2013 IEEE Sixth International Conference on Software Testing, Verification and Validation

Behaviour Abstraction Coverage as Black-Box Adequacy Criteria

Hernán Czemerinski*, Victor Braberman*
*Departamento de Computación
FCEyN, Universidad de Buenos Aires
{hczemeri,vbraber, suchitel}@dc.uba.ar

Sebastián Uchitel*†
†Department of Computing
Imperial College London
s.uchitel@doc.ic.ac.uk

***Abstract*—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 in that they conform to their intended *protocols* is an important and challenging problem. In this paper we propose and study experimentally conformance testing adequacy criteria based on covering an abstraction of the intended behavior'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 criterion is a good predictor for conformance failure detection and for classical structural coverage criteria such as code and branch coverage.**

***Keywords*-conformance testing; object protocols; coverage criteria**

I. Introduction

Despite progress made, the automatic generation of efficient high quality test suites is still a major challenge for many kinds of software [34], [15]. 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 [3]. Components with non-trivial protocols and options are particularly challenging for testing approaches [13], [34].

A particularly important type of testing for stateful components is done to gain confidence in that they conform to their intended *protocols* (e.g., [12]). For instance, 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 protocol (as the dual problem of typestate verification [9]). Thus, protocol conformance underlies settings like Model-based development [28] and model based testing [31]. In this context, testing focus is on verifying that the code under test accepts or rejects sequences of method calls according to the intended protocol.

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 [15]. Coverage criteria are then defined either in terms of structural elements of the specification or the executable code generated from it. 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 (e.g., [14], [26], [27], [22], [33], etc.). Moroever, 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 semantic failure domain is explored is not studied. This in turn hinders the generalisation of the scarce empirical results of this body of work [15] to the general problem of black box testing of 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. Consequently traditional black box criteria for conformance testing of protocols are not applicable as finite state spaces are assumed [24], [22], [17]. Various strategies for finitising protocol state spaces have been studied [6], [19], [32]. However, there are no empirical results on their effectiveness.

Our general hypothesis is that effective notions of behavior coverage are actually feasible by defining them in terms of finite abstractions defined over the semantic domain that describes the intended protocol behaviour.

In this paper we propose coverage criteria over finite abstractions of infinite state behaviour protocols and present experiments in which results show that those criteria are good predictors for conformance failure detection and for structural coverage criteria (statement and branch coverage) when applied over code under test.

The practical implications of these results may be that in the context of development approaches which advocate test development before coding, generating tests according to an abstraction of the protocol semantics of an artefact with non-trivial requirements on method call ordering would provide a good criteria for detecting conformance failures and allow a first (and early) shot at producing high code coverage test suites (which could then be extended if necessary when code is available).

The coverage criteria are defined over enabledness preserving abstractions (EPAs) [8]. 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. We evaluate failure detection ability on five industrially relevant classes with rich protocols by analysing the mutant detection capability of randomly generated test suites. Results show that the criteria are good predictors of mutant detection in general. As achieving high coverage requires large test suites, we also study the relation between test suite size, EPA coverage and mutant detection. Results show that for fixed-size, test suites with the highest behavioral adequacy are statistically better.

To compare the proposed coverage criteria against a structural black box criteria, and to avoid benefitting from bias in the selection of the specification language and model to be structurally covered, we select the (unmutated) code itself as the specification. We believe that this choice does not favour (on the contrary) the hypotheses we propose in this paper as the code can be considered as the most detailed specification and most likely consitutes an upper bound on what structural criteria on specifications can achieve as test suite quality predictors. Results show that the EPA coverage criteria, which is behavioral and hence independent of specification language bias, performs comparably in terms of predictability of test suite failure detection.

We believe this paper is a step to understanding how behavioral coverage of protocol behaviour correlates with protocol conformance failure detection. This approach could help to improve random testing, test driven development, test case selection and, in general, techniques for tests generation from formal specifications. The results seem to indicate that such approaches would benefit from introducing heuristics that aim to maximise EPA coverage.

The rest of this paper is organised as follows. We begin with a lay out the problem we aim to address, formalising a conformance relation (Section II-A), the coverage criteria (Section II-B) and research questions (Section II-C). In Sections III and IV we present the experimental design and results, related work (Section VI) and conclusions and future work (Section VII).

II. Problem statement

We are interested in studying behavioral coverage criteria for testing protocol conformance. In this section we formalise the problem by defining what is meant by the intended protocol to be provided by a code artefact, the actual protocol implemented by a code artefact, and a conformance relation that is expected to hold between the intended and actual protocols. We also define the test adequacy criteria and then formulate four research questions that we address in the following sections.

A. Conformance relation

Full formal treatment of programming language semantics is beyond the scope of this paper. We provide an intuitive definition, sufficient for defining rigorously protocol conformance.

The semantics of a class or API implementation can be defined as a 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 will be present between states s and s' 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'.

Similarly, a specification language designed to describe the intended protocol behaviour of a class or API to be developed can be given semantics in a similar fashion. The *intended protocol LTS* defines which are the (potentially infinite) set of valid (potentially infinite) 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, in this paper we adopt (intended and actual) 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. Note that we require protocol LTS to be deterministic.

Definition 1 (Protocol LTS): Let $m_1, \ldots, m_n$ be method names, and $\mathcal{D}_i$ the domain of m_i. A LTS protocol for $m_1, \ldots, m_n$ is 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.
- $\Sigma = \bigcup_{i \leq n}(\{m_i\} \times \mathcal{D}_i)$ is the set of possible method invocations.
- $\Delta : (\mathbb{S} \times \Sigma \times \mathbb{S})$ is the transition relation that maps pairs of a state and method invocation to the corresponding resulting state. The relation must be partial function on the first two elements of the tuple.

The expresion $s \rightarrow^{m_i(p)} s'$ 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 protocol LTS is defined as an inclusion with respect to the sequences of method invocations they accept.

Definition 2 (Protocol LTS Conformance): Given two Protocol LTS, I and A, over the same set of methods with initial states $\mathbb{S}_{\mathbb{I}0}$ and $\mathbb{S}_{\mathbb{A}0}$, we say that I is *in conformance to* A if for all sequence w of method invocations -with concrete parameters-, $\mathbb{S}_{\mathbb{A}0} \rightarrow^w_A$ then $\mathbb{S}_{\mathbb{I}0} \rightarrow^w_I$.

A *failure* is then a sequence of method invocations with concrete parameters which is part of the intended protocol but does not terminate or raises an exception when executed on the implementation.

Definition 3 (Failure): Given two Protocol LTS, I and A, over the same set of methods with initial states $\mathbb{S}_{\mathbb{I}0}$ and $\mathbb{S}_{\mathbb{A}0}$ we say that a sequence w of method invocations -with concrete parameters- is a failure if $\mathbb{S}_{\mathbb{A}0} \rightarrow^w_A$ and $\mathbb{S}_{\mathbb{I}0} \not\rightarrow^w_I$. In practice, a client following the intended protocol may fail if the implementation is non-conformant.

B. Test adequacy criteria

We aim to define adequacy criteria over protocol behaviour independently of the specification language used to express the intended protocol and the programming language used to implement it. We define adequacy criteria over a conservative abstraction of their infinite behavior.

The abstraction we propose is based on [8]. Basically, a EPA is a LTS where labels are method names. 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 for every method and concrete parameter enabled in one state, that method for some parameter is enabled in the other. 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).

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

Definition 5 (Enabledness-preserving Abstraction): Given a protocol LTS $L = \left\langle \Sigma = \bigcup_{i \le n}(\{m_i\} \times \mathcal{D}_i),\ \mathbb{S},\ \mathbb{S}_0,\ \Delta \right\rangle$ for a protocol over method names $m_1, \ldots, m_n$ and $\mathcal{D}_i$ as the domain of m_i, we say that the LTS $M = \langle \bigcup_{i \le n}(\{m_i\}), S, S_0, \delta \rangle$ is an *enabledness-preserving abstraction* (EPA) of L if there exists a total function $\alpha : \mathbb{S} \rightarrow S$ s.t. $\alpha(\mathbb{S}_0) = S_0$ and for every $s \in \mathbb{S}$, method name m_i and parameter $p \in D_i$ s.t. $s \rightarrow^{m_i(p)} s'$ holds, then $(\alpha(s), m_i, \alpha(s')) \in \delta$. Furthermore, given a pair of states s_1, s_2 on $\mathbb{S}$, it holds that $s_1 \equiv s_2 \iff \alpha(s_1) = \alpha(s_2)$. Figure 1 shows the EPA of JDK 1.4 `Socket` class.

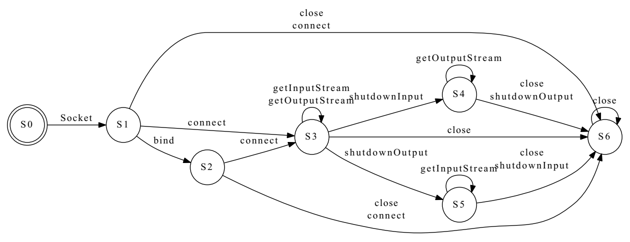

Figure 1. (Enabledness Preserving) Abstraction of the semantics of the JDK 1.4 `Socket` implementation.

We now define two adequacy criteria over EPAs of protocol LTS. 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. That path mimics the execution over the instance by transitioning over the LTS until either a state is reached where the next method invocation is not enabled in the protocol, or the execution was completely simulated by the LTS. In turn, (and because EPAs can simulate all paths of the LTS they abstract) a path on the protocol LTS can be univocally simulated by a path in the EPA by applying the abstraction function α. We call the later an *α-abstracted execution* of a unit test.

Definition 6 (EPA Transition K-adequacy Criterion): Let L be a protocol LTS, A its EPA, and TS a test suite. We say that TS is EPA-T-k adequate for L if the α-abstracted executions of all unit tests in TS cover at least $k\%$ of the transitions in A.

As an example, let us consider the EPA shown in figure 1. A EPA-T-100 adequate test suite would necessarily contain a unit test in which `shutdownOutput` is executed before `shutdownInput` and also a unit test in which `shutdownOutput` is executed after `shutdownInput`.

Probing the identity of state is typically part of most algorithms for finite state machine testing [18]. Therefore, transition pairs can also be regarded as an interesting criterion since it measures to which extent target states of transitions have been probed by executing the expected enabled transitions.

Definition 7 (EPA Transition Pairs K-adequacy Criterion): Let L be a protocol LTS, A its EPA, and TS a test suite. We say that TS is EPA-P-k adequate for L if the α-abstracted executions of all unit tests in TS cover at least $k\%$ of the transition pairs in A.

Note that neither the conformance relation nor the adequacy criteria make assumptions on the way the *intended protocol LTS* of a code artefact is described: we simply assume that the semantics of such language can be defined in terms of a protocol LTS as defined above. 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 MBT setting, it could be described in a technical documentation, 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, etc.

C. Research Questions

We now pose the research questions that are the focus of the experimentation reported in the next sections.

The first question analyses the correlation between the coverage of EPA transitions and the ability of a test suite to detect faults that manifest themselves as violations of the intended protocol, i.e. protocol conformance failures.

RQ1. To what extent does the value of K in the EPA transition (transition pairs) K-adequacy of the intended protocol LTS predict the ability of test suites to detect protocol conformance failures?

In order to understand if the correlation obtained is reasonably high, we also look at how well branch and statement coverage performed over the code of the subjects (i.e., as white box criteria) would predict test suite quality. The importance of this analysis is twofold. First it provides a baseline reference for correlations yielded by the proposed criteria. Second it enables a comparision against what would be measuring structural adequacy using an ideal specification in terms of closely mimicking the structure of code under test. The choice of code as the specification language against which to compare EPA coverage is discussed in Section V.

Since requiring coverage leads naturally to requiring longer test suites, it is standard to analyse if stronger correlations are just a consequence of length [23].

RQ2. Given a fixed test suite length, do test suites with higher EPA transition (transition pairs) K-adequacy on the intended protocol LTS perform better in terms of failure detection than those with lower EPA transition (transition pairs) K-adequacy?

More specifically, we aim to study if picking a test suite with higher adequacy is more likely to detect more failures than picking a test suite of the same size but with lower adequacy.

The third and fourth questions explore how well EPA transition (transition pairs) K-adequacy can predict statement and branch coverage of the code under test.

RQ3. To what extent does the value of K in the EPA transition (transition pairs) K-adequacy of the intended protocol LTS of an implementation predict the achieved level of code coverage on a conformant implementation?

That is, is it true than the more abstract behavior is covered then the more code is structurally covered? As before, we also study the relation between EPA transition (transition pairs) K-adequacy and code coverage for fixed test suite lengths:

RQ4. Given a fixed test suite length, do test suites with higher EPA transition (transition pairs) K-adequacy on the intended protocol LTS perform better in terms of code coverage than those with lower EPA transition (transition pairs) K-adequacy?

III. Experiment Design

A. Experiment Overview

To answer the four research questions proposed in section II-C two values associated to test suites must be studied: *number of failures detected* and *achieved code coverage*.

Failure detection involves (a) fixing both an intended protocol LTS and a conformant implementation and (b) obtaining implementations that fail to conform to the intended protocol in diverse ways. Our strategy involves selecting an implementation as a *reference implementation* to be used both as the specification of intended behaviour and as the basis for generating faulty implementations.

For obtaining *faulty implementations* we applied mutation operators to the reference implementation. Identification of failures is done by executing unit tests on both the referece implementation and on a mutated implementation. When mutation is unable to execute a valid sequence of calls in the subject implementation then the mutant is killed. 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 40% and 70% of mutations (depending on the case study) produce conformance failures. To have a representative set of flawed implementations, we decided not to filter a priori the applied mutation operators. Simply, mutants that were not killed by any unit test (i.e., no test sequence led to an unexpected exception of that mutant) were considered mutations that have the same actual protocol as the reference implementation of the class.

The strategy for test suite generation is random generation of unit tests and random grouping them into test suites. The selected code coverage criteria are statement and branch coverage, both measured on the reference implementation as unit tests are run.

B. Subjects

We restricted the universe of potential subjects to one programming language to allow for a uniform experimental platform regarding mutation, test genaration and code coverage tools, and infrastructure for detecting failures. In particular, we fixed the language to Java to take advantage of existing tools and the availability of Java classes that satisfied our 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 protocols); $ii)$ code that is of industrial relevance; and $iii)$ code for which its EPA can be obtained (see Section III-C).

We performed studies on 5 subject Java 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 `Signature` class is used to provide applications the functionality of a digital signature algorithm; `ListItr` provides functionality to go through the elements stored in a list; `Socket` provides the client-side functionality to establish a TCP connection between two hosts; `SMTPProcessor` is a core class of the Java Email Server responsible for processing all incoming SMTP requests; and `JDBCResultSet` represents a set of data which is generated by executing a query to a database. The class allows iterating over the result and making updates on the underlying database.

C. Construction of EPAs of Intended Protocol LTSs

A key resource for all subjects was the tool CONTRACTOR [7]. It constructs EPA either from contract-based specifications [7] or directly from source code [8].

Since we require intended protocol to be the actual protocol of 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 in [8].

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 [4] for the `ResultSet` JDBC interface. We validated the contract specification by comparing the preconditions of each method against the conditions that guard exception throwing statements in the reference implementation. Indeed, our analysis concludes that resulting EPA is that of actual protocol LTS for the `JDBCResultSet` reference implementation.

D. Experiment Implementation details

In this study, for each subject we generated 10000 unit tests by using RANDOOP [25], an automatic unit test generator for Java classes. These unit tests are grouped randomly into test suites. In order to get statistically significant results, test suites with different levels of coverage of EPA models are required. To address this requirement we randomly vary the number of unit tests that make up test suites. For each subject appropriate ranges of unit tests per test suite were defined to achieve varied EPA coverage.

For obtaining mutated versions of each subject class we used μ-JAVA [20], a mutation system for Java programs. In order to obtain as many implementations as possible, we let μ-JAVA apply every mutation operator whenever possible[1]. Some of these mutants may not be semantically equivalent but there is no evidence that their actual protocol is different to the one of the conformant implementation.

[1]Due to time limitations, in the case of `SMTPProcessor` we randomly selected only the 20% of generated mutants, since the execution of all tests on all mutants would have taken more than one year.

We measured statement and branch coverage using COBERTURA, a Java tool that calculates the percentage of code accessed by tests.

Table I summarises relevant information for each subject. Column 2 exhibit lines of code; column 3 the number of mutants detected; columns 4 and 5 expose information regarding EPAs (number of states and transitions); and column 6 shows the number of EPA transitions that were reached by at least one test suite.

Class	LOC	Mutants	States	Tx	Tx Covered
`Signature`	121	96	4	29	27
`ListItr`	59	145	8	68	63
`Socket`	144	59	7	20	18
`SmtpProcessor`	404	89	12	85	70
`JDBCResultSet`	785	259	9	247	199

Table I
SUBJECT CLASSES SUMMARY

IV. RESULTS

A. Research Question 1

In order to determine to what extent the value of K in EPA transition (transition pairs) K-adequacy of the intended protocol LTS predicts the ability of test suites to detect protocol conformance failures we use Spearman's rank correlation coefficient ρ. This coefficient does not make any assumption about the distribution of data. This is important as we were unable to establish that data fitted known distributions using goodness of fit tests such as Kolmogorov-Smirnov.

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

Class	Stmt	Brch	Tx	Tx Pairs
`Signature`	0.65	0.71	0.73	**0.78**
`ListItr`	0.48	0.61	0.84	**0.85**
`Socket`	0.81	**0.86**	0.72	0.73
`SmtpProcessor`	0.66	**0.78**	0.69	0.70
`JDBCResultSet`	0.89	**0.92**	0.68	0.66

Table II
CORRELATION BETWEEN COVERAGE AND FAILURE DETECTION. BOLD INDICATES THE HIGHEST VALUE FOR A ROW.

Table II shows the ρ values obtained for each criterion. Transition pairs coverage has high correlation ($\rho > 0.7$) for four subjects and moderate for `JDBCResultSet` (but very close to high). Transitions coverage has high correlation in three cases and moderate (but again very close to high) in the remaining two. As can be seen, even compared with structural criteria over code, the defined black box behavioral criterion does not perform poorly. Two case studies illustrate the best and worst cases for behavior coverage and structural coverage criteria. On the one hand, poor performance of behavior criteria on `JDBCResultSet` seems to be justified because API implementation is unbalanced in terms of amount of mutable code (one method out of forty-two collects the 35% of all mutations of the class). On the other hand, for `ListItr` EPA coverage criteria have high correlation while both statement and branch coverage correlate only moderately. This might be explained as follows: this subject presents a simple code structure (no loops, few branches) that can be easily covered by test suites, while 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. Finally, transition pairs seems to have similar predict power than transitions coverage.

B. Research Question 2

The impact of length (as the number of calls to the software under test) on the effectiveness of test suites has been addressed by several works [23], [1]. Although it is known that not only the size determines the quality of a test suite, in general it is expected that the greater the length, the greater the likelihood of revealing failures [23].

The tests that achieve higher EPA coverage are generally also longer. Thus, we analyse test suites on a "by length" basis: given a length we aim to study if the test suites with higher EPA transition (transition pairs) K-adequacy of the intended protocol LTS are likely to detect more failures than those with lower adequacy.

Samples for each length are not large enough for obtaining statistically significant results. Therefore, we divide them into bins grouping those of similar length in the same bin. For each subject, we ensure that the difference of length between test suites of the same bin do not exceed 10% of the difference of length between the longest and the shortest overall test suites.

We define the set of test suites in a bin with higher adequacy as those that achieve at least 80% of the coverage that is achieved with the test suite with highest coverage of that bin. This is because the degree of coverage varies from bin to bin due to the change in test suite lengths. In this way we can obtain the "best" test suites for a particular bin.

As explained in Section IV-A, detected failure data does not necessarily fit standard distributions. Therefore, as before, we choose a nonparametric test for our analysis. In order to compare higher coverage test suites of a bin against the rest of that bin, we use the Mann-Whitney U test, 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 criteria and those that do not form disjoint sets. It also requires the responses are ordinal or continuous measurements, which is also true because the variables considered here are EPA transition (transition pairs) coverage and detected failures. 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 p-value resulting from the hypothesis test is less than 0.05.

To also assess the magnitude of the improvement we use the Vargha and Delaney's A_{12} effect size (ES). This non-parametric measure has recently been advocated in [2] for randomized algorithms. In our case, in a given bin, A_{12} estimates the probability that choosing a tests suite of high transition coverage detects more mutants than a test suite chosen randomly from the population of low coverage. We also report the confidence interval (CI) for the effect size stated at the 95% confidence level.

Table III shows the test results for all subjects. Each row of the table corresponds to one bin. The second column specifies the test suite length interval of each bin. The third indicates the minimum and maximum number of EPA transitions covered by them, and the fourth shows the minimum 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 column show the number of tests that achive high and low EPA transitions coverage respectively. The seventh column exhibits the p-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 RQ4. The same analysis was performed to evaluate transition pairs coverage and results do not significatly differ from those obtained for transition coverage. [2]

Results suggest that EPA coverage makes a difference in terms of failure detection for tests suites of the same length.

C. Research Question 3

As with detected failures, code coverage achieved by test suites does not fit standard distributions either. Therefore, to find out how well the coverage of EPA predicts code coverage we again calculate Spearman's rank correlation coefficient ρ. The results are shown in table IV.

Results lead to believe that EPA coverage criteria are reasonably good predictors 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

[2]Due to space limitations they are not reported in this paper. They are available at `http://lafhis.dc.uba.ar/epa_testing`. The same applies to RQ4.

Subject	EPA Coverage					Detected Faults			Statement Coverage Percentage			Branch Coverage Percentage		
	Length Interval	Covered Tx	Threshold	H-population	L-population	p-value	ES	CI	p-value	ES	CI	p-value	ES	CI
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]	$0.16^{\dagger}$	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]
ListItr	[50,99]	16 - 37	30	90	222	~ 0	0.73	[0.67,0.79]	$0.23^{\dagger}$	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]	$0.66^{\dagger}$	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]	$0.90^{\dagger}$	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]	$1.00^{\dagger}$	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]	$0.93^{\dagger}$	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]	$1.00^{\dagger}$	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]	$1.00^{\dagger}$	0.50	[0.50,0.50]	$0.13^{\dagger}$	0.58	[0.52,0.63]
	[400,449]	35 - 52	42	161	26	$0.08^{\dagger}$	0.61	[0.51,0.72]	$1.00^{\dagger}$	0.48	[0.48,0.48]	$0.08^{\dagger}$	0.62	[0.54,0.70]
	[450,499]	36 - 53	43	145	28	~ 0	0.73	[0.63,0.82]	$1.00^{\dagger}$	0.48	[0.48,0.48]	0.03	0.62	[0.54,0.70]
	[500,549]	36 - 52	42	113	14	$0.08^{\dagger}$	0.63	[0.49,0.77]	$1.00^{\dagger}$	0.50	[0.50,0.50]	0.02	0.72	[0.60,0.85]
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	$1.00^{\dagger}$	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	$0.37^{\dagger}$	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	$1.00^{\dagger}$	0.40	[0.40,0.40]	$0.63^{\dagger}$	0.33	[0.29,0.38]	$0.30^{\dagger}$	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]	$0.20^{\dagger}$	0.55	[0.49,0.61]	0.03	0.58	[0.52,0.65]
	[1200,1349]	31 - 49	40	153	105	$0.13^{\dagger}$	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	$0.45^{\dagger}$	0.54	[0.46,0.61]	$0.50^{\dagger}$	0.53	[0.45,0.61]	$0.07^{\dagger}$	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	$0.73^{\dagger}$	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	$0.09^{\dagger}$	0.73	[0.69,0.76]	$0.09^{\dagger}$	0.76	[0.72,0.79]	$0.10^{\dagger}$	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	$0.86^{\dagger}$	0.47	[0.21,0.74]	$0.49^{\dagger}$	0.53	[0.40,0.67]	$0.94^{\dagger}$	0.45	[0.27,0.62]
	[2000,2199]	120 - 153	123	248	17	$0.06^{\dagger}$	0.70	[0.57,0.82]	0.01	0.79	[0.68,0.89]	$0.08^{\dagger}$	0.65	[0.57,0.73]

Table III
MAN-WHITNEY TEST RESULTS FOR FAILURE DETECTION AND CODE COVERAGE. VALUES NOT STATISTICALLY SIGNIFICANT ARE MARKED WITH †.

Class	Tx/St	Tx/Br	Pairs/St	Pairs/Br
Signature	0.63	0.70	0.61	0.66
ListItr	0.66	0.70	0.47	0.64
Socket	0.77	0.77	0.77	0.78
SmtpProcessor	0.70	0.73	0.69	0.69
JDBCResultSet	0.67	0.66	0.63	0.63

Table IV
CORRELATION BETWEEN EPA MODEL AND CODE COVERAGE.

in `JDBCResultSet`, in all cases the correlation with branch coverage is greater or equal than that of statement coverage. In the type 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 behavior coverage criteria work typically better as predictors of mutant detection than as code coverage predictor -which seems consistent to the rationale underlying criteria.

D. Research Question 4

We address this research question in a similar way as RQ2, using the same set of bins. Again, we do not assume code coverage achieved by test suites fits a common distributions. Thus, in order to determine if adequate tests achieve higher code coverage than non-adequate ones we perform Mann-Whitney tests. As for RQ2, the assumptions made by the test are also met in this case.

Table III 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/branch coverage. In our 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.

Again, results show that statistical significatively evidence exists for saying that test suites achieving higher EPA coverage criteria are more likely to achieve higher code coverage than the general population of a given size. The exception might be `ListItr`, but 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.

V. Threats to Validity

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

Threats to external validity concern our ability to generalise the results. The expectation is that our results can be generalised 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.

Threats to internal validity appear as a consequence of how we conducted the experiments. 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 failures, as for this the reference implementation itself and not its abstraction is used). We believe that 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.

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

We believe that threats regarding unintended effects of general experimental infrastructure needed for i) mutant generation, ii) coverage measurement and iii) failure detection are minor since we have used standard tools such as COBERTURA, μ-JAVA and RANDOOP whenever possible and simple code instrumentation techniques using AspectJ.

Finally, we mitigate internal validity threats by making data required for third party evaluation of our experiments at `http://lafhis.dc.uba.ar/epa_testing`.

Threats to construct validity mainly appear from our choice to compare our criterion with code coverage criteria. We recognise that 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 failure detection for EPA coverage are reasonable, some well accepted baseline is needed. We chose structural code coverage criteria. However, we also complement comparisons with code coverage in our experiments with the study of EPA coverage against failure detection.

On the other hand, the proposed criteria are black box while we compare with what in principle could be considered white box criteria in RQ1 (i.e. structural code coverage). It could be argued that a more suitable baseline would be some other black box criteria such as structural coverage over a specification. Unfortunately, there is no de facto standard black box baseline for rich modeling 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 (e.g., [14], [26], [27], [22], [33], etc.). We believe that taking the most detailed specification (the code itself) constitutes an upperbound 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.

VI. Related Work

A. Models for Testing

Much research effort on the testing has focused mainly on test case generation by exploiting to various degrees the code-under-test: from purely systematic white-box approaches (e.g., [30]) to search-based approaches [29]) 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 [19] the state space of a class is quotiented based on its parameterless boolean observers (similar approach for a different purpose is in [35]). In [32], abstract states are computed using shape abstraction, i.e., 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 internal state of the SUT. In [6], 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.

Our work 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 failure rate detection or structural coverage of the class under analysis.

A notable recent related work is that of [10], where adequacy criteria are based on the behaviour of the software under test. In that work, a test suite is considered adequate if a reasonable model can be inferred from it. Nevertheless, the authors do not address the problem of conformance testing: they model programs as functions (i.e, programs that receive inputs and produce outputs) and it is the relation between these values what constitutes the behavioural model.

B. Conformance Testing

There is plenty of work focused on defining coverage criteria for formal specifications ranging over a plethora of languages and computation models (see [15] as a survey). Here we focus on approaches that are straightforwardly applicable to conformance testing.

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 or of the executable code generated from the specification or simulation model. Representative examples are [24], [17], [22] 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 yet are scarce for state-based specification languages [15] which are particularly appropriate for conformance testing. Some notable exceptions we found are [27], [26], [22]. Interestingly enough, in these, experiments were conducted on criteria based on covering code generated from models [26] or simulation code [27]. In [22] hand-made test suites based on UML state machines are compared against test suites based on structural testing. Like authors of [26], we speculate 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. We believe the work presented herein is a step forward in this direction.

On the other hand, in behaviour approaches, coverage is defined in terms 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 [18]. In foundational work, the conformance problem is stated in terms of Mealy machines. However, in contrast to our approach, coverage and failure models assume finiteness of both the specification and the actual implementation. Early work that addresses drops the finiteness assumption is that of LTS based testing [16] where IOCO is a well established notion of conformance, however no notion of behaviour coverage has been defined in this setting of infinite state space.

Infinite behaviour models can be dealt with introducing abstraction. Several finitisation techniques exists: unfolding [5], domain bounding and slicing and state prunning [12]. However, no statistical studies on coverage for these finitisations is available. Other relevant work in this line is that of automatic under approximation of infinite behaviour from concrete [21] or symbolic [11] executions that can be later used for regression testing. Here, again, no statistical study is available.

VII. Conclusions and Future Work

This paper is a first step towards defining and understanding how semantic coverage of infinite state behaviour specifications relates to effective testing techniques for protocol conformance. We address this by studying coverage achieved on an abstraction of such behaviour, more specifically on enabling preserving abstractions (much in the vein of typestates [4]). We believe a good understanding of the relation between failure detection, white box coverage criteria, and coverage of abstractions of the semantic space of specifications could help to improve random testing, test case selection techniques and, in general, heuristics to generate tests from formal specifications.

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

In addition, results lead to believe that EPA coverage is a good predictor of statement and code 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, etc.) 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. 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 [7] or code [8]. Furthermore, EPA abstractions could be provided directly by testers as advocated by typestate approaches [3].

Future work should aim at looking at other protocol abstractions and comparing them with EPAs in terms of their effectiveness for testing protocol conformance. We plan to further our experimentation by addressing questions about the relationship with dataflow coverage criteria, the effect of test suite minimisation on failure detection, and the sort of bugs semantic behavior coverage is good for detecting. 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, 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.

Acknowledgments. The work reported herein was partially supported by CONICET PIP 955, PICT PAE 2272, PICT PAE 2278, UBACYT W0813, PICT 1774, ERC StG PBM-FIMBSE and MEALS 295261.

References

[1] J. Andrews, A. Groce, M. Weston, and R.-G. Xu. Random test run length and effectiveness. In *ASE '08*, pages 19–28, 2008.

[2] A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE '11*, pages 1–10, 2011.

[3] N. Beckman, D. Kim, and J. Aldrich. An empirical study of object protocols in the wild. In *ECOOP '11*, pages 2–26, 2011.

[4] K. Bierhoff, N. E. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *ECOOP '09*, pages 195–219, 2009.

[5] G. Bochmann, A. Petrenko, O. Bellal, and S. Maguiraga. Automating the process of test derivation from SDL specifications. In *SDL Forum '97*, 1997.

[6] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *ISSTA '10*, pages 85–96, 2010.

[7] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Validation of contracts using enabledness preserving finite state abstractions. In *ICSE '09*, pages 452–462, 2009.

[8] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Program abstractions for behaviour validation. In *ICSE '11*, pages 381–390, 2011.

[9] R. DeLine and M. Fähndrich. Typestates for objects. In *ECOOP '09*, pages 465–490, 2004.

[10] G. Fraser and N. Walkinshaw. Behaviourally adequate software testing. In *ICST '12*, pages 300–309, 2012.

[11] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *ISSTA '02*, pages 112–122, 2002.

[12] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *STVR*, 21(1):55–71, 2011.

[13] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *ISSTA '12*, pages 78–88, 2012.

[14] M. P. Heimdahl, D. George, and R. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In *HASE '04*, pages 178–186, 2004.

[15] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41:9:1–9:76, 2009.

[16] Jan and Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996.

[17] B. Korel, G. Koutsogiannakis, and L. H. Tahat. Model-based test prioritization heuristic methods and their evaluation. In *A-MOST '07*, pages 34–43, 2007.

[18] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[19] L. Liu, B. Meyer, and B. Schoeller. Using contracts and boolean queries to improve the quality of automatic test generation. In *TAP '07*, pages 114–130. 2007.

[20] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system. *STVR*, 15(2):97–133, 2005.

[21] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. Autoblacktest: a tool for automatic black-box testing. In *ICSE '11*, pages 1013–1015, 2011.

[22] S. Mouchawrab, L. C. Briand, Y. Labiche, and M. Di Penta. Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments. *TSE*, 37(2):161–187, 2011.

[23] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *ISSTA '09*, pages 57–68, 2009.

[24] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *STVR*, 13(1):25–53, 2003.

[25] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE '07*, pages 75–84, 2007.

[26] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *ICSE '05*, pages 392–401, 2005.

[27] M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Evaluating test suites and adequacy criteria using simulation-based models of distributed systems. *TSE*, 34:452–470, 2008.

[28] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[29] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov. Testing container classes: Random or systematic? In *FASE '11*, pages 262–277, 2011.

[30] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *OOPSLA '11*, 2011.

[31] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2007.

[32] W. Visser, C. Păsăreanu, and R. Pelánek. Test input generation for java containers using state matching. In *ISSTA '06*, pages 37–48, 2006.

[33] S. Weissleder. Simulated satisfaction of coverage criteria on uml state machines. In *ICST '10*, pages 117–126, 2010.

[34] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *ICSE '11*, pages 611–620, 2011.

[35] T. Xie and D. Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Formal Methods and Software Engineering*, volume 3308 of *LNCS*, pages 290–305. 2004.