Java SMTP and POP3 e-mail server; and `JDBCResultSet` class, which is the implementation of the `ResultSet` interface of the JDBC specification of HyperSQL 2.0.0 database.

The Java `Signature` class is used to provide applications for the functionality of a digital signature algorithm. There are three phases to the use of a `Signature` object for either signing data or verifying a signature: (i) initialization, with either a public key, which initializes for verification, or a private key, which initializes for signing; (ii) updating, which updates the bytes to be signed or verified; and (iii) signing or verifying a signature on all updated bytes. The `ListItr` class provides functionality to go through the elements stored in a list. If the end of the list has not been reached, the iterator can retrieve the next element. Conversely, it can retrieve the previous element if the current index is not 0. It has methods for adding, removing and replacing elements in the list. The `Socket` class provides the client-side functionality to establish a transmission control protocol connection between two hosts. A `Socket` instance can open and close connections to a server and also can operate on streams for either send or receive data, among other operations. Note that a smaller version of the class was used in which a subset of the public methods is considered. Methods considered are those that are part of rich protocol requirements for the class. This version was previously used in other works [42, 43]. The `SMTPProcessor`, which is a core class of the Java Email Server, is responsible for processing all incoming SMTP requests. It checks whether a request is valid or not in terms of the restrictions defined on the SMTP protocol specification. In the presence of valid requests, it processes them, whereas it rejects the invalid ones. Finally, the `JDBCResultSet` class implements the `ResultSet` interface of the JDBC specification. A `ResultSet` represents a set of data that is generated by

DOI: 10.1002/stvr

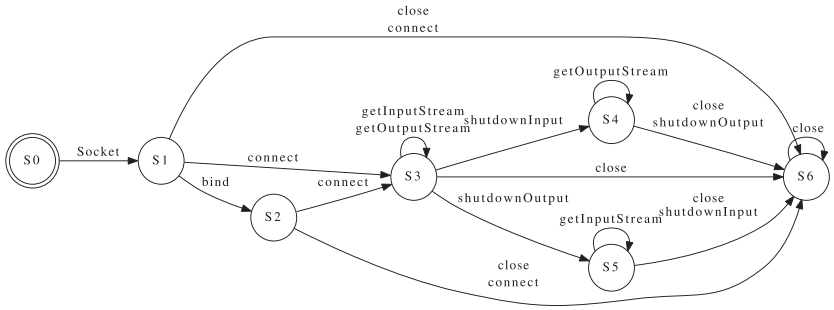


Figure 4. Enabledness-preserving abstractions of the semantics of `Socket` intended protocol.

executing a query to a database. The class allows iterating over the result and making updates on the underlying database. It maintains a cursor pointing to the current row of data and provides methods to scroll the data back and forth. Also, it supplies methods for making updates on the underlying database.

3.3. Construction of EPAs of intended protocol LTSs

The EPAs of the intended protocol for each subject are built from the original reference implementations, which act as infinite state specifications. We used different construction strategies for each case study.

A key resource for all subjects was the tool CONTRACTOR [36]. It constructs EPA either from contract-based specifications [36] or directly from source code [29]. CONTRACTOR's output models over-approximate EPAs in the sense that they may include spurious transitions. When computing the model from contracts, the tool relies on a first-order theorem prover. If neither a proof nor a counterexample can be found to determine whether a transition exists or not, the tool simply adds it because it may be present in the EPA. In a similar way, when the calculation is made from code, it uses a software model checker that tries to determine whether some statements present in the code are reachable or not (which is undecidable in the general case). Again, when no answer is found, CONTRACTOR conservatively adds the corresponding transition.

Because the intended protocol is required to be the actual protocol of the reference implementation, the EPAs of the intended protocol for subjects `Signature`, `ListItr`, `Socket` and `SMTPProcessor` were obtained by using CONTRACTOR on code of the reference implementation as performed by De Caso *et al.* [29]. In all cases, no spurious transitions were generated, so the constructed EPAs are those of the actual protocols of the reference implementations. On the other hand, the EPA of the actual protocol LTS for `JDBCResultSet` was obtained by using CONTRACTOR on the contract-based specification defined in the work of Bierhoff *et al.* [34] for the `ResultSet` JDBC interface.[§] The contract specification was validated by comparing the preconditions of each method against the conditions that guard exception throwing statements in the reference implementation. Indeed, the analysis performed concludes that resulting EPA is that of actual protocol LTS for the `JDBCResultSet` reference implementation. Figure 4 shows the EPA of JDK 1.4 `Socket` class.

3.4. Experiment implementation details

In this study, for each subject 10 000 unit tests were generated by using RANDOOP [44], an automatic unit test generator for Java classes. It uses a random approach that generates test sequences by randomly choosing method calls.

The μ-JAVA tool [45], a mutation system for Java programs, was used to generate mutations. In order to obtain as many implementations as possible, all available mutation operators of μ-JAVA were applied whenever possible. Some of these mutants may not be semantically equivalent, but

[§]Available at http://www.cs.cmu.edu/ kbierhof/plural/plural-java-apis.zip

DOI: 10.1002/stvr

Table I. Subject classes summary.

			Mutants			EPAs Transitions	
Subject	LOC	Unit tests	Total	Killed	States	Total	Covered
`ListItr`	59	10000	207	145 (70%)	8	68	63 (93%)
`Signature`	121	10000	150	96 (64%)	4	29	27 (93%)
`Socket`	144	10000	98	59 (60%)	7	20	18 (90%)
`SMTPProcessor`	404	10000	570	317 (56%)	12	85	70 (82%)
`JDBCResultSet`	785	10000	404	259 (64%)	9	247	199 (81%)

there is no evidence that their actual protocol is different to the one of the reference implementation. Although it may not be the case for every non-detected mutant, automatically detecting such equivalences has been proved to be undecidable [46]. Due to the large number of mutants generated, a manual inspection for detecting equivalences would have been unfeasible and certainly error prone. Mutants that were not killed were considered conformant as in the work of Andrews *et al.* [39].

The setting used is restricted to deterministic failure behaviour of API implementations. This restriction greatly simplifies experimental methodology. To perform a quantitative analysis of the effectiveness of the proposed criteria, information on which tests kill which mutants (i.e. which sequences of method invocations reveals that an implementation does not behave as expected) must be collected. If non-deterministic API implementations were included, the notion of test effectiveness (killing a mutant) would require probabilistic treatment because a non-deterministic mutant could behave differently in terms of failing or not to accept a given sequence of calls. Thus, each test would have a probability of killing each mutant. Assuming determinism allows us to perform mutation analysis, which is a common practice approach for empirical assessment of test techniques in software testing research [39, 47, 48]. The experiments were run in a controlled execution environment in order to minimize as much as possible a non-deterministic failure behaviour of the subjects. For instance, the experimentation performed on the case studies `SMTPProcessor`, `Socket` and `JDBCResultSet` were run on a single machine to avoid non-deterministic behaviour resulting from issues in network communication.

Statement and branch coverage were measured using COBERTURA, a Java tool that calculates the percentage of code accessed by tests. It instruments the class under test and records which lines of code are and are not executed as the test suite runs. It reports on both code and branch coverage.

Table I summarizes relevant information for each subject. Column 2 exhibit lines of code, column 3 shows the number of unit tests generated, columns 4 and 5 show the number of mutants generated and detected, respectively, column 6 indicates the number of states of the corresponding EPAs, and columns 7 and 8 show the total and reached (by at least one test suite) number of EPA transitions.

4. RESULTS

The analysis performed over the measurements obtained in the experiments detailed in the previous section is now reported.

4.1. Research question 1

The first research question seeks to quantitatively analyse the fault detection ability of EPA transition coverage. This question is addressed in two ways, first, looking at the correlation between EPA transition coverage and fault detection, and then looking at the impact of EPA transition coverage over fixed-size test suites.

4.1.1. Research question 1.1. In order to determine to what extent the EPA transition adequacy level predicts the ability of test suites to detect faults that manifests themselves as protocol conformance failures, the Spearman's rank correlation coefficient ρ was used. The coefficient measures how well the relationship between two variables can be described by a monotonic function. The coefficient

DOI: 10.1002/stvr

Table II. Correlation between coverage and fault detection.

Subject	Stmt coverage	Brch coverage	Tx coverage
`ListItr`	0.48	0.61	**0.84**
`Signature`	0.65	0.71	**0.73**
`Socket`	0.81	**0.86**	0.72
`SMTPProcessor`	0.66	**0.78**	0.69
`JDBCResultSet`	0.89	**0.92**	0.68

Bold emphasis indicates highest value for a row.

lies in the interval $[-1, 1]$. Values near 1 and -1 signal a strong dependence between both variables (direct and inverse, respectively), while values around 0 indicate no dependence between them. The choice of the Spearman's coefficient is because unlike others commonly used, such as Pearson product-moment coefficient, it does not make any assumption about the distribution of data. This is important as it was not possible to establish that data fitted known distributions using goodness of fit tests such as Kolmogorov–Smirnov.

The coefficient was computed not only to correlate EPA adequacy against detected faults but also for correlating statement and branch coverage against detected faults. The coefficients provide a measure of statistical dependence between the degree of coverage of each criterion and the number of detected faults by a test suite.

Table II shows the ρ-values obtained for each criterion. Transitions coverage has high ($\rho > 0.7$) or very close to high correlation for all cases. On the other hand, white box criteria correlation values are more disperse, indicating only moderate correlation in some subjects.

Two case studies illustrate the best and worst cases for transition coverage and structural coverage criteria. On the one hand, comparatively worse performance of transition criterion on `JDBCResultSet` can be explained by the unbalanced API implementation of the subject. There is one method out of 42 that collects the 35% of all mutations of the class. For killing these mutants, a test should execute that method (and consequently its α-abstracted execution traverses a few specific transitions of the 247 of the EPA). In other words, for killing a large number of the mutants, traversing many transitions does not help; in these cases, a specific transition should be exercised. On the other hand, for `ListItr` transition, coverage has a high correlation compared with that of statement and branch coverage. This can be explained by the fact that the subject presents a simple code structure (no loops, few branches) within methods that can be easily covered by test suites. However, the structure of the intended protocol LTS, and hence its EPA, is quite rich. Achieving coverage of the EPA requires executing more complex sequences of method calls that explore interesting states of the iterator (such as getting to the end of the list). High code coverage does not guarantee reaching such states.

As can be seen, black box EPA transition coverage has high correlation with fault detection. In some subjects, transition coverage seems to be a better proxy of exercised concrete behaviour than white box criteria. Because EPA transition coverage privileges diversity of transitions regardless of whether or not actions are covered, correlation with fault finding is reduced when a particular action is error prone.

4.1.2. Research question 1.2. The impact of size (as the number of calls to the software under test) on the effectiveness of test suites has been addressed by several works [49, 50]. Although it is known that not only the size determines the quality of a test suite, in general, it is expected that the likelihood of revealing faults increases with the number of invocations featured by a test suite [49].

In order to refute that EPA coverage correlates with fault detection only to required test suite size, experimentation included an analysis of whether test suites with higher EPA transition adequacy level are likely to detect more faults than those with lower adequacy when fixing the size of test suites.

Samples for each size are not large enough for obtaining statistically significant results. Therefore, they were divided into bins, grouping those of similar size in the same bin. For each subject, the

DOI: 10.1002/stvr

difference of size between test suites of the same bin do not exceed 10% of the difference of size between the largest and the smallest overall test suites.

Here, the question to answer is if given a bin, picking a test suite from the best test suites according to the defined criterion is more likely to give a test suite with higher fault detection ability than picking one from the rest of the test suites of that bin. The tests that achieve at least 80% of the coverage that is achieved by the test suite with highest coverage of that bin are considered to be those of high adequacy level. This is because the degree of coverage varies from bin to bin due to the change in test suite sizes. In this way the 'best' test suites for a particular bin can be obtained.

As explained in Section 4.1.1, detected fault data do not necessarily fit standard distributions. Therefore, as before, a non-parametric test for our analysis was chosen. In order to compare higher coverage test suites of a bin against the rest of that bin, the Mann–Whitney U-test was used, a non-parametric statistical hypothesis test for assessing whether the probability of an observation from one population exceeding an observation from a second population is not equal to 0.5. This hypothesis test assumes that all the observations from both groups are independent of each other, which is true because the test suites that do fit our criterion and those that do not form disjoint sets. It also requires that the responses are ordinal or continuous measurements, which is also true because the variables considered here are EPA transition adequacy level and detected faults. Under the null hypothesis, the probability of a random observation from one population P_1 exceeding a random observation from the second population P_2 equals the probability of an observation from P_2 exceeding an observation from P_1. Under the alternative hypothesis, the probability is not equal to 0.5. That is, values from one population tend to exceed those of the other. We reject the null hypothesis when the ρ-value resulting from the hypothesis test is less than 0.05. In this case, rejecting the null hypothesis means that tests with high transition coverage are better in terms of fault detection than those of low coverage, and that that is not just due to chance.

To also assess the magnitude of the improvement, the Vargha and Delaney's A_{12} effect size was used. This non-parametric measure has recently been advocated for randomized algorithms [51]. In this case, in a given bin, A_{12} estimates the probability that choosing a tests suite of high transition coverage detects more mutants than one chosen randomly from the population of low coverage. The confidence interval for the effect size stated at the 95% confidence level is also reported.

Table IV shows the test results for all subjects. Each row of the table corresponds to one bin. The second column specifies the test suite size interval of each bin. The third indicates the minimum and maximum number of EPA transitions covered by them, and the fourth shows the threshold number of transitions that a test suite must cover to be considered adequate (i.e. coverage of at least 80% of the best coverage in the bin). The fifth and sixth columns show the number tests suites that are adequate (i.e. high coverage) and not adequate. The seventh column exhibits the ρ-value of the Mann–Whitney test, the eighth the A_{12} effect size and the ninth its confidence interval. Remaining columns corresponds to results of RQ2.1.

Results indicate that EPA coverage makes a difference in terms of fault detection for tests suites of the same size. Considering all the five case studies, in 39 out of 50 bins, there is statistical evidence that test suite with higher coverage of EPA transitions perform better than those of lower coverage (column 7). Moreover, except for two bins of `JDBCResultSet`, those on which there is no statistical evidence correspond to bins containing very large test suites. That makes sense, because the impact of coverage tends to decrease as the size of test suites increases. For many of these bins, the sample size of test suites that does not reach the coverage threashold is quite small (ranging between

Table III. Correlation between EPA transitions and code coverage.

Class	TX/STMT CORR	TX/BRCH CORR
`ListItr`	0.66	**0.70**
`Signature`	0.63	**0.70**
`Socket`	0.77	**0.77**
`SMTPProcessor`	0.70	**0.73**
`JDBCResultSet`	**0.67**	0.66

Bold indicates highest value for a row.

DOI: 10.1002/stvr

Table IV. Mann–Whitney test results for fault detection and code coverage.

	EPA coverage					Fault detection			Statement coverage			Branch coverage		
Subject	SIZE	COV TX	THR	#TS$_{\geqslant \text{THR}}$	#TS$_{<\text{THR}}$	p	ES	CI	p	ES	CI	p	ES	CI
ListItr	[50,99]	16 - 37	30	90	222	~ 0	0.73	[0.67,0.79]	0.23†	0.54	[0.51,0.58]	~ 0	0.74	[0.69,0.79]
	[100,149]	22 - 44	36	68	219	~ 0	0.81	[0.75,0.86]	0.66†	0.52	[0.50,0.53]	~ 0	0.79	[0.75,0.82]
	[150,199]	24 - 48	39	66	215	~ 0	0.72	[0.66,0.79]	0.90†	0.50	[0.50,0.51]	~ 0	0.70	[0.67,0.74]
	[200,249]	29 - 48	39	163	135	~ 0	0.72	[0.66,0.77]	1.00†	0.50	[0.50,0.50]	~ 0	0.62	[0.58,0.66]
	[250,299]	32 - 50	40	189	113	~ 0	0.72	[0.66,0.78]	0.93†	0.49	[0.49,0.50]	~ 0	0.66	[0.61,0.70]
	[300,349]	34 - 51	41	203	89	~ 0	0.68	[0.62,0.74]	1.00†	0.50	[0.50,0.50]	0.02	0.58	[0.54,0.63]
	[350,399]	33 - 51	41	197	44	~ 0	0.75	[0.67,0.83]	1.00†	0.50	[0.50,0.50]	0.13†	0.58	[0.52,0.63]
	[400,449]	35 - 52	42	161	26	0.08†	0.61	[0.51,0.72]	1.00†	0.48	[0.48,0.48]	0.08†	0.62	[0.54,0.70]
	[450,499]	36 - 53	43	145	28	~ 0	0.73	[0.63,0.82]	1.00†	0.48	[0.48,0.48]	0.03	0.62	[0.54,0.70]
	[500,549]	36 - 52	42	113	14	0.08†	0.63	[0.49,0.77]	1.00†	0.50	[0.50,0.50]	0.02	0.72	[0.60,0.85]
Signature	[50,99]	9 - 22	18	98	189	~ 0	0.80	[0.71,0.90]	~ 0	0.76	[0.66,0.85]	~ 0	0.78	[0.68,0.88]
	[100,149]	11 - 25	20	114	110	~ 0	0.75	[0.64,0.86]	~ 0	0.76	[0.65,0.87]	~ 0	0.74	[0.63,0.85]
	[150,199]	12 - 26	21	110	130	~ 0	0.77	[0.66,0.87]	0.01	0.70	[0.60,0.80]	~ 0	0.80	[0.70,0.89]
	[200,249]	14 - 26	21	165	79	~ 0	0.77	[0.66,0.88]	~ 0	0.73	[0.61,0.85]	~ 0	0.80	[0.70,0.90]
	[250,299]	14 - 27	22	181	63	~ 0	0.83	[0.71,0.95]	~ 0	0.73	[0.62,0.84]	~ 0	0.81	[0.70,0.92]
	[300,349]	14 - 27	22	201	43	~ 0	0.85	[0.73,0.97]	~ 0	0.80	[0.67,0.92]	~ 0	0.83	[0.72,0.95]
	[350,399]	16 - 27	22	232	83	~ 0	0.75	[0.64,0.86]	0.02	0.67	[0.56,0.77]	~ 0	0.75	[0.65,0.85]
	[400,449]	18 - 27	22	205	35	~ 0	0.90	[0.80,1.00]	0.01	0.73	[0.61,0.85]	0.01	0.77	[0.63,0.90]
	[450,499]	17 - 27	22	220	31	~ 0	0.82	[0.70,0.94]	0.16†	0.65	[0.50,0.80]	~ 0	0.84	[0.69,0.98]
	[500,549]	17 - 27	22	197	14	0.01	0.86	[0.79,0.93]	0.01	0.83	[0.76,0.90]	0.01	0.77	[0.67,0.86]
Socket	[0,12]	2 - 9	8	29	141	~ 0	0.77	[0.70,0.84]	~ 0	0.86	[0.77,0.94]	~ 0	0.82	[0.74,0.91]
	[13,25]	5 - 13	11	36	261	~ 0	0.72	[0.64,0.79]	~ 0	0.89	[0.84,0.94]	~ 0	0.83	[0.77,0.89]
	[26,38]	5 - 14	12	64	227	~ 0	0.69	[0.63,0.76]	~ 0	0.78	[0.72,0.83]	~ 0	0.74	[0.68,0.80]
	[39,51]	7 - 16	13	73	227	0.01	0.60	[0.55,0.66]	~ 0	0.71	[0.66,0.76]	~ 0	0.70	[0.64,0.76]
	[52,64]	8 - 16	13	151	167	~ 0	0.59	[0.56,0.63]	~ 0	0.67	[0.62,0.71]	~ 0	0.66	[0.61,0.70]
	[65,77]	9 - 16	13	206	91	0.03	0.58	[0.54,0.62]	~ 0	0.67	[0.62,0.72]	~ 0	0.68	[0.63,0.73]
	[78,90]	10 - 17	14	170	143	1.00†	0.50	[0.48,0.52]	0.01	0.59	[0.55,0.62]	~ 0	0.59	[0.55,0.63]
	[91,103]	10 - 17	14	178	106	0.37†	0.53	[0.51,0.55]	0.01	0.59	[0.55,0.63]	0.01	0.59	[0.55,0.63]
	[104,116]	10 - 16	13	159	18	0.03	0.63	[0.54,0.72]	~ 0	0.72	[0.61,0.82]	~ 0	0.71	[0.61,0.81]
	[117,129]	11 - 16	13	45	8	1.00†	0.40	[0.40,0.40]	0.63†	0.33	[0.29,0.38]	0.30†	0.26	[0.12,0.39]
SMTPProcessor	[0,149]	4 - 28	23	12	184	~ 0	0.80	[0.68,0.93]	~ 0	0.86	[0.78,0.94]	~ 0	0.88	[0.80,0.96]
	[150,299]	11 - 37	30	21	242	0.03	0.65	[0.56,0.75]	0.02	0.66	[0.57,0.75]	0.01	0.68	[0.59,0.77]
	[300,449]	20 - 38	31	86	161	~ 0	0.63	[0.57,0.70]	~ 0	0.65	[0.59,0.72]	~ 0	0.65	[0.58,0.72]
	[450,599]	21 - 40	32	122	137	~ 0	0.61	[0.55,0.67]	0.03	0.59	[0.52,0.65]	0.04	0.57	[0.51,0.64]
	[600,749]	24 - 45	36	104	150	~ 0	0.62	[0.56,0.68]	~ 0	0.61	[0.55,0.68]	~ 0	0.62	[0.55,0.68]
	[750,899]	26 - 46	37	114	136	0.01	0.61	[0.55,0.67]	0.01	0.60	[0.54,0.66]	~ 0	0.62	[0.56,0.69]
	[900,1049]	28 - 50	40	96	186	~ 0	0.64	[0.58,0.70]	0.02	0.59	[0.53,0.65]	~ 0	0.62	[0.56,0.68]
	[1050,1199]	31 - 48	39	171	97	0.02	0.59	[0.53,0.65]	0.20†	0.55	[0.49,0.61]	0.03	0.58	[0.52,0.65]
	[1200,1349]	31 - 49	40	153	105	0.13†	0.55	[0.49,0.62]	0.01	0.61	[0.54,0.67]	~ 0	0.62	[0.56,0.68]
	[1350,1499]	30 - 50	40	168	55	0.45†	0.54	[0.46,0.61]	0.50†	0.53	[0.45,0.61]	0.07†	0.58	[0.50,0.66]
JDBCResultSet	[200,399]	69 - 107	86	75	81	~ 0	0.70	[0.62,0.79]	~ 0	0.66	[0.57,0.74]	~ 0	0.66	[0.57,0.75]
	[400,599]	80 - 114	92	213	56	~ 0	0.74	[0.66,0.81]	~ 0	0.72	[0.64,0.80]	~ 0	0.68	[0.60,0.76]
	[600,799]	90 - 123	99	258	25	~ 0	0.78	[0.67,0.88]	0.01	0.68	[0.58,0.79]	~ 0	0.72	[0.60,0.83]
	[800,999]	94 - 134	108	231	32	0.10	0.59	[0.48,0.70]	0.04	0.55	[0.44,0.66]	0.04	0.53	[0.43,0.64]
	[1000,1199]	102 - 143	115	172	63	0.02	0.62	[0.53,0.70]	0.02	0.61	[0.52,0.70]	0.04	0.57	[0.48,0.65]
	[1200,1399]	108 - 140	112	246	8	0.73†	0.41	[0.21,0.61]	0.03	0.33	[0.19,0.46]	0.04	0.39	[0.17,0.61]
	[1400,1599]	106 - 140	112	246	11	0.09†	0.73	[0.69,0.76]	0.09†	0.76	[0.72,0.79]	0.10†	0.68	[0.65,0.72]
	[1600,1799]	107 - 151	121	252	19	0.01	0.69	[0.58,0.80]	~ 0	0.76	[0.67,0.85]	~ 0	0.72	[0.64,0.81]
	[1800,1999]	119 - 153	123	230	17	0.86†	0.47	[0.21,0.74]	0.49†	0.53	[0.40,0.67]	0.94†	0.45	[0.27,0.62]
	[2000,2199]	120 - 153	123	248	17	0.06†	0.70	[0.57,0.82]	0.01	0.79	[0.68,0.89]	0.08†	0.65	[0.57,0.73]

Values not statistically significant are marked with †.

DOI: 10.1002/stvr

8 and 28 individuals in five of these bins), difficulting the possibility of obtaining statistical conclusions. The case of the bin corresponding to the largest test suites of `JDBCResultSet` illustrates this situation: although the effect size is 0.70, the number of test suites that do and do not reach the coverage threshold are 248 and 17, respectively. In this case, the ρ-value is 0.06. Consequently, according to the significance level chosen in our experiments in this case, the null hypothesis cannot be rejected.

4.2. Research question 2

This research question aims at providing more qualitative results that may provide insight on why positive results for RQ.1 are obtained. The question is analysed from three different perspectives: (i) correlation between EPA transition coverage and code coverage (*RQ.2.1*); (ii) the characteristics of category partition criterion implicitly defined by EPA transition coverage (an input i belongs to the subdomain defined by a transition t if and only if i exercises t) (*RQ.2.2*); and (3) how faults are detected by the traversal of EPA transitions (*RQ.2.3*).

4.2.1. Research question 2.1. To measure the correlation between EPA transition coverage and code coverage, the code coverage achieved by test suites must be analysed to see if it fits a standard distribution. As with the relation between EPA coverage and fault detection, this is not the case. Consequently, to find out how well the coverage of EPA predicts code coverage, Spearman's rank correlation coefficient ρ is again calculated. The results are shown in Table III.

Results lead to believe that the EPA coverage criterion is a reasonably good predictor of statement and branch coverage adequacy. Correlation against branch coverage is moderate to high depending on the case and values are consistently close to 0.7. Interestingly, except for transition coverage in `JDBCResultSet`, in all cases, the correlation with branch coverage is greater than or equal to that of statement coverage. In the types of software under analysis, the selection of branches of conditional expressions is often based on the internal state of the object. Thus, the high correlation values may be an indication that indeed, EPA states capture important properties of the object states and, therefore, may be useful to use them for abstracting concrete object states. It is worth noting that in almost all cases, the EPA transition coverage criterion works better as predictor of mutant detection than as code coverage predictor – which seems consistent to the rationale underlying criteria.

Similarly to RQ.1.2, tests that achieve higher EPA coverage are generally also larger, and also, larger tests tend to cover a larger portion of the code. Thus, the relation between EPA and code coverage in a 'by size' basis is analysed using the same set of bins as in RQ.1.2. Again, it is not assumed that code coverage achieved by test suites fits a common distribution. Thus, in order to determine if adequate tests achieve higher code coverage than non-adequate ones, Mann–Whitney U-tests were performed. As for fault detection, the assumptions made by the test are also met in this case.

Table IV shows the results for all subjects. Columns 10 to 12 and 13 to 15 exhibit the results for the relation between EPA transitions coverage and statement and branch coverage, respectively. In this case, in a given bin, A_{12} estimates the probability that choosing a test suite of high transition coverage has higher statement/branch coverage than a test suite chosen randomly from the population of low transition coverage. The confidence interval for the effect size is also reported at the 95% confidence level.

Again, results show that statistical significant evidence exists for saying that test suites achieving higher EPA coverage adequacy are more likely to achieve higher code coverage than the general population of a given size. Considering the five case studies, there is such evidence in 34 out of 50 bins in the case of statement coverage. Interestingly, 10 of the bins for which there is no evidence are the bins of `ListItr`. As explained before, the simplicity of its code makes it easy to achieve high code coverage, and therefore, covering many transitions does not make a significant difference (but it does affect positively fault detection, as seen). For `Socket` and `Signature`, the evidence exists in eight out of 10 bins, and for `JDBCResultSet` and `SMTPProcessor` in nine out of 10. Regarding branch coverage, in 43 out of 50 bins, there is statistical evidence of the benefits of tests with high EPA coverage.

DOI: 10.1002/stvr

As with fault detection, those bins for which there is no statistical evidence correspond to very large test suites (with the exception of `ListItr`), which is consistent with the fact that the impact of high coverage tends to decrease as the size of the test suite increases.

Summarizing, results indicate that EPA transition coverage has high correlation with branch coverage (except for the case of `JDBCResultSet`, which is quite close to 0.7 anyway), and a little lower correlation with statement coverage (but still with values close to 0.7 in all cases). In addition, for fixed-size test suites, there is statistical evidence that in most cases those tests with higher transition coverage perform better in terms of structural coverage.

4.2.2. Research question 2.2. Category partition testing refers to a general family of testing strategies [30]. They consist in dividing the program's input domain into subdomains in such a way that they span the domain's input space. Then the program is tested, selecting some inputs from each subdomain. Ideally, each subdomain should be such that the program under test behaves as expected for every element or fails for every element. When all the inputs of a subdomain cause the program to fail, the subdomain is called *100% revealing* or simply *revealing* [52]. The purpose is to make the partition in a way that the resulting test set results in a good representation of the whole program's domain.

As an example, given a program P with domain $\mathbb{Z}$, the subdomains $\mathbb{Z}^-, \{0\}$ and $\mathbb{Z}^+$ can be considered (negative integers, zero and positive integers). In this case, the input domain is divided into *disjoint* subdomains. However, the partition may also result in *overlapping* subdomains. Let us illustrate this with the subdomains derived from the statements of a program. They divide the input domain into non-disjoint subdomains where each subdomain consists of all test cases, which cause a particular statement to be executed. A test case causes many statements to be executed and, consequently, is a member several subdomains.

An important *analytical* study on partition testing is presented in [30]. The paper compares partition testing against random selection of inputs and shows the conditions under which one of them outperforms the other. It concludes that a partition testing strategy is neither good nor bad per se. Its effectiveness lies in how the domain is divided into subdomains. More precisely, it depends on how the inputs which cause a program to fail are concentrated within the subdomains defined by the partition. As aforementioned, an ideal scenario contains at least one revealing subdomain. No matter which input is selected from it, the program will fail and be revealed as incorrect. Of course, that such situation is not realistic: when testing a program, the location of faults is unknown and there is no guarantee that one subdomain contains only error-revealing inputs. On the other hand, when only one or a few elements of a subdomain make a program fail, then the density of faults in the relevant subdomains will be relatively low, and therefore, the strategy of picking from all subdomains will be of low effectiveness. What is more, under the latter circumstances, it may be the case that the probability of detecting a fault when using category partition is lower than the one resulting of a random selection of inputs. Still, if there is at least one subdomain that is dense in failures – that is, that most of its inputs are error-revealing – then the approach is effective [30].

The proposed coverage criterion implicitly divides the input's domain of an intended protocol LTS according to the transitions of its associated EPA. A unit test u belongs to the category defined by a transition t if and only if the α-abstracted execution of u exercises t. As with statements, the subdomains defined by transitions of an EPA may overlap. For instance, the test of Figure 3 belongs to the subdomains derived from transitions (S_0, push, S_1) and (S_1, pop, S_0).

In order to determine if there is a subdomain dense for a faulty implementation I, the failure rate of each subdomain must be known. That is, the proportion of inputs that are error-revealing for I. In other words, the probability p of a randomly chosen input of each subdomain to be error-revealing. In the case of finite domains, the calculation is quite simple, because it corresponds to the ratio between error-revealing inputs and the domain size. In general – and in particular in the selected subjects – the input domain space for protocols is infinite. Therefore, for each case study, estimate p is estimated by $\hat{p}$ based on experimental observations as follows:

1. for each EPA transition t, the sample size of the subdomain d_t derived from it (i.e. the number of unit tests whose α-abstracted execution traverses t) is calculated;

DOI: 10.1002/stvr

Table V. Total and filtered subdomains.

Subject	Total	Filtered
`ListItr`	68	14 (20.59%)
`Signature`	29	2 (6.90%)
`Socket`	20	4 (20.00%)
`SMTPProcessor`	85	35 (41.18%)
`JDBCResultSet`	247	87 (35.22)

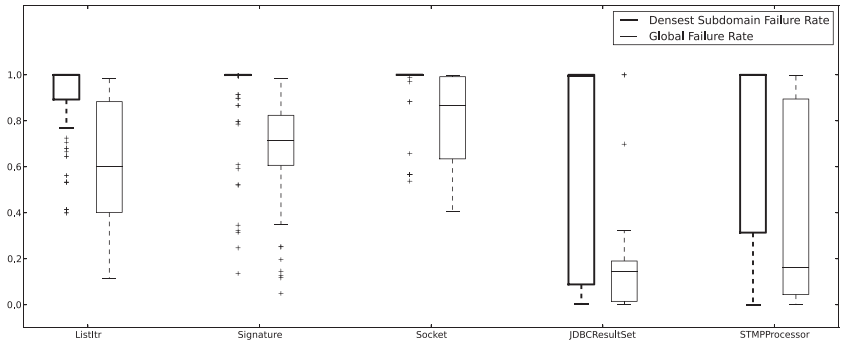


Figure 5. Failure rates of the densest subdomain and the entire domain.

2. for each faulty implementation I and subdomain d_t, the number of error-revealing inputs of the sample $er_{d_t,I}$ are counted;
3. for each faulty implementation I and subdomain d_t, $\hat{p}_{d_t,I} = \frac{er_{d_t,I}}{|d_t|}$ is calculated.

For determining if there is at least one subdomain dense in failures, for each faulty implementation I, the value $\hat{p}_{d_j,I}$ is selected, where d_j is the densest subdomain for I. As a reference, the density of the entire domain is also provided.

It is worth noting that not all subdomain sample sizes are large enough in order to obtain statistically significant results. For instance, in some cases, there are EPA transitions reached by only one unit test α-abstracted execution. That is, there are transitions t for which only one unit test was generated such that its $\alpha-$abstracted execution traverses t. In this case, the estimator $\hat{p}_{d_t,I}$ would equal 1 or 0 depending on whether the unit test kills the mutant I or not (and in the first case, it would be considered that there is a *revealing subdomain* for I). In any case, a sample of size 1 is too small to consider $\hat{p}_{d_t,I}$ as a good estimator of $p_{d_j,I}$. It would be misleading to consider such input as representative of the entire subdomain d_j.

Because the observations are independent, the estimator $\hat{p}$ has a binomial distribution. The maximum variance of this distribution is $0.25 * |d_t|$, which occurs when the real parameter is $p = 0.5$. Because p is unknown, the maximum variance is used for sample size assessments. For sufficiently large $|d_t|$, the distribution of $\hat{p}$ will be closely approximated by a normal distribution. Using this approximation, around 95% of this distribution's probability lies within two standard deviations of the mean. Using the Wald method for the binomial distribution, an interval of the form $(\hat{p} - 2\sqrt{0.25/|d_t|}, \hat{p} + 2\sqrt{0.25/|d_t|})$ will form a 95% confidence interval for the true proportion. If this interval needs to be no more than W units wide, the equation $4\sqrt{0.25/|d_t|} = W$ can be solved for $|d_t|$, yielding $|d_t| = 4/W^2 = 1/B^2$ where B is the error bound on the estimate. Fixing $B = 5\%$, samples sizes greater or equal to 400 are required for considering them representative. Table V shows for each subject the total number of subdomains (or transitions) and how many are filtered. Note that filtering subdomains does not favour the approach proposed. On the contrary, it could be the case that there is a subdomain for which there are not enough inputs whose real failure rate is greater than that of those of the subdomains that are not filtered.

Figure 5 shows boxplots of the failure rates of both, the densest subdomains with statistcally significant rates – that is, for which there are at least 400 inputs – and the entire domain. Numerical values of percentiles, means and means excluding outliers are shown in Table VI.

DOI: 10.1002/stvr

Table VI. Statistical info of failure rates considering subdomains and the entire domain.

	By subdomains					Entire domain				
Subject	Q_1	Med	Q_3	Mean	Mean EO	Q_1	Med	Q_3	Mean	Mean EO
`ListItr`	0.89	1.0	1.0	0.91	0.95	0.4	0.6	0.88	0.61	0.61
`Signature`	1.0	1.0	1.0	0.93	0.98	0.61	0.72	0.83	0.67	0.7
`Socket`	1.0	1.0	1.0	0.94	0.99	0.56	0.87	0.99	0.8	0.8
`JDBCResultSet`	0.09	1.0	1.0	0.67	0.67	0.01	0.14	0.19	0.14	0.13
`SMTPProcessor`	0.31	1.0	1.0	0.71	0.71	0.05	0.16	0.86	0.4	0.4

Table VII. Statistical info on failure rate for hard mutants considering subdomains and the entire domain.

			By subdomains					Entire domain				
Subject	#MUT	#HMUT	Q_1	Med	Q_3	Mean	Mean EO	Q_1	Med	Q_3	Mean	Mean EO
`ListItr`	145	14	0.41	0.53	0.56	0.62	0.62	0.14	0.16	0.16	0.16	0.16
`Signature`	96	5	0.25	0.31	0.32	0.27	0.27	0.12	0.13	0.15	0.13	0.13
`JDBCResultSet`	323	201	0.05	0.99	1.0	0.57	0.57	0.01	0.1	0.15	0.09	0.09
`SMTPProcessor`	259	176	0.0	0.31	1.0	0.49	0.49	0.0	0.01	0.16	0.08	0.08

It is noteworthy that in the five case studies, for at least 50% of the non-conformant implementations, the division into subdomains according to EPA transitions produces a revealing subdomain. What is more, in the cases of `Signature` and `Socket`, the same observation is valid for at least 75% of the faulty implementations; and in the case `ListItr` for 75% of versions that are not conformant, there is a subdomain with a density of 0.89 or greater. Although it does not happen the same in the cases of `JDBCResultSet` and `SMTPProcessor`, these have in common a significant increase of density with respect to the density of the entire domain. As aforementioned, half of the faulty implementations have a revealing subdomain for `JDBCResultSet` and `SMTPProcessor`, but when considering the entire domain, the numbers dramatically decrease: their medians are 0.14 and 0.16, respectively (i.e. in both cases, there is a difference around 85%).

It can also be observed the difference between subdomains failure-rate mean values and entire domain mean values: they range between 0.14 (for `Socket`) and 0.53 (for `JDBCResultSet`). In the case of `Socket`, the difference is small. This is explained by the (non-)difficulty of the mutants generated by μ-JAVA for this class: all non-conformant implementations are discovered by at least 40% of the unit tests and 65% of them are killed by 85% of the unit tests or more. That is, all of them are easy to discover, and therefore, the failure rates considering the entire domain are also high (mean values removing outliers present only a little difference).

The phenomenon described earlier does not occur in the other case studies, which, incidentally, all have harder to detect non-conformant implementations. The question that arises is if the phenomenon observed remains when restricting the set of mutants to those that are hard to kill. Thus, the same analysis is repeated but only considering mutants that are discovered by less than 20% of the unit tests. Columns 2 and 3 of Table VII show the total number of mutants and number of mutants that are hard to detect of each case study. Note that there are no such mutants for `Socket`. Results are shown in Figure 6 and Table VII.

As expected, when restricting the set of non-conformant implementations to those that are not easy to detect, failure rate values decrease. Still, the numbers of the subdomains derived from EPA transitions outperform, by far, those of the entire domain. In the cases of `JDBCResultSet` and `SMTPProcessor`, there are revealing subdomains for at least 25% of the faulty version (in fact, in the case of `JDBCResultSet`, for 50% of them, there is a subdomain with failure rate of at least 0.99). Interestingly, these are the two cases in which the number of considered implementations is relatively high. Regarding failure rate mean values, as before, there are no significant differences when removing outliers.

Experimental observations show that for a large number of faulty implementations, there are subdomains with high failure rates. Moreover, in all case studies, there are *revealing subdomains* for at least half of them. In other words, for a large number of faults, partitioning the input space

DOI: 10.1002/stvr

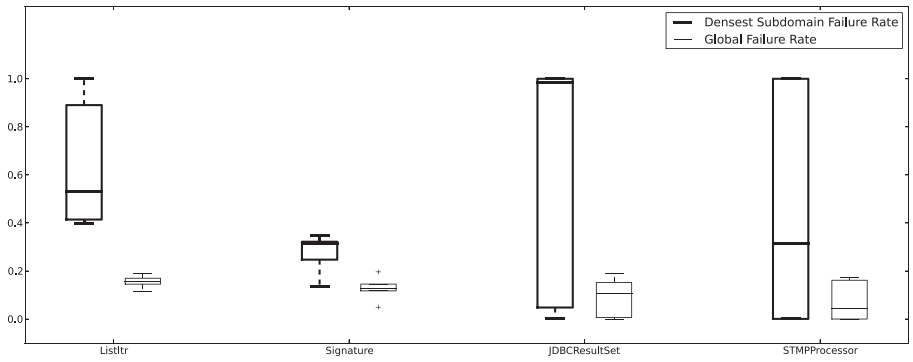


Figure 6. Failure rates of the densest subdomain and the entire domain of hard mutants.

according to the categories defined by transitions generates subdomains that meet the conditions under which a category partition approach results effective [30].

4.2.3. Research question 2.3. From RQ.2.2, answer one is tempted to explain the phenomenon as follows: given a faulty implementation I, there is at least one EPA transition that, when exercised, it is very likely that I is exposed as non-conformant. Note that the existence of dense subdomains does not necessarily imply the existence of effective transitions: a transition might only exhibit a fault after being exercised several times (low effectiveness). However, if most test cases traverse the transition several times, the subdomain defined by that transition would be dense in failure detection terms. This research question explores if intuition on the existence of effective transitions is supported by data.

More formally, suppose that when executing unit tests over an implementation I, every time that a transition t is traversed by a unit test, the concrete execution raises an exception. That is, that every time t is traversed, I is exposed as a non-conformant implementation. In such a case, t can be considered as highly effective for exposing I. It is investigated whether this is the case or not.

Given an implementation I and a transition t, the effectiveness of t for I (notation: $\text{eff}_{t,I}$) is defined as the ratio between the number of times that I is exposed while traversing t and the number of times that t is traversed. It is easy to see that $\text{eff}_{t,I} \in [0, 1]$. While values near 1 denote that t is highly effective for revealing I, values near 0 indicate the opposite.

In order to understand the effectiveness of transitions for exposing non-conformant implementations (i) the effectiveness of transitions were divided into 10 buckets, corresponding to values in intervals $[0, 0.1], (0.1, 0.2], \ldots, (0.9, 1.0]$, and (ii) for each non-conformant implementation, the percentage of EPA transitions whose effectiveness resides in each bucket was counted. Boxplots in Figure 7 show the distribution of the effectiveness of transitions across the buckets for the five case studies.

Something similar happens in the five case studies: given a non-conformant implementation, most of the transitions have low effectiveness, a few of them have high effectiveness, and in the middle, there is almost nothing. The fact that in general there is at least one transition of high effectiveness supports previous observations. Suppose that a transition t is highly effective for exposing an implementation I as non-conformant – that is, that almost always that t is traversed I raises an exception or hangs. All the inputs of the subdomain derived from t have in common that at some point they traverse t. Therefore, it is not surprising that that subdomain results dense for I.

It can be observed that for non-conformant implementations in general, there is a highly effective transition for it. A key question, thus, is whether there is one (or a small number of) highly effective transition for all non-conformant implementations or if the highly effective transition varies according to the fault. Figure 8 shows the percentage of non-conformant implementations for which each transition is highly effective in the case of `ListItr`. There is one bar for each transition, they are grouped by actions, and for each action, the number of associated transitions is indicated.

Interestingly, the transition that is highly effective for detecting a non-conformant implementation varies significantly. No transition is of high effectiveness for more than the 22% of the non-

DOI: 10.1002/stvr

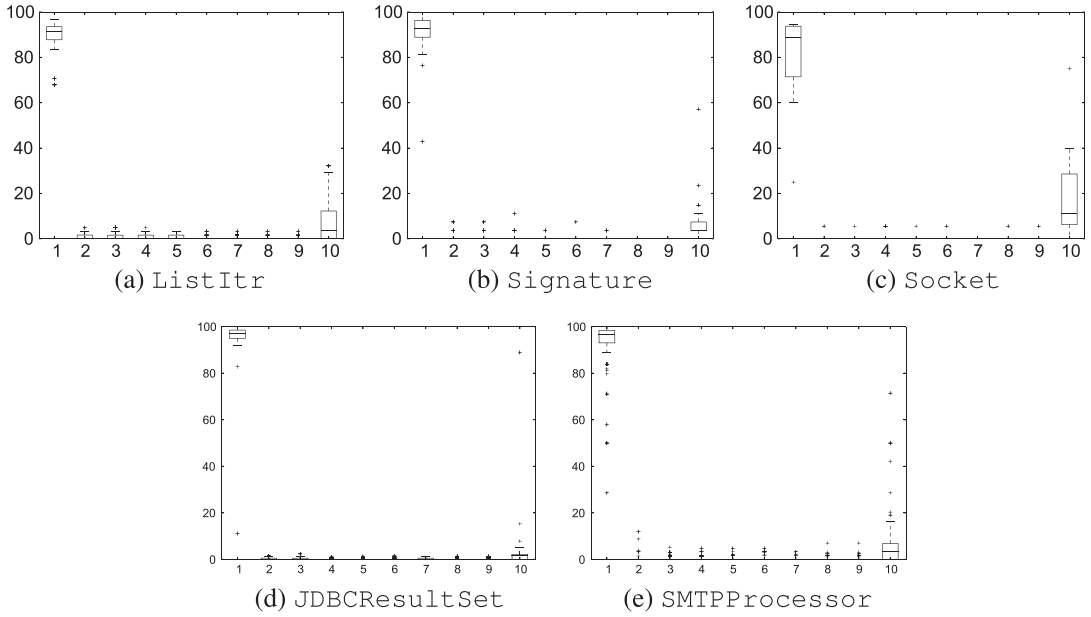


Figure 7. Effectiveness of transitions for exposing non-conformant implementations.

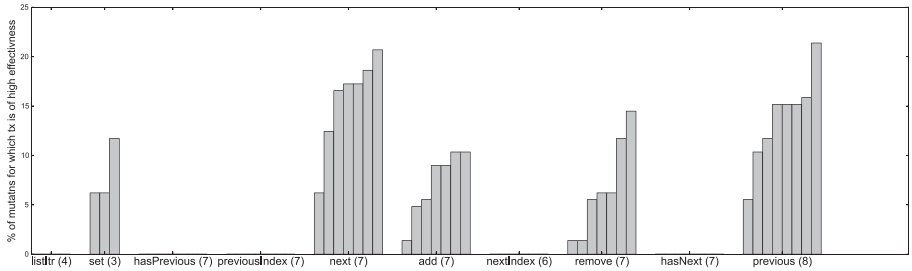


Figure 8. Percentage of mutants for which transitions are highly effective in `ListIterator`.

conformant implementations, and the same happens with `Signature` and `SMTPProcessor`. In the case of `JDBCResultSet` and `Socket`, no transition is of high effectiveness for more than the 35% and 39% of the faulty implementations, respectively. In general, different transitions of the same method are not highly effective for the same number of non-conformant implementation, and therefore, they are not effective for the same set of implementations.

Summarizing, given a faulty implementation I, in general there is a transition t such that most of the times t is traversed, I is revealed as non-conformant. Additionally, the highly effective transition varies across different non-conformant implementations. These observations reinforce the proposed criterion. For a given faulty implementation, it is unknown which transitions are likely to expose it. The higher the adequacy level of a test suite, the higher the probability of covering the highly effective transition. What is more, by adopting a category partition approach – which implies selecting some from each subdomain – the highly effective ones are covered.

4.3. Research question 3

Each transition of an EPA is associated with a particular action in a many-to-one relation. For instance, in the EPA of the JDK 1.4 `Socket` implementation shown in Figure 4, there are two different transitions corresponding to the method `getOutputStream`. The difference between those states is that while in S_3 it is legal to invoke `getInputStream`, that invocation in S_4 would result in a protocol violation. Still, `getOutputStream` is enabled on both. It is easy to see that as

DOI: 10.1002/stvr

Table VIII. Correlation between transitions and actions with fault detection and structural coverage.

	Fault detection		Statement coverage		Branch coverage	
Subject	Transition	Action	Transition	Action	Transition	Action
`ListItr`	**0.84**	0.48	0.66	**1.00**	**0.70**	0.58
`Signature`	**0.73**	0.68	0.63	**0.79**	0.70	**0.75**
`Socket`	0.72	**0.76**	0.77	**0.93**	0.77	**0.89**
`JDBCResultSet`	0.68	**0.77**	**0.67**	0.60	**0.66**	0.57
`SMTPProcessor`	**0.69**	0.55	0.70	**0.92**	0.73	**0.76**

coverage metrics transitions properly covers [53] actions. The purpose of this research question is to understand the improvement of covering transitions over actions, if any.

4.3.1. Covering actions. In the subsequent discussion is an analysis of the effect of covering actions and compare the results with those of covering transitions. Table VIII shows Spearman's rank correlation coefficient values for both actions and transitions with fault detection and structural coverage.

With regard to fault detection, covering transitions perform better than covering actions in three out of five case studies. Additionally, the correlation values for transition coverage are more stable than those of action coverage.

In terms of structural coverage, action coverage perform better than transition coverage in seven out of 10 cases. This is not surprising because correlation measures how well the relationship between two variables can be described by a monotonic function. In the case of action and structural coverage, it measures to what extent is true that as more actions are executed more code is exercised. Every time that an action that had not been executed before is executed, some portion of the code that had not been exercised before is exercised. This observation does not hold for transition coverage, because covering two different transitions of the same action may not necessarily imply covering a new portion of the code.

The case of `ListItr` is particulary interesting. As already mentioned, it has a quite rich protocol and an extremely simple code structure. The correlation between action coverage and statement coverage is the best possible, whereas the one with fault detection is the lowest one. On the one hand, the poor performance of actions regarding fault detection indicates that just executing different actions is not enough for killing more mutants. On the other hand, covering more transitions (which implies executing actions from different enabledness states) has a positive impact on the number of mutants killed.

4.3.2. Actions versus transitions from a category partition perspective. The failure rates of the subdomains derived from covering actions is now analysed in the same way as was performed with transitions. A unit test u is considered to belong to the subdomain associated with an action a if and only if a is invoked in u. Note that, as with transitions, the partition may result in overlapping subdomains. For instance, the unit test of Figure 3 belongs to the subdomains associated to both actions $push$ and pop. Figure 9 shows boxplots of failure rates for the densest subdomains for both transitions and actions. Numerical values of percentiles, means and means excluding outliers are shown in Table IX.

Whereas in the case of transitions there is a revealing subdomain for at least 50% of non-conformant implementations, in the case of actions, this observation holds for only 25% of them. Still, failure rates values of actions' subdomains are acceptable: subdomains' failure rates for 50% of the mutants range (at least) between 0.78 and 1.00 depending on the case study.

Also in this case, the analysis restricting the universe of mutants to those that are hard to detect (i.e. that are killed by less than 20% of the unit tests) was performed. Boxplots are shown in Figure 10 and numerical values in Table X. Except for the case of `ListItr`, where clearly failure rate values of transitions' subdomains outperform those of actions' subdomains, results for actions and transitions do not present major differences.

DOI: 10.1002/stvr

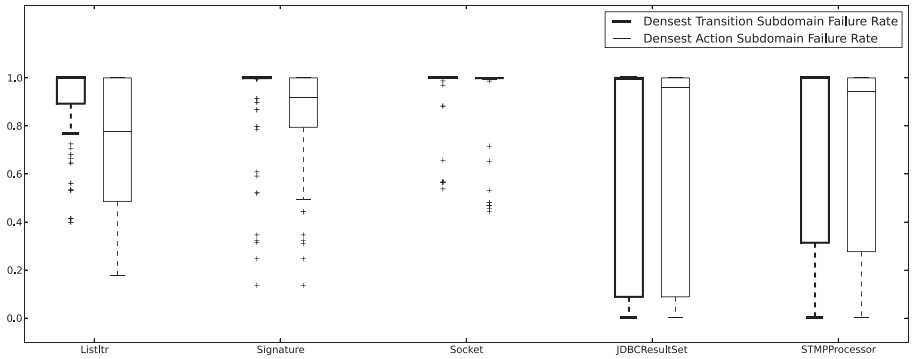


Figure 9. Failure rates of the densest subdomains defined by transitions and actions.

Table IX. Statistical info of failure rates considering subdomains derived from transitions and actions.

	By transition subdomains					By action subdomains				
Subject	Q_1	MED	Q_3	MEAN	MEAN EO	Q_1	Med	Q_3	Mean	Mean EO
`ListItr`	0.89	1.0	1.0	0.91	0.95	0.49	0.78	1.0	0.71	0.71
`Signature`	1.0	1.0	1.0	0.93	0.98	0.79	0.92	1.0	0.87	0.92
`Socket`	1.0	1.0	1.0	0.94	0.99	0.99	1.0	1.0	0.89	0.96
`JDBCResultSet`	0.09	1.0	1.0	0.67	0.67	0.09	0.96	1.0	0.65	0.65
`SMTPProcessor`	0.31	1.0	1.0	0.71	0.71	0.28	0.94	1.0	0.65	0.65

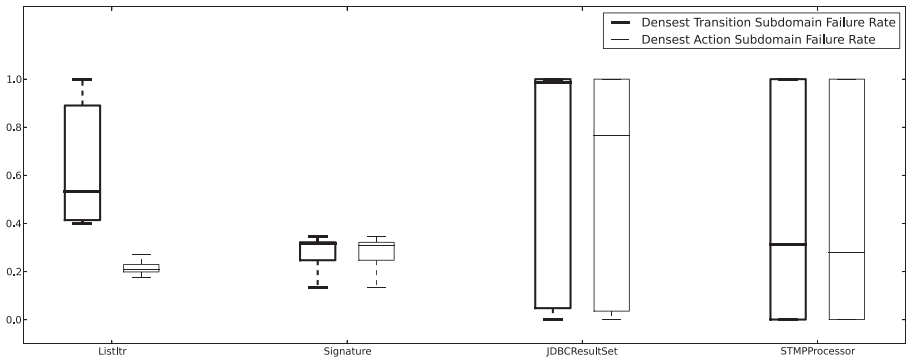


Figure 10. Failure rates of the densest subdomains defined by transitions and actions for hard mutants.

Table X. Statistical info on failure rate for hard mutants considering transition and action subdomains.

			By transition subdomains					By action subdomains				
Subject	#MUT	#HMUT	Q_1	MED	Q_3	MEAN	MEAN EO	Q_1	MED	Q_3	MEAN	MEAN EO
`ListItr`	145	14	0.41	0.53	0.56	0.62	0.62	0.2	0.21	0.21	0.21	0.21
`Signature`	96	5	0.25	0.31	0.32	0.27	0.27	0.25	0.31	0.32	0.27	0.27
`JDBCResultSet`	323	201	0.05	0.99	1.0	0.57	0.57	0.04	0.77	1.0	0.55	0.55
`SMTPProcessor`	259	176	0.0	0.31	1.0	0.49	0.49	0.0	0.28	1.0	0.46	0.46

The partition of inputs as studied in this section is, however, rather coarse grained for comparing actions as transitions. Consider, for instance, the unit test of Figure 3. A faulty implementation I may fail when *push* is invoked, but because the input belongs to the subdomains derived from both actions (*push* and *pop*), the killing of I is counted for both, although the *pop* statement is not even reached when executing the unit test on implementation I. A finer-grained perspective for evaluating differences between transitions and actions is developed in the following section.

DOI: 10.1002/stvr

4.3.3. Effectiveness of transitions versus effectiveness of actions. As with transitions, the effectiveness of an action a for an implementation I (notation $\text{eff}_{a,I}$) is defined as the ratio of the number of times that I is exposed as a non-conformant implementation when executing a and the total number of times that a is executed on I. Note that it is not possible to have an action with greater effectiveness to all its associated transitions.

Note also that there are some situations in which transitions cannot improve the effectiveness of actions. There are actions associated with only one transition. While this situation is not present neither in `SMTPProcessor` nor in `ListItr`, this is the case of five out of 14 actions of `Signature`, one out of 41 actions of `JDBCResultSet` and one out of seven actions of `Socket`. When the most effective transition is the only transition of some action, there is no distinction between their highest effectiveness. In all such cases, it is meaningless to study the distinction between transitions and actions. Besides, there are some faulty implementations such that there is an action a whose effectiveness equals 1 – that is, every time a is executed, the implementation is revealed as non-conformant. Again, no transition can improve actions in this case. Still, there are many implementations for which the described situations do not hold, and therefore, transitions may improve the effectiveness of actions. Table XI shows information regarding improvable and not improvable faulty implementations. The second column of the table specifies the number of mutations that resulted in non-conformant implementations; the third and fourth columns show the number of faulty implementation for which there is an action with effectiveness 1 and for which the most effective transition is the only transition of that action; the fifth and sixth columns expose the number of faulty implementation for which transitions cannot and can improve actions, respectively. Note that column 5 does not always correspond to the sum of columns 3 and 4, because there are implementations for which there is an action a of effectiveness 1 and a has only one associated transition. For instance, action `bind` of `Socket` has effectiveness 1 for 15 faulty implementations, and it has only one associated transition (Figure 4).

What happens where there is a chance for improvement? For each faulty implementation such that transitions could improve actions, the highest effectiveness of both is calculated, transitions and actions. Boxplots of Figure 11 shows the percentage of improvement of transitions over actions for

Table XI. Improvable and not improvable non-conformant implementations.

Subject	#Total	$\#\text{EFF}_{a,I} = 1$ (%)	#1A–1T (%)	#Filtered (%)	#Improvable (%)
`ListItr`	145	45 (31.03)	0 (0)	45 (31.03)	100 (68.97)
`Signature`	96	36 (37.5)	26 (27.08)	43 (44.79)	53 (55.21)
`Socket`	59	44 (74.58)	23 (38.98)	52 (88.14)	7 (11.86)
`JDBCResultSet`	259	116 (44.79)	0 (0)	116 (44.79)	143 (55.21)
`SMTPProcessor`	317	206 (64.98)	0 (0)	206 (64.98)	111 (35.02)

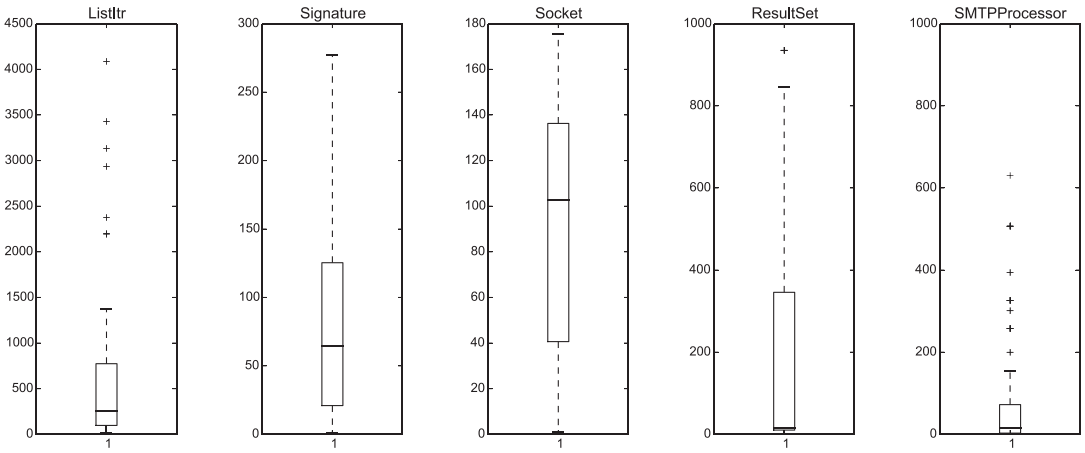


Figure 11. Improvements of transitions over actions.

DOI: 10.1002/stvr

Table XII. Numerical values of improvement of transitions over actions.

Subject	Q_1 (%)	Median (%)	Q_3 (%)	Mean (%)	Mean EO (%)
`ListItr`	97.25	260.23	773.77	708.52	350.49
`Signature`	20.98	64.34	125.37	81.32	81.32
`Socket`	32.96	102.52	103.43	90.39	90.39
`JDBCResultSet`	10.27	15.73	345.83	193.78	163.51
`SMTPProcessor`	2.02	6.27	128.16	110.10	54.76

the five case studies. Numerical values of Q_1, median, Q_3, mean and mean excluding outliers are shown in Table XII.

The level of improvement of transition over actions varies significantly across case studies. Nevertheless, the improvement is considerably high in all cases for a large number of non-conformant implementation. Mean values of improvements range between 81.32% and 708.52 and when excluding outliers between 54.76% and 350%. From this, it can be concluded that transitions are much more effective than actions. This observation reinforces the idea of covering transitions (instead of just executing actions) for revealing non-conformant implementations.

4.4. Research question 4

Results of RQ.3 indicate that there may be an opportunity at combining action and EPA transition coverages in order to define the level of adequacy of a test suite. This section defines and reports how such a combined level of adequacy performs for metrics studied in RQ.1 and RQ.2.

Definition 7 (EPA action–transition adequacy level)
Let L be a protocol LTS, A its EPA and TS a test suite. The action–transition adequacy level of TS with respect to A is defined as the ordered pair (j, k) where j indicates the percentage of actions and k the percentage of transitions of A that are covered by the α-abstracted execution of the unit tests of TS.

Two values are involved in this definition: number of actions invoked by a test suite and number of transitions that are exercised by the α-abstracted execution of the test suite. To calculate how this criterion correlates with fault detection and structural coverage, a lexicographical ordering of test suites is used.

Definition 8 (action–transition adequacy order)
Let L be a protocol LTS, A its EPA, and TS_1 and TS_2 two test suites with adequacy (j_1, k_1) and (j_2, k_2), respectively. If $j_1 > j_2$ or $j_1 = j_2$ and $k_1 > k_2$, then TS_1 has a greater level of adequacy than TS_2. If $j_1 = j_2$ and $k_1 = k_2$, both are equally adequate. Otherwise, TS_2 has greater level of adequacy than TS_1.

The rationale behind this combined criterion is that if an enabled action has not been executed, then choose to execute it. Covering a new action improves code coverage and additionally also implies to exercise a new transition. When all available actions have already been executed, then choose one such that its execution may cover a yet uncovered transition. This way, strengths of both transitions and actions are being exploited. Spearman's correlation values for fault detection and structural coverage are shown in Table XIII for the three criteria: action coverage (A), transition coverage (T) and the combined coverage criterion ((A+T)).

Regarding fault detection, in four out of five case studies, the combined criterion outperforms action and transition criteria. For the remaining case study (`Socket`), the correlation for the combined criterion is just 0.01 less than the best one for this subject. With respect to statement coverage, action coverage is the best in four out of five case studies. As mentioned before, this is not surprising considering that always that a yet unexecuted action is executed some yet unexercised statements are exercised (some of those corresponding to the action). More interestingly, in the case of branch coverage in three cases, criteria involving covering transitions have higher correlation values than those of action coverage. Executing different branches is often related to the internal state of objects. The

DOI: 10.1002/stvr

Table XIII. Correlation between transitions and actions with fault detection and structural coverage.

	Fault detection			Statement coverage			Branch coverage		
Subject	A+T	T	A	A+T	T	A	A+T	T	A
`ListItr`	**0.84**	**0.84**	0.48	0.71	0.66	**1.00**	0.50	**0.70**	0.58
`Signature`	**0.73**	**0.73**	0.68	0.78	0.63	**0.79**	**0.79**	0.70	0.75
`Socket`	0.75	0.72	**0.76**	0.74	0.77	**0.93**	0.70	0.77	**0.89**
`JDBCResultSet`	**0.80**	0.68	0.77	**0.78**	0.67	0.60	**0.91**	0.66	0.57
`SMTPProcessor`	**0.69**	**0.69**	0.55	0.69	0.70	**0.92**	0.66	0.73	**0.76**

existence of different transitions of the same action in the EPA also reflects differences on internal object states. Therefore, covering more transitions implies executing actions from distinct internal states, which in turn increases the chances of covering more branches.

4.5. Summary of results

Results suggest that the EPA transition criterion is a good predictor of mutant detection (RQ.1.1) with correlations between 0.68 and 0.84 depending on the case study. Results also show that EPA transition correlations are comparable with or better than those of structural (rather than behavioural) white box criteria. To avoid benefitting from bias in the selection of the specification language and model to be structurally covered, the (unmutated) code itself was chosen as the specification. This choice does not favour (on the contrary) the hypotheses proposed in this paper as the code can be considered as the most detailed specification and most likely constitutes an upper bound on what structural criteria on specifications can achieve as test suite quality predictors. Results suggest that the EPA transition coverage criterion, which is behavioural and hence independent of specification language bias, performs comparably in terms of predictability of test suite fault detection, resulting in higher values for all case studies against statement coverage and is better than branch coverage in two out of five case studies.

Results also suggest that for fixed size, test suites with the highest behavioural adequacy are statistically better (RQ.1.2). In all the five case studies, the test suites were divided in 10 different bins grouping those of similar size. Mann–Whitney U hypothesis test was performed, and in 39 out of 50 bins, there is statistical evidence that those of higher adequacy perform better. Those for which no such evidence exists correspond to bins of the largest test suites. As expected, the impact of EPA transition coverage diminishes as size increases. Additionally, to obtain a quantitative measure of this fact, the Vargha–Delaney A_{12} effect size was calculated, which measures the difference between two populations in terms of the probability that a score sampled at random from the first population will be greater than a score sampled at random from the second.

Results for RQ.2 aiming at getting a deeper understanding of the observed phenomena reinforce the results of RQ.1. Results indicate that the proposed criterion is a good predictor of structural coverage criteria (statement and branch coverage) when applied over code under test (RQ.2.1). Correlation of EPA transition coverage and branch coverage are all 0.7 or above except for one case study which yielded 0.67. It is also shown that for fixed size, test suites with the highest behavioural adequacy are statistically better in terms of structural coverage (RQ.2.1); however, as expected, the impact of EPA coverage diminishes as size increases. These results suggest that a first and early shot at producing high code coverage test suites (which could then be extended if necessary when code is available) can be achieved through EPA transition coverage.

Results also indicate that the domain partition implicitly derived from EPA transitions is likely to produce subdomains that are dense in failures, that is, a subdomain with high failure rate (RQ.2.2). This is close to what is known to be optimal in the context of partition testing [30].

More specifically, in all case studies for at least 50% of the faults, partitioning subdomains according to EPA transitions produces a 100% revealing subdomain, and that for 75% of the faults, partitioning subdomains results in at least a sixfold failure rate (i.e. density) improvement over the entire domain. Average (eliminating outliers) density per best subdomain is also high ($> 67\%$) and significantly better than for the full domain. Looking only at the hardest faults, for 50% of the faults,

DOI: 10.1002/stvr

subdomains start from 31% revealing and are at least a twofold (but up to two orders of magnitude) improvement over the complete domain. The Q_1 percentile and average without outliers also shows a significant improvement.

A finer-grained study (RQ.2.3), investigating effectiveness of transitions (the likelihood of revealing a mutant when traversing the transition) rather than the subdomains defined by them reveals that for each fault there is almost always one transition that is highly effective in detecting it ($> 90\%$ effectiveness), while nearly all the rest of the transitions have poor effectiveness ($< 10\%$). Furthermore, it is shown that the effective transition is not always the same one; it depends on the fault. This is in line with previous results: as faults are a priori unknown, it makes sense to consider more adequate those tests that cover more transitions. Test suites with highest adequacy cover all transitions, and consequently, the one highly effective is exercised. However, a more significant implication is that, as uniqueness of effective subdomain per fault does not hold but does for transitions, there is an opportunity to improve on EPA transition coverage criterion.

Experimentation regarding RQ.3 aimed at investigating that promising results for EPA transition coverage are not simply because the criterion is a refinement of action coverage provides positive indicators. Results suggest that for correlation with fault detection and structural coverage, in most cases, EPA transition coverage performs comparably or better than action coverage.

Also, results indicate that subdomains defined by actions are less dense than those derived from transitions. The most interesting results arise when comparing effectiveness of individual transitions and actions; here, the former significantly improves the latter. The average after eliminating outliers shows improvements starting at 50% and up to 350%; while looking at percentiles, for 50% (resp. 75%) of faults transition, effectiveness improves over action effectiveness in four out of five case studies between 15% and 260% (resp. 10% and 97%). The last case study shows a smaller improvement 6% (resp. 2%). These results suggest that EPA transition coverage can provide benefits over action coverage and justify studying a combined criterion.

The last set of results (corresponding to RQ.4) shows that a coverage criterion based on actions and transitions outperforms the criteria of those of actions and transitions (when considered in isolation) as a fault detection predictor in all the case studies except for one, in which the highest correlation is only 0.01 greater. Values for the combined criterion range between 0.69 and 0.84.

In conclusion, this paper is a step to understanding how behavioural coverage of protocol behaviour correlates with fault detection in the context of call protocol conformance testing. Results show, to the extent of the external validity threats of the experiments, that the proposed mixed criterion is a good predictor for fault detection and for classical structural coverage criteria such as statement and branch coverage.

The implication being that the criterion could help improve random testing, test-driven development, test case selection and, in general, techniques for tests generation from formal specifications when applied to API code with rich protocols. The results seem to indicate that such approaches would benefit from introducing heuristics that aim to maximize action and EPA coverage.

5. THREATS TO VALIDITY

The results presented in this paper are subject to threats to validity including internal, external and construct validity threats.

Threats to external validity concern the ability to generalize the results. The proposed approach is not suitable for testing code which has trivial requirements regarding the order in which methods or procedures must be called. Furthermore, even fixing the code to that which has rich intended protocols, the results cannot be generalized to identification of faults other than the ones captured by the failure model embedded in the notion of protocol conformance discussed in this paper. The expectation is that the results can be generalized to classes featuring rich intended protocols and faults that are expressed as unexpected occurrence of exceptions or non-terminating methods. However, the study presented only covers five subjects which may not be sufficiently representative of rich protocol code artefacts. Moreover, restricting faulty implementations to those that unexpectedly raise exceptions or timeout is limitating, because APIs may fail in other ways too. Still, despite its

DOI: 10.1002/stvr

weakness, it is still considered relevant by the software engineering community in typestate verification [4, 31–33] and concurrent systems [54, 55].

A threat to the generalization of the results to the breadth of EPA coverages is that for some subjects, high EPA coverage was not achieved by any test suite (maximum coverage for one test suite achieved in subjects are 65%, 78%, 90%, 56% and 100% for `SMTPProcessor`, `ListItr`, `Socket`, `JDBCResultSet` and `Signature`, respectively). Not achieving very high coverage is due, firstly, to the use of random generation of test units which makes reaching 'deep' EPA states unlikely. Possibly, a much larger number of much longer test units could have achieved more coverage. Secondly, there is an essential difficulty in reaching deep states due to the complexity of code under test. In fact, these problems appear when trying to achieve high structural coverage in general [2] and for the subjects chosen in particular. Guided test case generation could have been used to address this issue; however, the threat of a biased pool of test suites would have increased.

Another threat to generalization is due to the choice of restricting the scope of this work to deterministic APIs. Extending the work to implementations that may exhibit non-deterministic failures would require to revisit the coverage criteria and the conformance relation. Besides, it would certainly require a more sophisticated experimental strategy. If non-deterministic API implementations were included, the notion of test effectiveness (killing a mutant) would require probabilistic treatment because a non-deterministic mutant could behave differently in terms of failing or not to accept a given sequence of calls. Therefore, each test would have a probability of killing each mutant.

Threats to internal validity appear as a consequence of how the experiments were conducted. One major threat is the validity of the EPAs of the actual protocol LTS for the subjects studied. The use of LTS that do not abstract appropriately the behaviour of the subject implementations could lead to skewed results regarding coverage (although not for detecting faults, as for this the reference implementation itself and not its abstraction is used). The risk of having used models that are not proper abstractions of the subjects (i.e. not EPAs) is mitigated by our systematic construction process, validation against third party constructed models and manual inspections performed.

Another threat to internal validity appears as a consequence of the adopted fault model. The faults are introduced using a mutation tool. Whether mutation analysis is a realistic fault model or not is still open and beyond the scope of this paper. Nevertheless, some work has shown statistical evidence of the validity of using mutation analysis for the evaluation of testing techniques [56], and it is a very widespread approach for evaluating testing techniques.

Another threat to internal validity is due to the adopted failure model. Incorrect computations can go unnoticed and exceptions can be triggered later: this is a problem that affects unit test in general and not something particular of this work. The goal is, in this case, to see if the sequences generated according to the proposed criteria produce inputs that not only reach the state infection but also its manifestation. It could be the case, that reduced oracle power might worsen this general phenomena and thus imply the need of longer test cases to detect manifestation but this is mitigated by using a set of long unit tests.

As with other experiments using mutants and unit tests, the threat of using weak tests that fail to identify fault-inducing mutants exists. This could lead to different results if these harder to kill mutants were included. However, the correlations over subsets of mutants found were analysed, and in particular, those least killed showed no significant changes in correlation.

Threats regarding unintended effects of general experimental infrastructure needed for mutant generation and fault detection are minor because standard tools such as μ-JAVA and RANDOOP were used whenever possible and simple code instrumentation techniques using AspectJ to record mutant detection.

Threats to construct validity mainly appear from the choice to compare the proposed criterion with code coverage criteria. Code coverage as a measure of effectiveness of a test suite is still being studied by the testing community. However, in order to asses if the correlations with fault detection for EPA coverage are reasonable, some well-accepted baseline is needed, and structural code coverage criteria were chosen. However, the comparisons with code coverage in the experiments were complemented with the study of EPA coverage against fault detection.

On the other hand, the proposed criteria are black box and they are compared with what in principle could be considered white box criteria in RQ1 (i.e. structural code coverage). It could be argued

DOI: 10.1002/stvr

that a more suitable baseline would be some other black box criteria such as structural coverage over a specification. The question here would be what specification language would be suitable. Given that the intended protocol LTS of the subjects that are of interest for this study are infinite state, a rich specification language such as extended finite state machines (essentially, LTS with variables) as used in other works [11, 12] is required or even combined with programming languages such as C# or Java – as used in model-based testing approaches such as SpecExplorer [57] and ConformiQ [9]. Unfortunately, there is no de facto standard black box baseline for rich modelling languages and any choice of language, tool and specification style will introduce bias as it is known that specification structure can have significant impact on coverage criteria adequacy [16–21]. Taking the most detailed specification (the code itself) constitutes an upper bound on what structural criteria on specifications can achieve as test-suite quality predictor. In fact, it is highly likely that in model-based development, there will be more structural discrepancy between specification and code under test. Hence, it could be argued that choosing the code as the specification hinders validation of the hypothesis being proposed in this paper.

6. RELATED WORK

6.1. Models for testing

Generation of test sequences for software with internal state has captured the attention of researchers in recent years [27, 28, 58–60]. Those works aim at generating internal states that improve structural coverage by constructing a set of test sequences. Pure random testing approaches like the one of Pacheco *et al.* [44] are based on reusing previously returned values. Those approaches are truly black box and thus are applicable to conformance testing and offline test generation. In fact, the work of Pacheco *et al.* [44] constitutes the baseline for the experiments conducted. Evolutionary approaches [58] represent initial randomly generated method sequences as a population of individuals and evolve this population by mutating its individuals until a desirable set of method sequences is found. Although they do not use program structure or semantic knowledge to directly guide test generation, they do require coverage achieved by test sequences on the system under test (SUT) for computing the fitness function. This makes them less portable to the problem setting studied in this paper. Recent genetic techniques instead aim at finding settings to parameters which control aspects of randomized testing [59]. The systematic and randomized method of Inkumsah *et al.* [60] is even less applicable to the problem setting because it requires access to the program code of SUT to generate inputs by demand by the use of some symbolic computation approaches.

Much research effort on testing has focused mainly on test case generation by exploiting to various degrees the code-under-test: from purely systematic white box approaches [61]) to search-based approaches [62] in which fitness functions are based on achieved coverage. None of these approaches can tackle conformance checking to its full extent: they are not driven by any form of actual or intended behaviour. However, some works explicitly or implicitly define or mine models to improve the quality of tests. For instance, in the work of Liu *et al.* [27] the state space of a class is quotiented based on its parameterless boolean observers. Visser *et al.* [28] proposed an approach where abstract states are computed using shape abstraction, that is, ignoring the concrete values in containers and taking into account only the shape in which the container nodes are connected. Note that this work requires access to the internal state of the SUT. In a work of Dallmeier *et al.* [26], a type-state model – similar to our EPA – is inferred and used to guide the generation of new test cases that try to cover uncovered transition of the type state. The goal is to dynamically discover typestate models. The quality of such models is then measured in the context of detecting misuse of the class protocol by client programs.

The work presented herein differs substantially in various ways: The mentioned approaches do not (i) look at the problem of black box conformance testing; (ii) articulate equivalence criteria declaratively as an adequacy criterion (rather, they are tightly coupled to the particular technique); (iii) provide statistical evidence on the effectiveness of covering such abstractions in terms of fault rate detection.

In previous work [63], it was shown that the EPA transition coverage criterion is a good predictor for fault detection and structural coverage. This paper extends it in several ways. Here, the results

DOI: 10.1002/stvr

are studied from a category partition perspective and, as shown, the partition of the domain according to the criterion produces dense subdomains. Also, the effectiveness of transitions for detecting conformance faults is studied and it is shown that in most cases, given a faulty implementation, there is a transition highly effective for detecting it. The results of covering EPA transitions are compared with those of covering actions, and results indicate that the former outperforms the latter. Finally, in this paper, a new criterion is defined which involves covering both, actions and transitions, and it is even more effective than previously defined ones in terms of fault detection predictability.

6.2. Conformance testing

There has been plenty of work focused on defining coverage from formal specifications and models [3]. Works range on a plethora of languages: algebraic specifications [64–66], Z [13], VDM [14], boolean specifications [67], tabular notations [68], reactive modelling language like SDL [69], EFSM [12, 16, 70], sequence charts [71] temporal logics [15, 72, 73], C# [57], ADL [74], simulation code [17], and so on. The rest of the section focuses the discussion on approaches able to straightforwardly aim at testing object protocols.

Existing approaches can be classified in two categories: structural (or specification-based) and behaviour (or semantic) coverage criteria. Structural coverage criteria are either defined in terms of the specification [11–15, 69] or of the executable code generated from the specification or simulation model [16, 17]. There are many representative examples [11, 12, 20] where criteria are defined over syntactic elements as transitions and predicates featured in expressive state-based specification languages like EFSMs or UML state machines.

Although empirical studies looking for statistical evidence on the suitability of coverage criteria are rather common for code coverage criteria [75] yet are scarce for state-based specification languages [3] which are particularly appropriate for conformance testing. Nevertheless, there are some notable exceptions [16, 17, 20]. Interestingly enough, in these, experiments were conducted on criteria based on covering code generated from models [16] or simulation code [17]. In the work of Mouchawrab *et al.* [20] handmade test suites based on UML state machines are compared against test suites based on structural testing. As mentioned in the work of Pretschner *et al.* [16], it may be the case that difficulties on automation criteria over specification could be a symptom of a lack of comprehensive definitions and tools for specification-based coverage criteria for rich state-based languages. The work presented herein is a step forward in this direction.

On the other hand, in behavioural approaches, coverage is defined in terms of formalisms which straightforwardly denote the intended protocol behaviour. This line of work is that of seminal work on black box testing in the context of finite state machine and protocol testing [11, 12, 20, 22–25, 35]. In foundational work, the conformance problem is stated in terms of Mealy machines. However, in contrast to the approach presented in this paper, coverage and failure models assume finiteness of both the specification and the actual implementation.

A work that drops the finiteness assumption is that of LTS-based testing [76] where IOCO is a well-established notion of conformance. In IOCO, it is required that for any valid sequence (according to the specification), the implementation's outputs after the sequence should be a subset of the specified outputs. In the presented approach, if {`OK, ERROR`} are considered as valid outputs, then the notion of conformance is the same as the one used in IOCO. If the specification says `OK`, then the implementation must say `OK` too; and if the specification says `ERROR`, also, the implementation should do the same. However, no notion of behaviour coverage has been defined in this setting of infinite state space.

A way of dealing with infinite behaviour models is by introducing abstraction. This is the case for some application domains of testing, like protocol testing [77] and architecture-based testing [74], where the level of abstraction in which the designer expresses the specification leads to the finiteness of underlying LTS semantics. However, if the underlying semantic model is truly infinite, then some sort of finitization is made available to the tester and so on. Several finitization techniques exist: unfolding [69], domain bounding and slicing [6, 57], and state pruning [6]. However, no statistical studies on coverage for these finitizations are available. Other relevant work in this line is that of

DOI: 10.1002/stvr

automatic under approximation of infinite behaviour from concrete [78] or symbolic [79] executions that can be later used for regression testing. Here, again, no statistical study is available.

Mori *et al.* [37] use the term 'cover' in the topological sense, defining a finite number of subspaces that cover an infinite metric space. Covering (in the testing sense) a representative of each of these subspaces amounts to defining a coverage criterion over an abstraction of the infinite state space, which is what is also performed in this paper. As discussed in the paper, the positive impact of partitioning an input space to defining coverage criteria is highly dependent on how failures fall within these partitions. Thus, experimental evaluation of coverage criteria is important. We have not found experimental evaluation of the predictive power of criteria defined over finite abstractions of infinite input spaces for API call protocols.

Summarizing, the approach studied statistically in this paper fits in the category of behaviour or semantic approaches to conformance testing in which infinite state space is handled by means of an abstraction that over-approximates the state space much in the vein of typestates [34]. This abstraction produces a partition of states and transitions thus constituting a case of category partition [80] for infinite LTS.

7. CONCLUSIONS AND FUTURE WORK

This paper is a step towards defining and understanding how semantic coverage of infinite state behaviour specifications relates to effective testing techniques for call protocol conformance. This was addressed by studying coverage achieved on an abstraction of such behaviour, more specifically on EPAs (much in the vein of typestates [34]). A good understanding of the relation between fault detection, white box coverage criteria and coverage of abstractions of the semantic space of specifications could help to improve random testing, test case selection and prioritization techniques and, in general, heuristics to generate tests from formal specifications.

The results obtained in the experiments reported in this paper are promising and suggest that EPA coverage performs well in term of predictability of test-suite fault detection. This is particularly important in a black box testing setting, and it constitutes an opportunity for defining criteria that are independent of modelling notation and accidental characteristics of models themselves. Results also suggest that EPA coverage criterion can make a difference in terms of fault detection for tests suites of the same length.

In addition, results lead to believe that EPA coverage is a good predictor of statement and branch coverage and that, for same-sized test suites, high EPA coverage is more likely to achieve high code coverage. This may have practical implications in the context of development approaches which advocate test development before coding (e.g. test driven development, interoperability) or automated generation of test suites (model-driven development). In these contexts, developing tests according to the EPA of the intended protocol would allow a first (and early) shot at producing high code coverage test suites. These test suites could later be extended, if necessary, when code is available. As far as we know, existing approaches for generating high code coverage test suites are all white box, and therefore, they typically need the source code to be applied. It is important to note that the construction of EPA abstractions of intended protocol behaviour is feasible, practical and tool supported from contract-based specifications [36] or code [29]. Furthermore, EPA abstractions could be provided directly by testers as advocated by typestate approaches [4].

Besides, results show that the domain partition implicitly derived from EPA transitions tends to produce subdomains that are dense in failures, which is near to an ideal scenario from a category partition viewpoint [30]. Furthermore, for each fault there is almost always one transition that is highly effective in detecting it and that the effective transition varies from fault to fault. Additionally, it is shown that the positive results are not just a consequence of covering actions, and that when both actions and transitions are taken into account, the resulting criterion has highest correlation values regarding fault detection.

The results could also be the basis for arguing for test case generation strategies based on abstractions of behaviour rather than code itself (which requires significantly more heavyweight machinery). In addition, they also make a case for stating that test case generation for rich call pro-

DOI: 10.1002/stvr

tocol artefacts (from random black box to sophisticated concolic white box techniques) can benefit from the heuristics that are implied by the results discussed in this paper.

Future work should aim at looking at other protocol abstractions and comparing them with EPAs in terms of their effectiveness for testing protocol conformance. Besides, we want to conduct experiments in a more general setting that includes both APIs with non-deterministic expected behaviour and output values as part of the conformance relation considered. We also plan to study cost/benefit analysis when these ideas are instantiated to a guide the generation of test suites. In fact, we speculate that random generation could benefit from EPAs not only due to the results shown in this paper but also the availability of an abstract protocol would help in implementing heuristics aimed at the early execution of particular actions or functionalities.

ACKNOWLEDGEMENTS

The work reported herein was partially supported by ANPCyT PICT 2012-0724, ANPCyT PICT 2011-1774 and 2013-2341, UBACYT 0384 and MEALS 295261.

REFERENCES

1. Frase G, Arcuri A. A sound empirical evidence in software testing. *ICSE '12,* Zürich, Switzerland, 2012; 178–188.
2. Xiao X, Xie T, Tillmann N, de Halleux J. Precise identification of problems for structural test generation. *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, Honolulu, Hawaii, USA, 2011; 611–620.
3. Hierons RM, Bogdanov K, Bowen JP, Cleaveland R, Derrick J, Dick J, Gheorghe M, Harman M, Kapoor K, Krause P, Lüttgen G, Simons AJH, Vilkomir S, Woodward MR, Zedan H. Using formal specifications to support testing. *ACM Computing Surveys* 2009; **41**:9:1–9:76.
4. Beckman NE, Kim D, Aldrich J. An empirical study of object protocols in the wild. *ECOOP '11,* Lancaster, UK, 2011; 2–26.
5. Groce A, Zhang C, Eide E, Chen Y, Regehr J. Swarm testing. *ISSTA '12,* Minnesotaj, Minneapolis, USA, 2012; 78–88.
6. Grieskamp W, Kicillof N, Stobie K, Braberman V. Model-based quality assurance of protocol documentation: tools and methodology. *STVR* 2011; **21**(1):55–71.
7. DeLine R, Fähndrich M. Typestates for objects. *ECOOP '04,* Oslo, Norway, 2004; 465–490.
8. Selic B. The pragmatics of model-driven development. *IEEE Software* 2003; **20**(5):19–25.
9. Utting M, Legeard B. *Practical Model-based Testing: A Tools Approach*. Morgan Kaufmann: San Francisco, CA, USA, 2007.
10. Beck K. *Test-driven Development: By Example*. Addison-Wesley Professional: Boston, MA, USA, 2003.
11. Korel B, Koutsogiannakis G, Tahat LH. Model-based test prioritization heuristic methods and their evaluation. *A-MOST '07,* London, UK, 2007; 34–43.
12. Offutt J, Liu S, Abdurazik A, Ammann P. Generating test data from state-based specifications. *STVR* 2003; **13**(1): 25–53.
13. Hierons RM. Testing from a Z specification. *STVR* 1997; **7**(1):19–33.
14. Dick J, Faivre A. Automating the generation and sequencing of test cases from model-based specifications. *FME '93: Industrial-Strength Formal Methods.* LNCS, Odense, Denmark, 1993; 268–284.
15. Pecheur C, Raimondi F, Brat G. A formal analysis of requirements-based testing. *ISSTA '09,* Chicago, Illinois, USA, 2009; 47–56.
16. Pretschner A, Prenninger W, Wagner S, Kühnel C, Baumgartner M, Sostawa B, Zölch R, Stauner T. One evaluation of model-based testing and its automation. *ICSE '05,* St. Louis, Missouri, USA, 2005; 392–401.
17. Rutherford MJ, Carzaniga A, Wolf AL. Evaluating test suites and adequacy criteria using simulation-based models of distributed systems. *TSE* 2008; **34**:452–470.
18. Heimdahl MPE, George D, Weber R. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? *HASE '04,* Tampa, Florida, USA, 2004; 178–186.
19. Rothermel G, Harrold MJ, Ostrin J, Hong C. An empirical study of the effects of minimization on the fault detection capabilities of test suites. *ICSM '98,* Bethesda, MD, USA, 1998; 34–43.
20. Mouchawrab S, Briand LC, Labiche Y, Di Penta M. Assessing, comparing, and combining state machine-based testing and structural testing: a series of experiments. *TSE* 2011; **37**(2):161–187.
21. Weissleder S. Simulated satisfaction of coverage criteria on uml state machines. *ICST '10,* Paris, France, 2010; 117–126.
22. Drira K, Azéma P, Soulas B, Chemali AM. Characterizing and ordering errors detected by conformance testing. *IWPTS '92,* Montreal, Canada, 1993; 67–78.
23. Miller RE, Chen DL, Lee D, Hao R. Coping with nondeterminism in network protocol testing. *TESTCOM '05,* Montreal, Canada, 2005; 129–145.

DOI: 10.1002/stvr

24. Cihan Yalcin M, Yenigun H. Using distinguishing and uio sequences together in a checking sequence. *TESTCOM '06,* New York, NY, USA, 2006; 259–273.
25. Hierons RM. Testing from a nondeterministic finite state machine using adaptive state counting. *IEEE Transactions on Computers* 2004; **53**(10):1330–1342.
26. Dallmeier V, Knopp N, Mallon C, Hack S, Zeller A. Generating test cases for specification mining. *ISSTA '10,* Trento, Italy, 2010; 85–96.
27. Liu L, Meyer B, Schoeller B. Using contracts and boolean queries to improve the quality of automatic test generation. *TAP '07,* Zurich, Switzerland, 2007; 114–130.
28. Visser W, Păsăreanu CS, Pelánek R. Test input generation for java containers using state matching. *ISSTA '06,* Portland, Maine, USA, 2006; 37–48.
29. de Caso G, Braberman V, Garbervetsky D, Uchitel S. Program abstractions for behaviour validation. *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, Honolulu, Hawaii, USA, 2011; 381–390.
30. Jeng B, Weyuker E. Some observations on partition testing. *ACM SIGSOFT Software Engineering Notes* 1989; **14**(8):38–47.
31. Field J, Goyal D, Ramalingam G, Yahav E. Typestate verification: abstraction techniques and complexity results. *Science of Computer Programming* 2005; **58**(1-2):57–82.
32. Fink SJ, Yahav E, Dor N, Ramalingam G, Geay E. Effective typestate verification in the presence of aliasing. *TOSEM* 2008; **17**(2):9:1–9:34.
33. Bierhoff K, Aldrich J. Modular typestate checking of aliased objects. *SIGPLAN Notice* 2007; **42**(10):301–320.
34. Bierhoff K, Beckman NE, Aldrich J. Practical API protocol checking with access permissions. *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*: Genoa, 2009; 195–219.
35. Lee D, Yannakakis M. Principles and methods of testing finite state machines – a survey. *Proceedings of the IEEE* 1996; **84**(8):1090–1123.
36. de Caso G, Braberman V, Garbervetsky D, Uchitel S. Validation of contracts using enabledness preserving finite state abstractions. *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, Vancouver, British Columbia, Canada, 2009; 452–462.
37. Mori M, Vuong ST. On finite covering of infinite spaces for protocol test selection. *PSTV '94,* Hyderabad, India, 1995; 237–251.
38. Inozemtseva L, Holmes R. Coverage is not strongly correlated with test suite effectiveness. *ICSE '14,* Hyderabad, India, 2014; 435–445.
39. Andrews JH, Briand LC, Labiche Y, Namin AS. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 2006; **32**(8):608–624.
40. Andrews JH, Zhang Y. General test result checking with log file analysis. *IEEE Transactions on Software Engineering* 2003; **29**(7):634–648.
41. Briand LC, Labiche Y, Wang Y. Using simulation to empirically investigate test coverage criteria based on statechart. *ICSE '04,* Edinburgh, Scotland, United Kingdom, 2004; 86–95.
42. Caso GD, Braberman V, Garbervetsky D, Uchitel S. Enabledness-based program abstractions for behavior validation. *ACM Transactions on Software Engineering and Methodoly* 2013; **22**(3):25:1–25:46.
43. Henzinger TA, Jhala R, Majumdar R. Permissive interfaces. *ACM Sigsoft Software Engineering Notes* 2005; **30**: 31–40.
44. Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-directed random test generation. *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, Minneapolis, Minnesota, USA, 2007; 75–84.
45. Ma YS, Offutt J, Kwon YR. Mujava: an automated class mutation system: research articles. *Software Testing Verification and Reliability* 2005; **15**(2):97–133.
46. Offutt AJ, Pan J. Automatically detecting equivalent mutants and infeasible paths. *Software Testing Verification and Reliability* 1997; **7**(3):165–192.
47. Gligoric M, Groce A, Zhang C, Sharma R, Alipour MA, Marinov D. Comparing non-adequate test suites using coverage criteria. *ISSTA '13,* Lugano, Switzerland, 2013; 302–313.
48. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *TSE* 2011; **37**(5):649–678.
49. Namin AS, Andrews JH. The influence of size and coverage on test suite effectiveness. *ISSTA '09,* Chicago, IL, USA, 2009; 57–68.
50. Andrews JH, Groce A, Weston M, Xu RG. Random test run length and effectiveness. *ASE '08,* L'Aquila, Italy, 2008; 19–28.
51. Arcuri A, Briand LC. A practical guide for using statistical tests to assess randomized algorithms in software engineering. *ICSE '11,* Honolulu, Hawaii, USA, 2011; 1–10.
52. Weyuker EJ, Ostrand T. Theory of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering* 1980; **6**:236–246.
53. Frankl PG, Weyuker EJ. A formal analysis of the fault-detecting ability of testing methods. *TSE* 1993; **19**:202–213.
54. Schneider S. *Concurrent and Real Time Systems: The CSP Approach*, 1st edn. John Wiley & Sons, Inc.: New York, NY, USA, 1999.
55. Magee J, Kramer J. *Concurrency – State Models and Java Programs*, 2nd edn. Wiley: New York, NY, USA, 2006.
56. Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments?. *ICSE '05,* St. Louis, Missouri, USA, 2005; 402–411.
57. Veanes M, Campbell C, Grieskamp W, Schulte W, Tillmann N, Nachmanson L. Model-based testing of object-oriented reactive systems with Spec explorer. *Formal methods and testing*, LNCS, vol. 4949, 2008; 39–76.

DOI: 10.1002/stvr

58. Tonella P. Evolutionary testing of classes. *Proceedings of the 2004 ACM Sigsoft International Symposium on Software Testing and Analysis*, ISSTA '04, Boston, Massachusetts, USA, 2004; 119–128.
59. Andrews JH, Menzies T, Li FCH. Genetic algorithms for randomized unit testing. *IEEE Transactions on Software Engineering* 2011; **37**(1):80–94.
60. Inkumsah K, Xie T. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, L'Aquila, Italy, 2008; 297–306.
61. Thummalapenta S, Xie T, Tillmann N, de Halleux J, Su Z. Synthesizing method sequences for high-coverage testing. *OOPSLA '11,* Portland Oregon, USA, 2011; 189–206.
62. Sharma R, Gligoric M, Arcuri A, Fraser G, Marinov D. Testing container classes: random or systematic? *FASE '11,* Saarbrücken, Germany, 2011; 262–277.
63. Czemerinski H, Braberman V, Uchitel S. Behaviour abstraction coverage as black-box adequacy criteria. *Proceedings of the 6th International Conference on Software Testing, Verification, and Validation*, ICST '13, Luxembourg, 2013; 222–231.
64. Gannon J, McMullin P, Hamlet R. Data abstraction, implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*; **3**:211–223.
65. Doong RK, Frankl PG. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology* 1994; **3**:101–130.
66. Bernot G, Gaudel MC, Marre B. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal* 1991; **6**(6):387–405.
67. Weyuker E, Goradia T, Singh A. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering* 1994; **20**(5):353–363.
68. Gargantini A, Heitmeyer C. Using model checking to generate tests from requirements specifications. In *Software Engineering - ESEC/FSE '99*, vol. 1687, Nierstrasz O, Lemoine M (eds), Lecture Notes in Computer Science. Springer: Berlin / Heidelberg, 1999; 146–162.
69. Bochmann G, Petrenko A, Bellal O, Maguiraga S. Automating the process of test derivation from SDL specifications. *SDL Forum '97,* Evry, France, 1997; 261–276.
70. Petrenko A, Boroday S, Groz R. Confirming configurations in EFSM testing. *IEEE Transactions on Software Engineering* 2004; **30**:29–42.
71. Dan H, Hierons RM. Conformance testing from message sequence charts. *2008 International Conference on Software Testing, Verification, and Validation*, Vol. 0, Berlin, Germany, 2011; 279–288.
72. Ammann PE, Black PE. A specification-based coverage metric to evaluate test sets. *HASE 99*, Vol. 0, Washington, DC, USA, 1999; 239.
73. Hong H, Lee I, Sokolsky O, Ural H. A temporal logic based theory of test coverage and generation. *Tools and Algorithms for the Construction and Analysis of Systems,* Grenoble, France, 2002; 151–161.
74. Muccini H, Inverardi P, Bertolino A. Using software architecture for code testing. *TSE* 2004; **30**(3):160–171.
75. Frankl PG, Weiss SN. An experimental comparison of the effectiveness of branch testing and data flow testing. *TSE* 1993; **19**:774–787.
76. Tretmans J. Conformance testing with labelled transition systems: implementation relations and test generation. *Computer Networks and ISDN Systems* 1996; **29**(1):49–79.
77. Bochmann GV, Petrenko A. Protocol testing: review of methods and relevance for software testing. *ISSTA '94,* Seattle, WA, USA, 1994; 109–124.
78. Mariani L, Pezzè M, Riganelli O, Santoro M. Autoblacktest: a tool for automatic black-box testing. *ICSE '11,* Honolulu, Hawaii, USA, 2011; 1013–1015.
79. Grieskamp W, Gurevich Y, Schulte W, Veanes M. Generating finite state machines from abstract state machines. *ISSTA '02,* Roma, Italy, 2002; 112–122.
80. Ostrand TJ, Balcer MJ. The category-partition method for specifying and generating functional tests. *Communications of the ACM* 1988; **31**(6):676–686.

DOI: 10.1002/stvr