White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of Sheffield.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/154925/

Version: Published Version

Article:

Simons, A. and Lefticaru, R. (2020) A verified and optimized Stream X-Machine testing method, with application to cloud service certification. Software Testing, Verification and Reliability, 30 (3). e1729. ISSN: 0960-0833

https://doi.org/10.1002/stvr.1729

Reuse
This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here:
https://creativecommons.org/licenses/

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

UNIVERSITY OF LEEDS
University of Sheffield
UNIVERSITY of York

SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2020;e1729.
Published online in Wiley Online Library (wileyonlinelibrary.com). https://doi.org/10.1002/stvr.1729

RESEARCH PAPER

A verified and optimized Stream X-Machine testing method, with application to cloud service certification

Anthony J. H. Simons*,† and Raluca Lefticaru

Department of Computer Science, University of Sheffield, Sheffield S1 4DP, UK

SUMMARY

The Stream X-Machine (SXM) testing method provides strong and repeatable guarantees of functional correctness, up to a specification. These qualities make the method attractive for software certification, especially in the domain of brokered cloud services, where arbitrage seeks to substitute functionally equivalent services from alternative providers. However, practical obstacles include the difficulty in providing a correct specification, the translation of abstract paths into feasible concrete tests and the large size of generated test suites. We describe a novel SXM verification and testing method, which automatically checks specifications for completeness and determinism, prior to generating complete test suites with full grounding information. Three optimization steps achieve up to a 10-fold reduction in the size of the test suite, removing infeasible and redundant tests. The method is backed by a set of tools to validate and verify the SXM specification, generate technology-agnostic test suites and ground these in SOAP, REST or rich-client service implementations. The method was initially validated using seven specifications, three cloud platforms and five grounding strategies. ©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

Received 14 May 2018; Revised 10 September 2019; Accepted 15 December 2019

KEY WORDS: cloud computing; cloud service broker; functional testing; service certification; specification; state-based testing; test grounding; verification; X-machines

1. INTRODUCTION

Software certification is the process of guaranteeing that a piece of software performs exactly according to its specification. Software certification is increasingly relevant in cloud computing, especially in multi-partner cloud service ecosystems [1], in which services are offered by many providers to many consumers, brokered by intermediaries who resell customized and repackaged service bundles. In a market where consumers can select from competing service offerings on the basis of functionality, performance and cost, *cloud brokers* [2] play a significant business role, offering *intermediation* (added-value services, unified access and identity management), *aggregation* (construction of composite services out of simple services, with secure data movement) and *arbitrage* (dynamic selection and substitution of services, to optimize cost or performance‡,§).

*Correspondence to: Anthony J. H. Simons, Department of Computer Science, Regent Court, 211 Portobello, Sheffield, S1 4DP, UK.

†E-mail: a.j.simons@sheffield.ac.uk

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

‡Forrester Research Inc., 2012: Cloud brokers will reshape the cloud – getting ready for the future cloud business models. https://toolsynergie.files.wordpress.com/2016/09/cloud-business-modellen.pdf

§D. C. Plummer, B. J. Lheureux, M. Cantara, T. Bova, 2011: Cloud services brokerage is dominated by three primary roles. https://www.gartner.com/doc/1857618/cloud-services-brokerage-dominated-primary

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

Enhancing the role of the *cloud broker* as guarantor of quality assurance [3] was the premise behind the EU FP7 BrokerCloud project [4], which investigated methods and mechanisms for continuous quality assurance and optimization of brokered software services in the cloud. The project demonstrated a brokerage platform which could validate and test software services prior to uploading (*certification at onboarding*) [5], manage the service lifecycle from creation to decommissioning (*lifecycle governance*) [6], regulate the performance and availability of services (*monitoring* and *adaptation*) [7] and recommend alternative service bundles, according to customer preferences (*preference-based arbitrage*) [8]. The current article describes the novel verification and testing approach that was developed to assure functional quality and substitutability, as part of a service certification strategy.

1.1. Functional service certification

Certification of services includes testing their functional behaviour (to assure their correctness) and non-functional aspects (to assure their performance). While the monitoring and enforcement of service-level agreements (SLAs) has been investigated [9–11] as a kind of performance testing, the functional certification of service behaviour has received much less attention [12].

Our vision was to provide a web services aligned XML specification format that could be used by tools hosted at distributed locations in the cloud, at different stages in the service lifecycle. A specification should be amenable to checking for consistency and completeness; it should later be used as the basis for model-based test generation, yielding test suites that check an implementation for full conformance to the specification. The ability to test services for conformance not only guarantees that the service is implemented correctly but also ensures that any substituted service behaves in exactly the same expected way. By offering a common XML specification format, we aim to apply a gentle standardizing pressure, to promote the creation of compatible services that may be substituted by a broker during arbitrage.

Testing software services in the cloud is challenging for several reasons. Firstly, service-oriented architectures (SOA) are implemented using a diverse range of technologies, which include standard web service protocols (WSDL[¶] and SOAP[‖]), popular internet conventions (REST[**] and JSON[††]) or bespoke client-server streaming (rich-client desktops and AJAX[‡‡]). Finding a suitable common specification model and test generation approach that might suit all of these is extremely difficult. Secondly, the stateless nature of HTTP protocols makes tracking the states and transitions of web services hard, unless this information is exposed by the services through design-for-test conventions. Thirdly, cloud-based web services are highly complex in their handling of concurrent requests, sessions and multiple tenancies; however, because tenancy and sessions are handled by a different authorization layer of the software, we believe this may be treated independently of service functional behaviour.

1.2. Stream X-Machines in theory and in practice

We adopted the Stream X-Machine as the formal basis for the certification method. An X-Machine has a Turing-complete ability to model any realistically complex software system as an extended finite state machine (EFSM) with processing functions acting on memory [13]. A Stream X-Machine (SXM) is also controllable through inputs and observable through outputs [14, 15], so facilitating the associated complete functional testing method [16, 17], which offers strong guarantees of conformance to the specification (see also Section 3).

[¶] Web Services Description Language (WSDL) Version 2.0: https://www.w3.org/TR/wsdl20/

[‖] SOAP version 1.2 part 1: Messaging framework (second edition), W3C recommendation: https://www.w3.org/TR/soap12-part1/

[**] REST discussion draft, with no formal status at W3C: https://www.w3.org/2001/sw/wiki/REST

[††] JSON data interchange format, standard ECMA-404: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[‡‡] XMLHttpRequest living standard, last updated 18 February 2019: https://xhr.spec.whatwg.org/

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

A specification is developed by modelling the control states of the software as a finite state machine (FSM), whose transitions are functions acting upon memory. The memory is an arbitrary tuple of variables, of arbitrary types. The simple state-transition graph is augmented by guarded transitions, where the guards are sensitive to inputs and memory states. The functions may also update values stored in memory. The classic SXM testing method requires a deterministic, complete and minimal specification [14, 15, 18]. Not only does the fundamental testing method [19] generate positive paths that should execute in the software but also negative paths that should be prevented by the software, thereby providing strong guarantees of correctness [16, 17].

However, there remain gaps between the theory and practice of using SXMs, which can constitute serious obstacles to their adoption. We enumerate a number of these as follows:

- The mathematical formalism is potentially opaque to software engineers, such that it may be hard for the engineer to provide a deterministic (complete and non-blocking) specification.
- The testing method elides over how abstract sequences are to be converted into executable test sequences, by assuming the existence of a hypothetical *test function* [15] that maps sequences onto test inputs.
- The test suite is generated by exploring the associated automaton, disregarding whether sequences are blocked by the guards, which leads to formal workarounds (*controllability, input uniformity* [20, 21]) to deal with *infeasible* paths.
- Though generated test suites are highly discriminatory, they are still too large and could be optimized by making reasonable design-for-test assumptions; in particular, the use of extended characterization sequences in the fundamental testing method [19] to identify reached states multiplies the size of the test suite by a factor equal to the size of the characterization set.

1.3. Theoretical and practical innovations

Our solutions to these problems are a mix of theoretical innovation and practical engineering. They depend on making a complete model of the Stream X-Machine specification available to a set of verification and testing tools that can reason about every aspect of the specification. The tools complement each other in the way they detect faults at the appropriate stage in design. Altogether, we claim the following innovations:

- a wholly available specification model that exposes not only the abstract state-transition behaviour of the automaton but also the concrete input, output, precondition and effect (IOPE) behaviour of operations acting upon memory, to the tools that reason about the specification;
- a novel verification algorithm that, for each operation, computes all symbolic partitions of inputs and memory, eliminating inconsistent partitions by constraint satisfaction, prior to determining whether the operation is consistent (deterministic) and complete (non-blocking);
- a novel test generation algorithm that also determines path-feasibility during test generation, addressing concerns about machine *controllability* or *input uniformity* and state reachability via the *r-state cover* [20, 21];
- a novel test optimization algorithm that eliminates infeasible paths and redundant sequences with proven trivial prefix cycles from the generated test suite, replaces extended characterization sequences by reliable state oracles and supports test-compression via merged multi-objective tests; and
- a novel test grounding method that uses test input constraints from the specification to generate concrete test inputs and invocations, generating code via a combination of design patterns, creating executable tests in a variety of implementation technologies.

1.4. Design-for-test conditions

To ensure that tested software services are aligned with the assumptions required by the testing method, service providers must implement a number of design-for-test criteria. These are not onerous and are frequently already part of internal logging mechanisms. A service must provide a reliable *clean reset* operation to put the service into its initial state and rebind its memory to the initial values; this is cheaper than re-initializing a cloud service from scratch and may already be provided as

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

DOI: 10.1002/stvr

part of an abort mechanism. It must provide a reliable *transaction log* that records which responses were triggered in reply to which requests; this is to satisfy the *output distinguishability* [15, 16] criterion directly, in cases where operations do not naturally produce outputs. Finally, it must provide a reliable *state oracle* that reports the abstract state in which the service finds itself; this replaces extended characterization sequences [19] to identify reached states.

The *clean reset* is needed, because tests generated by the W-method [19] always assume restarting in the initial state. We also considered the transition tour method [22] which avoids reset; however, there is no guarantee that such a tour exists in the specification and this method cannot assure correct state transfer. We considered the unique input–output (UIO) method [23] which identifies each state via one sequence, but this method cannot detect all faulty implementation states. The *state oracle* method to verify states was inspired by *reliable state oracles* [24] that avoid extended characterization sequences; we developed proofs to integrate this within existing SXM theory, and other optimizations depend on this.

1.5. BrokerCloud Verification and Testing Tool Suite

The novel algorithms described earlier are demonstrated in the *Verification and Testing Tool Suite* (VTTS), one of the outputs of the EU FP7 BrokerCloud project [4]. The tool suite is freely available to download under an Apache 2.0 license, and individual tools may be explored online, using sample service specifications [25]. Users may also upload their own specifications, developed according to the user guide. This demonstration via the Internet may also constitute an early example of Testing as a Service.

The tools all use an Internet-transmissible XML format for encoding Stream X-Machine specifications and generated test suites. The specification language combines *finite state machines* (FSMs) with a popular web services functional protocol known as *input, output, precondition and effect* (IOPE), which in our previous work was shown to be compatible with Stream X-Machines [26]. The XML specification language maps directly to a Java model, defined by a metamodel, which supports the model-based reasoning performed by the tools.

Checking of model specifications is performed by two tools: a *validation tool*, which checks the explicit and implicit state-transition behaviour against the designer's expectations, by exploring the state machine, and a *verification tool*, which checks the specification's operations for consistency and completeness, by a combination of symbolic model checking and constraint satisfaction. These tools annotate the specification with warnings if faults are discovered.

The *test generation tool* generates complete functional test suites, in a technology-agnostic XML format, but which contain full grounding information about invocation sequences, test inputs and corresponding outputs, triggered transitions and reached states. The test suites are optimized to remove infeasible and redundant tests and may be further compressed as multi-objective tests. For a typical service with 2–7 states and 10–50 transitions, these optimizations reduce the size of a generated test suite to as little as 10% of its original size, without loss in fault detection.

The *test grounding tool* translates these abstract test suites into three sample concrete web service execution formats, by model-based code generation. As a proof of concept, we provide groundings to Java web services, creating JUnit test drivers for JAX-WS (SOAP services), JAX-RS (REST services) and for plain Java. However, these are not the only possibilities. BrokerCloud industry partner SAP SE created a bespoke grounding for use with the Selenium test engine, executing a SAP OpenUI5 rich-client application on the HANA platform [27]. We later developed a further bespoke grounding for the SOAP UI test engine [28].

1.6. Overview of the article

The rest of this article is structured as follows. Section 2 justifies the XML specification format, based on its logical adequacy and its relevance to service-oriented standards. Section 3 presents the novel theoretical optimizations we make to the Stream X-Machine (SXM) test generation approach, and Section 4 formalizes the novel verification method used to check specifications. Section 5 describes the *BrokerCloud Verification and Testing Tool Suite*, and Section 6 develops a complete example of specifying, verifying and testing a cloud-based data warehouse and then summarizes

similar results for seven different case studies on three cloud platforms. Section 7 contrasts our approach with related work in testing service-oriented architectures, and Section 8 concludes with an evaluation of its benefits and future research opportunities.

2. SPECIFICATION LANGUAGE AND METHOD

There are several considerations when choosing a format for testable specifications that aims to become a useful standard for the cloud. Firstly, the format must be adequate to capture the semantics of the system under test, so that all deviations from required system behaviour may be detected during design and testing. Secondly, the format should be open and portable, communicable via the Internet and amenable to automatic machine processing at distributed locations. Thirdly, the format should be reasonably close to the culture of the community that is expected to use it, to encourage adoption.

2.1. Adequacy and acceptability criteria

In the web services community, the *de facto* standards for web protocols are the XML-based WSDL interfaces with SOAP data wrappers and the simpler HTTP-based REST interfaces with JSON data packets. Alternatively, the semantic web community offers MSM (the Minimal Service Model[§§]), a minimal extension to other RDF ontologies (GR, SKOS and FOAF) based on linked data principles [29]. These formats specify required interfaces and input/output data types but fail to capture the underlying state-related semantics of services, as has been noted many times in the literature [26, 30–33]. While the data types could be used to synthesize test inputs, there is no way of linking these to corresponding outputs, because there is no internal model of service behaviour.

Proposals for modelling service semantics have included UML state machines [34], OCL contracts [35], graph transformation rules [12] or dependency information [36]. The adoption of SAWSDL (semantic annotations for WSDL and XML [37]) catered to this trend, by supporting linkage from WSDL to arbitrary semantic documents. Two precursors that influenced our work [26, 32] used this approach to link WSDL interfaces respectively to the SWRL[¶¶] and RIF-PRD[‖‖] XML rule dialects to express the semantics of operations abstractly, in terms of their inputs, outputs, preconditions and effects (a style known as IOPE). Both approaches noted the affinity between the IOPE format and EFSMs; our earlier work [26] showed how a Stream X-Machine could in principle be extrapolated from the domain partitioning effect of the preconditions. Stream X-Machines are Turing-complete, so are adequate to model any software system [13].

Our chosen specification format is therefore one that unites the EFSM and IOPE views. We model a Stream X-Machine as a combination of the EFSM automaton and the IOPE protocol, linked through common labelling of transitions and guarded branches. The whole specification is an XML document, for reasons of Internet transmission. This is particularly relevant in cloud brokerage scenarios, where the *cloud broker* will host a collection of specifications and offer these as templates to potential service providers or as guarantees to service consumers. All three cloud roles (*broker, provider and consumer*) [2] will want to certify services in distributed locations at different points in the service lifecycle [6].

2.2. Overview of a service specification

In the following, we specify a simple bank account service, as a motivating example. The starting point is to create the state machine of the service, as shown in Figure 1(a). This machine has two control states (*closed and open*) that represent a high-level abstraction over the service's memory and various transitions, for example, *withdraw/ok, withdraw/blocked and withdraw/error*, indicating when particular operations are available. The absence of a transition indicates non-availability (interpreted as a null operation, rather than an error). The designer need only specify explicit transitions,

[§§]C. Pedrinaci, 2014: iServe vocabulary. http://iserve.kmi.open.ac.uk/wiki/Home.html#iServe_Vocabulary
[¶¶]SWRL: a semantic web rule language combining OWL and RuleML. https://www.w3.org/Submission/SWRL/
[‖‖]RIF production rule dialect (second edition), W3C recommendation: https://www.w3.org/TR/rif-prd/

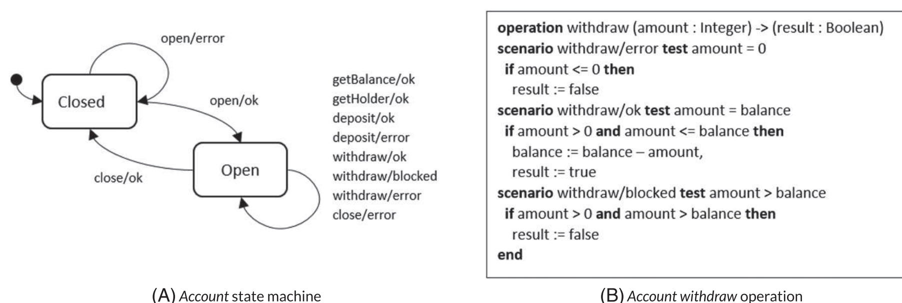


Figure 1. Visual rendering of the *Account* state machine and its *withdraw* operation.

for economy, and the tools later complete all missing transitions. The transition labelling indicates request/response pairs, where the same request may trigger a different response, in different memory state or input contexts.

The state machine is linked through this labelling to a more detailed protocol specification, which describes the memory and operations of the service. The memory declares a list of constants and variables, including the *balance* of the account. Operations have names such as *deposit and withdraw*, which correspond to requests submitted to the service. Each operation consists of a set of scenarios (cf. the UML sense of a single execution path), with labels such as *withdraw/ok, withdraw/blocked and withdraw/error*, which correspond to distinct request/response pairs. The scenario labels correspond exactly to the transition labels in the state machine. A scenario may specify an output, or an update to memory, or both. The IOPE protocol for the *withdraw* operation is shown in Figure 1(b).

In terms of the IOPE protocol [26, 32], all inputs and outputs for the operation are named explicitly (viz. *amount* and *result*), and each scenario is guarded by a precondition, which if satisfied, triggers an effect (*if . . . then . . .*). A precondition may examine any input or memory variable (respectively *amount* and *balance*), while an effect may bind any output or memory variable (respectively *result* and *balance*). An operation with only one scenario may have a trivial precondition *true*, otherwise all the scenarios of an operation must have mutually exclusive and exhaustive preconditions. Where no effect is specified, or fewer than the available variables are rebound, this is interpreted as a no-change axiom (avoiding the logical frame problem).

Another aspect that is relevant to automated testing is the inclusion of test input constraints to trigger each scenario (indicated by the *test* clause in Figure 1(b)). In principle, test inputs could be synthesized by analysing the precondition; however, there are cases where the constraint on inputs and memory is unsatisfiable (e.g. when the *balance* is zero and one scenario in Figure 1(b) is infeasible), so we prefer to allow the designer to suggest an input constraint that, under suitable memory conditions, will eventually trigger the scenario, as a way of limiting the search for test inputs.

The state machine in Figure 1(a) and the protocol fragment in Figure 1(b) are to be understood as visualizations of the XML specification. In principle, different tool vendors may provide their own editors for developing service specifications that render these in different visual styles.

2.3. *Service specification model*

A service specification is an XML document conforming to the XML schema *ServiceSchema.xsd* [25] visualized in Figure 2(a), where elements are indicated along with their required multiplicities. Each node in the figure corresponds to an XML element in the schema. Each element is also mapped to a corresponding class in a Java metamodel, which models the behaviour of that element. An XML specification may be unmarshalled directly to a Java model-instance, which is then capable of being analysed or manipulated by the various tools in the BrokerCloud Verification and Testing Tool Suite (Section 5). Further details of the mathematical and logical expression language are elaborated in Figure 2(b) and are described in Section 2.4.

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

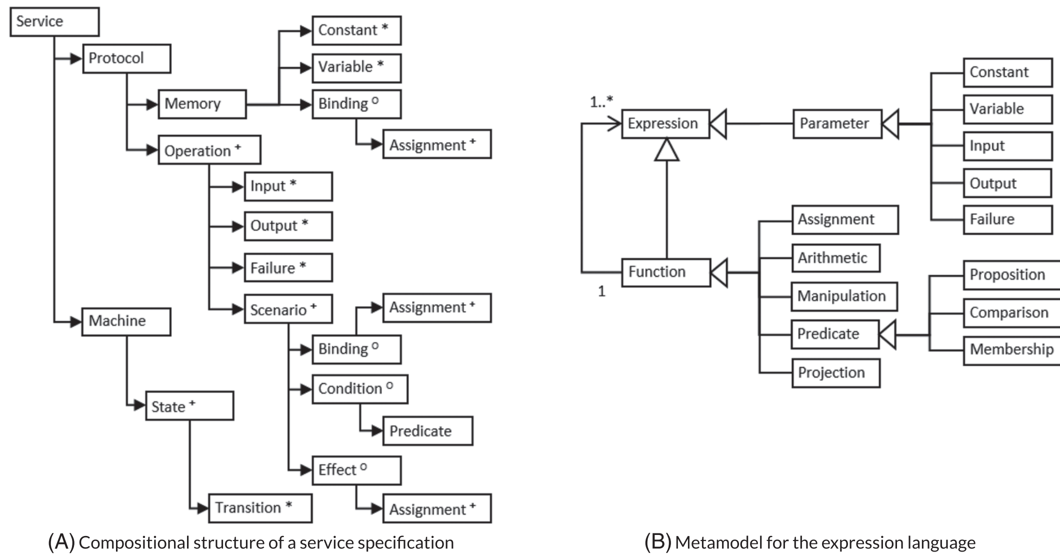


Figure 2. Compositional structure of a service specification and expression language metamodel.

As shown in Figure 2(a), a *Service* consists of a *Machine*, describing the control logic, and a *Protocol*, describing the functional logic. The *Machine* consists of one or more *States*, each of which specifies zero to many *Transitions* exiting that state. Exactly one *State* is marked as the initial state. The transitions correspond to events handled explicitly by the machine. Where a machine is not fully specified, missing transitions are treated implicitly as trivial cycles returning to the same state. Each *Transition* refers to its source and target *State* and is labelled with the name of the handled event, styled as a request/response pair. The same binary names are used to label *Scenarios* (described in the following paragraph) and so connect the *Machine* and *Protocol*.

The *Protocol* consists of a *Memory* and one or more *Operations*. The *Memory* is a tuple of *Constants* and *Variables*, with an initial *Binding* of values to variables. The signature of each *Operation* is described in terms of its *Inputs* and its *Outputs* (or *Failures*) and its executable body is described by one or more *Scenarios*. Each *Scenario* represents a distinct branching path, guarded by a mutually exclusive and exhaustive *Condition*, eventually triggered by an input *Binding* constraint. The resulting *Effect* is a posterior binding of *Outputs* (or *Failures*) and memory-*Variables* to values. The schema allows some elements to be optional in context; model-consistency is automatically checked and co-references are resolved when the XML specification is unmarshalled to a Java model-instance.

The earlier specification model is an extended finite state machine (EFSM). Because its transitions are atomic functions acting upon memory, it is also an X-Machine (XM) [38]. Because all atomic functions are triggered by inputs and memory, yielding outputs and updated memory, it is also a Stream X-Machine (SXM) [14, 15], a class of X-Machine that is fully testable under known design-for-test conditions. The correspondence with an SXM is obtained deliberately through the equivalence between a *Scenario* (a mutually exclusive guarded branch of an operation) and an *atomic function* in SXM theory [14], in order to leverage the power of the associated complete functional testing method [15–17].

2.4. Formal expression language

The specification language includes a mathematical language for describing Boolean, arithmetical and set-theoretic operations, inspired by the widely known Z notation [39]. The language supports built-in types (e.g. *Void*, *Boolean*, *Integer*, *Double*, and *String*) with all the usual primitive operations and arbitrary uninterpreted set-theoretic types (e.g. *Document* and *Person*) with equality. It supports Z's powerset, sequence, product and function types (constructed *Set[T], List[T], Pair[K,V]* and *Map[K,V]* types) with all their structural operations. It supports predicate logic over variable terms

Table I. Functions supported by each expression language meta-type.

Meta-type	Standard functions
Assignment	equals, lessThan, moreThan
Arithmetic	plus, minus, times, divide, modulo, negate
Projection	pair, first, second
Manipulation	size, insert, remove, insertAll, removeAll, searchAt, insertAt, replaceAt, removeAt
Comparison	equals, moreThan, lessThan, notEquals, notMoreThan, notLessThan
Membership	isEmpty, notEmpty, includes, excludes, includesAll, excludesAll, includesKey, excludesKey
Proposition	not, and, or, implies, equivalent

that are implicitly universally quantified but offers no explicit universal and existential quantifiers, in order to limit the complexity of verification. The specification style is similar to writing a Z specification. So a phone book recording phone numbers against names might be modelled as a *Map[String, Integer]* with a suitable initial binding to a constant representing the empty map.

The expression language is defined by a metamodel, shown in Figure 2(b). All *Expression*s are either *Parameter*s or *Function*s. The *Function*s are refined into further meta-types. Each meta-type describes a family of related functions, such as *Arithmetic* (all arithmetical functions) or *Manipulation* (all set-theoretic manipulations). The *Predicate* class (all Boolean-valued functions) is elaborated further into distinct kinds of Boolean predicate, distinguishing *Comparison* (scalar inequalities) from *Membership* (set-theoretic relations) and *Proposition* (Boolean compounds). The complete list of functions for each metatype is shown in Table I. These functions have standard names and are polymorphic: for example, the function *searchAt* returns an element mapped by a key in a *Map* or indexed by an *Integer* in a *List*. All functions are side-effect free, such that *insertAt* and *removeAt* return a fresh copy of the data structure in which the structural changes are manifest. *Assignment* must be used explicitly to rebind a variable to a new value; assignment can be exact (*equals*) or bind the variable to a boundary value just inside an exclusive limit (*lessThan* and *moreThan*).

An XML specification is unmarshalled as an instance of the earlier metamodel. The whole model specification is available to different reasoning tools. Not only can the state-transition graph be explored (cf. previous studies [33, 40]), but the expression language can also be simulated, both forwards and backwards. The different metaclasses support specific reasoning strategies through their meta-methods, for example, *Arithmetic* expressions can be simulated forwards for execution or backwards for constraint solving, *Comparison* expressions can be treated as symbolic values amenable to symbolic subsumption, and *Proposition* compounds can be split and complemented, to obtain atomic expressions. It is this capability that enables the novel verification algorithm (Section 4) on which the novel test generation algorithm depends (Section 3). In particular, all operations may be proven deterministic and complete before testing; and the whole Stream X-Machine may be simulated, updating memory and observing the blocking effects of guards, during test generation, such that all generated paths are known to be feasible.

3. TEST GENERATION APPROACH

In this section, we present the optimized Stream X-Machine testing method. X-machines were first explored as an interesting class of EFSMs, whose transitions are processing functions acting upon memory [38]. They were later found to be well suited for specifying complex software systems, through an ability to model the control and data of a system separately [13]. A fully controllable and observable testing method was later developed for the variant known as the Stream X-Machine (SXM), which includes input and output streams as part of memory [14, 15, 21].

The earliest application of the SXM testing method guaranteed correct integration of processing functions that were assumed to be individually correct [14, 15]. Later work used hierarchical SXMs to prove the correctness of complex systems recursively, using a divide-and-conquer approach [16]. Recent work has shown that it is possible to perform integration and component testing at the same time [41].

3.1. Stream X-Machine foundations

We introduce the following notation. For a finite alphabet Σ, Σ^* represents the set of all finite sequences with members in Σ. For sequences $a, b \in \Sigma^*$, ab denotes the concatenation of the two sequences a and b and ε denotes the empty sequence. For sets of sequences $U, V \subseteq \Sigma^*$, $UV = \{ab \mid a \in U, b \in V\}$ denotes the concatenated product. The language consisting of sequences of finite length U^n is defined by $U^0 = \{\varepsilon\}$ and $U^n = U^{n-1}U$, $n \geq 1$. The bounded Kleene star language consisting of all sequences up to length n is defined by $U[n] = U^0 \cup U^1 \ldots \cup U^n$. We assume that the reader is otherwise familiar with finite automata and related concepts, such as reachable states, distinguishable states, the minimal automaton and the accepted language (see Ipate [18] for a brief introduction).

A Stream X-Machine (SXM) differs from a simple FSM, in that it has internal data storage or *memory* (a tuple of variables), and its transitions are labelled by atomic *processing functions*, whose execution may be guarded, instead of simple input/output symbols.

Definition 1 (SXM)
A *Stream X-Machine* (SXM) is a tuple $Z = (\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ in which: Σ is the finite *input alphabet*; Γ is the finite *output alphabet*; Q is the finite *set of states*; M is a finite set called *memory*;[***] Φ is a finite set of distinct *processing functions*, where every $\phi \in \Phi$ is a non-empty (partial) function of the type $\phi : M \times \Sigma \longrightarrow \Gamma \times M$; and Φ is also known as the *type* of the SXM; F is the (partial) *next-state function* of the type $F : Q \times \Phi \longrightarrow Q$; $q_0 \in Q$ is the *initial state* and $m_0 \in M$ is the *initial memory value*.

Definition 2 (DSXM)
A *Deterministic Stream X-Machine* (DSXM) is an SXM in which, for every $\phi_1, \phi_2 \in \Phi$, if there exists $q \in Q$ such that $(q, \phi_1), (q, \phi_2) \in dom\ F$ then either $\phi_1 = \phi_2$ or $dom\ \phi_1 \cap\ dom\ \phi_2 = \emptyset$.

In the rest of this paper, we mostly consider DSXMs. A sequence $p \in \Phi^*$ of processing functions induces a function $||p||$ that shows the correspondence between a (memory, input sequence) pair and the (output sequence, memory) pair produced by the application, in turn, of the processing functions in the sequence p.

Definition 3
Given $p \in \Phi^*$, $||p|| : M \times \Sigma^* \longrightarrow \Gamma^* \times M$ is defined by

- $||\epsilon||(m, \epsilon) = (\epsilon, m), m \in M$
- Given $p \in \Phi^*$ and $\phi \in \Phi$, $||p\phi||(m, s\sigma) = (g\gamma, m')$, for $m, m' \in M$, $s \in \Sigma^*$, $g \in \Gamma^*$, $\sigma \in \Sigma$, $\gamma \in \Gamma$ such that there exists $m'' \in M$ with $||p||(m, s) = (g, m'')$ and $\phi(m'', \sigma) = (\gamma, m')$.

A *computation* of the SXM Z represents the traversal of all transition sequences in the associated automaton A_Z and the application of all the corresponding processing functions. These are applied successively, consuming inputs, possibly updating memory and producing outputs. The correspondence between the input sequence and the output produced gives rise to the relation (or function) computed by Z.

Definition 4
The relation computed by SXM Z, $f_Z : \Sigma^* \longleftrightarrow \Gamma^*$ is defined by $(s, g) \in f_Z$ if there exist $p \in \Phi^*$ and $m \in M$ such that $(q_0, p) \in dom\ F^*$ and $||p||(m_0, s) = (g, m)$. We say that Z computes f_Z. Note that for a DSXM Z, the relation f_Z is a function, that is, $f_Z : \Sigma^* \longrightarrow \Gamma^*$.

Definition 5
An SXM Z is said to be *completely defined* if $dom\ f_Z = \Sigma^*$.

[***] Usually in SXM theory the memory is considered a *possibly infinite* set. However, in order to ensure decidability, in this context, we will consider the memory finite.

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

DOI: 10.1002/stvr

In other words, an SXM is completely defined if every sequence of inputs can be processed by at least one sequence of functions accepted by the associated automaton. An SXM that refuses some inputs can always be transformed into a completely defined one by adding a distinct error output (not in the output alphabet) and completing the automaton with self-looping transitions (ignored events), or alternatively, transitions to an extra error-state (fatal errors).

3.2. The W-method for testing finite automata

The fundamental DSXM testing method [42] is an adaptation of Chow's W-method for testing finite automata (FA) [19]. This assumes naturally that the specification and implementation have the same input alphabet Σ and testing seeks to ensure that every path in the specification exists in the implementation (both accepted and refused paths, for robust positive and negative testing). Apart from this, whereas the specification is minimal with $n > 0$ states, the implementation may contain $n' \geq n$ estimated states. Other important notions include the following:

- a *state cover*, a set V consisting of sequences that reach every state of the machine; V is either chosen or determined by exploring the automaton;
- a *transition cover*, a set T consisting of sequences that reach every state of the automaton and then exercise every transition in the alphabet from that state; the transition cover can be computed by $T = V \cup V\Sigma$; and
- a *characterization set*, usually labelled W, that distinguishes between every pair of states, according to whether sequences from W are accepted or refused from that state.

Given the earlier definitions, test suites may be defined according to the W-method formula:

$$Wtest(\Sigma, V, W, n', n) = V\Sigma[n' - n + 1]W = V(\{\varepsilon\} \cup \Sigma \ldots \cup \Sigma^{n'-n+1})W.$$

The idea behind this is that the product $V\Sigma[n' - n + 1]$ will exercise at least the transition cover (where $n' = n$, this is equal to $V\Sigma[1] = V \cup V\Sigma = T$) and the final product with W ensures that the implementation reaches the same state as expected by the specification. In case the implementation contains $n' - n > 0$ extra states, longer test sequences up to length $n' - n$ ensure that these states are reached and behave like duplicates of the expected states.

The W-method requires a reliable *reset* in the implementation that places the system in its initial state, before each test sequence is executed, but requires no direct state inspection, relying instead on W to identify states. It is robust in identifying correct and incorrect paths and states. The transition tour method [22] avoids reset but cannot guarantee correct state transfer (and the tour may not exist). Similarly, the unique input–output (UIO) method [23] cannot reliably test for unwanted states but is less expensive than W, using single state-identification sequences.

3.3. Adaptation for DSXMs using design-for-test conditions

Because the transitions of a DSXM represent (partial) functions $\phi \in \Phi$, rather than inputs $\sigma \in \Sigma$, the adapted W-method therefore constructs sequences of atomic processing functions rather than sequences of inputs:

$$Wtest(\Phi, V, W, n', n) = V\Phi[n' - n + 1]W = V(\{\varepsilon\} \cup \Phi \ldots \cup \Phi^{n'-n+1})W.$$

Each (partial) processing function $\phi \in \Phi$ may have a restriction on input/memory that may prevent it from firing unconditionally. Therefore, test sequences which cover all states and transitions of the associated automaton might not reach all states or transitions in the DSXM, due to the blocking effects of guards. Certain extreme settings of memory may be hard to reach, leading to theoretical treatments of the *controllability* or *input completeness* of the SXM [15, 43]. Similarly, the state cover of the automaton must be replaced by a realizable *r-state cover* in the DSXM [20, 21]. Likewise, states which are distinguishable using W in the automaton may require function sequences that can never be applied, due to blocking; W must be replaced by realizable *separating sets* [21]. To ensure that tested systems are controllable and observable, it is therefore necessary to adopt a number of design-for-test conditions.

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

DOI: 10.1002/stvr

Definition 6
A DSXM Z is called *input complete* if $\forall \phi \in \Phi, m \in M, \exists \sigma \in \Sigma$ such that $(m, \sigma) \in dom\ \phi$.

The *input completeness* (or *controllability*) of a DSXM assures that any sequence of processing functions in the associated automaton can be triggered by suitable input sequences. This property is rather strict; most real-world systems are not by default input complete. Testable systems must admit special inputs, used only during testing, that circumvent the blocking effect of guards and drive the system directly into extreme memory states. Recent work has relaxed *controllability* only slightly, replacing this by *input uniformity* [20], a property that requires concrete inputs to be found, one at a time, for each processing function in a sequence.

Definition 7
A DSXM Z is said to be *output distinguishable* if for all $\phi_1, \phi_2 \in \Phi$, whenever there exist $m, m_1, m_2 \in M, \sigma \in \Sigma, \gamma \in \Gamma$ such that $\phi_1(m, \sigma) = (\gamma, m_1)$ and $\phi_2(m, \sigma) = (\gamma, m_2)$, then $\phi_1 = \phi_2$.

The *output distinguishability* property assures that it is possible to determine which atomic processing function was applied, from the output produced in response to any given input. This is an important test oracle, serving to determine whether the correct or incorrect function was triggered in the implementation. However, many real-world systems do not produce distinguishing outputs for every action. Testable systems must in practice instrument their operations to produce extra output symbols, where needed.

3.4. Fundamental DSXM testing theorem

Given a finite automaton (FA) specification A and a class of implementations C, a *test set* is a set of input sequences that, when applied to any implementation A' in the class C, will detect any response in A' that does not conform to the response specified by A. We show in the following discussion how this definition generalizes to DSXMs. In the following, L_A is the language accepted by the automaton A, A_Z is the associated automaton of the DSXM Z and L_{A_Z} is the language accepted by this automaton.

Definition 8
Let A be a deterministic FA and C a set of deterministic FAs having the same input alphabet as A. Then, a finite set $Y \subseteq \Sigma^*$ is called a *test set* of A w.r.t. C if $\forall A' \in C, (L_A \cap Y = L_{A'} \cap Y) \Rightarrow L_A = L_{A'}$.

Similarly, for DSXM, a *test set* is a finite set of input sequences constructed from the DSXM specification that produces identical results when applied to the specification and the implementation only if the specification and the implementation compute identical functions.

Definition 9
Let Z be a DSXM and C a set of DSXMs having the same input alphabet Σ and output alphabet Γ as Z. Then, a finite set $X \subseteq \Sigma^*$ is called a *test set* of Z w.r.t. C if $\forall Z' \in C, (f_Z \mid X = f_{Z'} \mid X \Rightarrow f_Z = f_{Z'})$.

Definition 10
Two DSXMs Z and Z' are called *weak testing compatible* if they have identical input alphabets, output alphabets, memory sets and initial memory values. Two weak testing compatible DSXMs are called *testing compatible* if they have identical types, namely, their corresponding sets of processing functions Φ and Φ' are identical.

The W-method generates sequences of inputs (for the FA) or processing functions (for the DSXM) from the specification. However, in DSXM testing, these abstract sequences must first be converted into concrete test inputs (requests with their actual parameters), in order to test implementations. For this, we assume the existence of a *test function* that translates sequences of processing functions into sequences of inputs.

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

DOI: 10.1002/stvr

Definition 11
A *test function* of an SXM Z is a function $t : \Phi^* \longrightarrow \Sigma^*$ that satisfies the following conditions:

- $t(\varepsilon) = \varepsilon$ (1)
- Let $\rho = \phi_1 \dots \phi_k \in \Phi^*, k \geq 1$
  - Suppose $\phi_1 \dots \phi_{k-1} \in L_{A_Z}$ and there exists $\sigma_1, \dots, \sigma_k \in \Sigma$, $\gamma_1, \dots, \gamma_k \in \Gamma$ and $m_1, \dots, m_k \in M$ such that $\phi_i(m_{i-1}, \sigma_i) = (\gamma_i, m_i)$, $1 \leq i \leq k$. Then, $t(\rho) = \sigma_1 \dots \sigma_k$ for some $\sigma_1 \dots \sigma_k$ that satisfy this condition (2)
  - Otherwise, $t(\rho) = t(\phi_1 \dots \phi_{k-1})$. (3)

The *test function* associates a sequence of inputs that exercises the longest prefix of $\phi_1 \dots \phi_n$ that is a path in the SXM and, if $k < n$, also exercises ϕ_{k+1}, the function that follows after this prefix. If the type Φ is input complete, the input sequence $\sigma_1, \dots, \sigma_n$ will always exist. Otherwise, the sequences produced may not all be feasible in the DSXM and consequently they cannot be mapped into actual input values. In this case, as Definition 11 case (3) specifies, only the longest subsequence will be mapped into actual input values. Furthermore, the *test function* of a DSXM is not uniquely determined; many suitable input sequences may exist. Notwithstanding how the *test function* is to be constructed (an issue that is elided in the SXM-testing literature, which we address in Sections 2 and 5), the earlier considerations lead to the expression of the fundamental DSXM testing theorem. This is the basis for a number of important results, such as the guarantee of correct integration in the divide-and-conquer approach [15, 16].

Theorem 1
[42] Let A be a deterministic FA having input alphabet Σ, n the number of states of A, $n' \geq n$ and $C_{n'}$ the set of deterministic FAs having input alphabet Σ whose number of states does not exceed n'. If T is a transition cover and W a characterization set of A, then $Y_{n'-n} = T(\Sigma^{n'-n} \cup \dots \cup \{\varepsilon\})(W \cup \{\varepsilon\})$ is a test set of A w.r.t. $C_{n'}$.

Theorem 2
[15, 16] Let Z be a DSXM having type Φ input complete and output distinguishable and C a set of DSXMs testing compatible with Z. If t is a test function of Z and $Y \subseteq \Phi^*$ a test set of A_Z w.r.t. A_C, where $A_C = \{A_{Z'} \mid Z' \in C\}$, then $X = t(Y)$ is a test set of Z w.r.t. C.

3.5. DSXM testing improvements and optimizations

Our optimized test generation method makes slightly different assumptions. We do not require strong *input completeness* to control the DSXM but instead require the test generator to produce only *feasible sequences* that eventually will cover all transitions and states, using a *test function* that is built into the specification. This supports testing real-world systems that are not input complete. Similarly, we finesse the *output distinguishability* criterion and the *state separation* criterion through different design-for-test assumptions about the system under test (SUT):

1. The SUT has a reliable *reset* function r that is guaranteed to place the SUT in its initial state and memory bindings. We do not include r in Φ but assume it is executed before every test sequence. This is the same requirement as for the W-method.
2. The SUT has a reliable *log* function g, which reports which processing function in Φ was triggered in response to the most recent request, without modifying the SUT. We do not include g in Φ but assume it may be executed after any sequence. This implementation detail ensures *output distinguishability* via a side-channel, so does not interfere with the SUT's natural outputs.
3. The SUT has a reliable *state oracle* function s, which reports the current state of the SUT, without modifying the SUT. We do not include s in Φ but assume it may be executed after any sequence. This implementation detail ensures *state separation* without need for the W state characterization set, so does not interfere with the SUT's state after executing a given sequence of processing functions.

These features can be added easily to service-oriented systems, which often already supply the additional observers as part of their internal diagnostic systems. The observer functions g, s can be invoked immediately after any transition has been fired, respectively, to observe which $\phi \in \Phi$ was triggered and which state $q \in Q$ was reached. Where unique test sequences are built systematically, these observers need only be checked at the end of each sequence, because all prefix sequences will already have been checked. However, it is possible to interleave functions and observations when merging multi-objective test sequences (Section 5). If the *test function* $t(\phi)$ is deterministic, the memory of the SUT is uniquely determined by the sequence of processing functions $\phi \in \Phi$ that were exercised, so the observers g, s are sufficient to confirm the memory bindings.

Our improved testing method not only tests for correct integration of component functions [15, 16] but also performs equivalence-partition testing. The linked verification method (Section 4) ensures, by symbolic reasoning, that every possible input partition is handled by exactly one operation branch (viz. DSXM processing function), which must be tested at least once. Test input constraints, assumed from domain knowledge, cause each branch eventually to be executed, discharging the assumption through test coverage reports.

3.5.1. Test generation replacing W by a state oracle. As a precursor to later optimizations (such as test compression by merging), we replace the characterization set W by an abstract state oracle function s. What is important is to preserve the existing test properties of the W-method, as justified by the following lemma.

Lemma 1
If the SUT satisfies the design for test conditions, having a reliable state oracle s, and V is a feasible *r-state cover* for the DSXM Z that has n states, and the SUT has at most n' states, then the set $S = Wtest(\Phi, V, \{s\}, n', n)$ is a test set, where $S \subseteq \Phi^* \cdot \{s\}$.

Proof
Intuitively, the state oracle function s is replacing the characterization set W in the set $Wtest(\Phi, V, W, n', n) = V\Phi[n' - n + 1]W$, which is a test set for the associated automaton A_Z, according to Theorem 1. The set becomes $Wtest(\Phi, V, \{s\}, n', n) = V\Phi[n' - n + 1]\{s\}$ and it is still a test set for the automaton. □

Note that the product with singleton set $\{s\}$ does not multiply the number of generated sequences. The size of the generated test set is therefore reduced by a factor of $card(W)$, with respect to the set generated by the original W-method. The earlier treatment must also be mapped via a *test function* to yield suitable test inputs to drive the DSXM. For this, we require an extended version of the *test function* that also verifies states. Where $\phi_i \in \Phi$ are processing functions of the DSXM Z, and $s \notin \Phi$ represents the state oracle:

Definition 12
An *extended test function* for a DSXM Z and a state oracle s is a function $t' : \Phi^* \cdot \{s\} \longrightarrow \Sigma^* \cdot Q$ that satisfies the following condition: $t'(\phi_1 \dots \phi_n \cdot s) = t(\phi_1 \dots \phi_n) \cdot q$, where t is a test function for Z and $q \in Q$ is the state reached in A_Z after processing $t(\phi_1 \dots \phi_n)$.

Lemma 2
If $Wtest(\Phi, V, \{s\}, n', n)$ is a test set of the associated automaton A_Z, then $t'(Wtest(\Phi, V, \{s\}, n', n))$ will be a test set of the DSXM Z.

Proof
By the equivalence presented in Definition 12, the conditions of Theorem 2 hold and the corresponding set $t'(Wtest(\Phi, V, \{s\}, n', n))$ will be a test set of Z, where a sequence from this set $\phi_1 \dots \phi_n \cdot s$ will be used for testing. □

3.5.2. Elimination of redundant and infeasible sequences. An important property of the sets of exploratory paths generated by our algorithm is that they are *prefix closed*, that is, all the prefixes of a given path exploring from a given state are also in the same test set. They are also *path complete*, that

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

DOI: 10.1002/stvr

is, all possible alternative paths of a given length are explored from each state, before optimization. These properties apply to the paths explored, not necessarily to the state cover prefix.

Definition 13
We say that a language $L \subseteq \Phi^*$ is *prefix closed* if all the prefixes ρ' of non-empty $\rho \in L$ are also included in L, that is, $\forall(\rho|\rho \neq \epsilon) \in L \cdot (\rho = \phi_1 \ldots \phi_n, n > 0) \cdot \forall(k|1 \leq k < n) \cdot (\rho' = \phi_1 \ldots \phi_k) \in L$.

Definition 14
We say that a language $L \subseteq \Phi^*$ is *path complete* if every non-empty $\rho \in L$ has an immediate prefix ρ' and for all ϕ in Φ, every alternative sequence $\rho'\phi$ is also included in L, that is, $\forall(\rho|\rho \neq \epsilon) \in L \cdot (\rho = \phi_1 \ldots \phi_{n-1}\phi_n, n > 0) \cdot \forall\phi \in \Phi \cdot (\rho' = \phi_1 \ldots \phi_{n-1})\phi \in L$.

Remark 1
The language $\Phi[n]$ is *prefix closed* and *path complete* by construction. This is by definition of $\Phi[n]$ and Φ^n, which build the result by breadth-first exploration. The language $\Phi[n]$ also has the property that if $\rho = \rho_1\phi\rho_2 \in \Phi[n]$, then $\rho' = \rho_1\rho_2 \in \Phi[n]$.

The first optimization is to remove redundant test sequences, which test properties that have already been confirmed by other test sequences in the same test set. Consider that an automaton A_Z which blocks for some events in some states can always be *completed* by adding explicit trivial transitions representing the ignored events (service-oriented systems are designed this way, to avoid blocking). Such transitions ϕ_ϵ denote trivial functions (nullops), having no effect upon the SUT, and are circular, returning to the same state.

Proposition 1
Suppose that the automaton A_Z has been completed, that $\rho_1, \rho_2 \neq \epsilon$ are sequences and that the test set $S \subseteq V\Phi[n]$ contains the path $\rho = \rho_1\phi_\epsilon\rho_2$, where ϕ_ϵ is a trivial transition. Then, a shorter path $\rho' = \rho_1\rho_2 \in S$ will test the same properties as the original path ρ. We say that ρ is a *redundant* sequence, which may be deleted from the test set, without loss of coverage. That is, if S is a test set of A_Z, then $S \setminus \{\rho\}$ is also a test set.

Proof
Trivial ϕ_ϵ cannot occur in the minimal state cover V but only in exploratory sequences generated by $\Phi[n]$. The prefix sequences $\rho_1\phi_\epsilon$ and ρ_1 must leave the SUT in the identical state and memory configuration, because ϕ_ϵ is a trivial transition, which by definition has no effect on the SUT. Because all exploratory paths are *prefix closed*, then the triviality of ϕ_ϵ will have been determined by testing $\rho_1\phi_\epsilon$, and the condition $\rho_2 \neq \epsilon$ ensures that this sequence is not deleted. Because the exploratory paths are *path complete* and the test set includes ρ, it will also already include ρ', by the property derived in Remark 1. □

The second optimization is to remove infeasible test sequences, which cannot be executed in the SUT. Previously, the classic DSXM testing method either forced all sequences to be feasible (by *input completeness*) or truncated test sequences after their maximally feasible prefix. Our test generator deletes infeasible sequences, on the basis that *prefix closed* test sets include the maximally feasible prefix, and *path complete* test sets supply witnesses for blocked paths by other means.

Proposition 2
Suppose that the memory and input type $M \times \Sigma$ is exhaustively partitioned into n equivalence classes $(m_i, \sigma_i), 1 \leq i \leq n$ and that an operation $Op = \{\phi_1 \ldots \phi_k\}, k \leq n$ consists of a set of related processing functions, handling similar requests, such that $\forall(m_i, \sigma_i) \in M \times \Sigma \cdot \exists!\phi_j \in Op \cdot \phi_j(m_i, \sigma_i) = (\gamma_j, m_j)$. Then, if some path $\rho = \rho_1\phi_i\rho_2 \in S$ is found to be infeasible when ϕ_i blocks at some (m_i, σ_i), there will always be exactly one feasible ϕ_j that can execute instead. Therefore, if $S \subseteq V\Phi[n]$ is test set, then $S \setminus \{\rho\}$ is a test set.

Proof
Blocking ρ cannot occur in the feasible state cover V but only in the exploratory sequences gen-

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

DOI: 10.1002/stvr

erated by $\Phi[n]$. The different $\phi_i \in Op$ are mutually exclusive and exhaustive, by definition. If the path $\rho = \rho_1 \phi_i \rho_2 \in S$ blocks at ϕ_i, then by virtue of *prefix closure*, a maximally feasible prefix ρ_1 exists and will be tested. By virtue of *path completeness* and the property derived in Remark 1, some other path $\rho_1 \phi_j \rho_2$ will also exist, where the accepted prefix $\rho_1 \phi_j$ is a witness to the blocking prefix $\rho_1 \phi_i$ by mutual exclusion. □

3.5.3. Merging test sequences with shared test objectives. The third optimization compresses the test suite. Shorter sequences are merged with longer sequences of which they are a prefix. The test objectives of a path are normally checked by assertions, added after all execution steps. Merged sequences, which are multi-objective tests, may also have assertion checks interleaved with the path's execution steps. We define an assertion check α as a side-effect-free function that inspects the result of the triple (q, ϕ, γ), where $q \in Q$ is returned by the *state oracle* s, $\phi \in \Phi$ is the last triggered function returned by the log g and $\gamma \in \Gamma$ is the output of the last ϕ. We call any path ending with an assertion check a *checked path*, and any test set consisting of checked paths a *checked test set*.

Proposition 3
Suppose that assertions $\alpha \in \Lambda$ are side-effect free and may be added onto the end of any sequence in S. Suppose that S' is a checked test set, which contains the checked path $\rho_1 = \phi_1 \dots \phi_k \dots \phi_n \alpha_n$ and the checked prefix $\rho_2 = \phi_1 \dots \phi_k \alpha_k$. If a merged sequence $\rho_3 = \phi_1 \dots \phi_k \alpha_k \dots \phi_n \alpha_n$ is included in S', this meets the test objectives of ρ_1, ρ_2, which may be removed. That is, if $S' \supseteq \{\rho_1, \rho_2\}$ is a checked test set, then $S' \cup \{\rho_3\} \setminus \{\rho_1, \rho_2\}$ is also a checked test set.

Proof
Because assertions $\alpha \in \Lambda$ are side-effect free, medial insertion of α_k into ρ_3 w.r.t. ρ_1 has no effect on the state or memory of the SUT, so the check α_n will yield the same result in ρ_3 as it did in ρ_1. The sequence ρ_2 is a prefix of ρ_3. Therefore, ρ_3 checks the objectives of ρ_1 and ρ_2. □

The various optimizations described earlier achieve significant test set size reductions reported in Section 5. The test compression optimization could not work without replacing W, because products with W tend to produce sequences that are not prefixes of other sequences; whereas with *prefix closed* exploratory sequences, there are many prefixes of other sequences. After eliminating redundant sequences with trivial prefix cycles, exploratory sequences are still *prefix closed*, but after removing infeasible sequences, they are no longer *path complete*.

4. VERIFICATION APPROACH

In this section, we present a novel verification approach for Stream X-Machines, based on the notion of making the IOPE protocol of the SXM explicit. Verification ensures that the DSXM protocol is correct, in that its operations are *complete* (non-blocking), always having a response for any input/memory combination, and *deterministic*, having a single response for any such input/memory combination.

Apart from the fact that it is desirable to have a correct specification, verification ensures that the DSXM specification meets the necessary conditions for applying the DSXM testing method (Section 3), which expects a deterministic and complete SXM. Furthermore, specific test optimizations also require this property, to ensure that removing infeasible paths still leaves a witness to blocked paths, based on guaranteed mutual exclusion of certain functions.

4.1. Protocol specification

The associated protocol P_Z for a DSXM Z is an abstraction over its operations and memory states. The protocol is related to, but distinct from, the associated automaton A_Z. Whereas A_Z dictates through its explicit transitions when an operation is valid (invalid requests are ignored), P_Z dictates how valid requests should respond, given the current memory state. The protocol is a tuple:

$$\mathrm{P_Z = (F, M, m_0)},$$

in which F is a set of operation specifications; M is specification of memory and m_0 is the initial state of memory. The memory tuple $m : M \subseteq T^*$ is a finite product of constants and variables $c_i, v_j \in T$, where the type T is restricted to finite computational types taken from the type domain:

$$\mathrm{T ::= Obj \mid Bool \mid Num \mid Char^* \mid List[T] \mid Set[T] \mid Map[T_k, T_v]},$$

where Obj is the union of finite uninterpreted sets used to model object identifiers, $Bool$ is the Boolean type, Num is the union of finite computational numeric types including the usual short and long precision versions of the IEEE integral and floating point numbers, $Char^*$ is the type of bounded-length character strings, $List[T]$ is the type of bounded-length lists of elements of type T, $Set[T]$ is the type of finite sets of elements of type T and $Map[T_k, T_v]$ is the type of finite maps from T_k to (distinct) T_v. All types are finite to ensure decidability.

An operation $f \in F$ is one of the declared operations of the service. Each operation is a tuple:

$$\mathrm{f = (req_f, in_f : T^*_{fin}, out_f : T^*_{fout} \cup Err, \Phi_f)},$$

in which req_f is a unique label or name for the operation f denoting a request, in_f is a finite input tuple for the operation f consisting of values taken from the domain type T^*_{fin} and out_f is a finite output tuple for the operation f consisting either of values taken from the codomain type T^*_{fout}, or of a single value from Err, the set of exceptions including $\perp$ the undefined result. The input and output tuples in_f, out_f may be zero-length to indicate respectively a no-input or no-output operation.

Finally, $\Phi_f \neq \emptyset$ is a non-empty finite set of atomic functions $\phi_{f_j} \in \Phi_f$, known as the *scenarios* of operation f, denoting its distinct execution paths as specified by the designer. Each scenario $\phi_{f_j} \in \Phi_f$ may also be described as a tuple:

$$\phi_{\mathrm{f_j}} = (rsp_{\mathrm{f_j}}, \mathrm{g_{f_j}} : (\mathrm{T^*_{fin}} \times \mathrm{M} \to \mathrm{Bool}), \mathrm{beh_{f_j}}),$$

in which rsp_{f_j} is a unique label within function f, denoting the distinct response described by the scenario ϕ_{f_j}, g_{f_j} is a Boolean guard testing the function's input domain and memory, yielding a response which, when true, indicates that the scenario ϕ_{f_j} should be triggered and beh_{f_j} is the behaviour performed by the scenario ϕ_{f_j}, which is not analysed further here. The behaviour specifies posterior variable bindings, such as outputs or memory updates, and can also be empty.

4.2. *Protocol-automaton congruence*

The protocol P_Z must describe the same system as the automaton A_Z of the DSXM Z. The set of processing functions $\phi \in \Phi$ labelling the transitions in the automaton A_Z is the union of the sets of scenarios for each operation $\phi_{f_j} \in \Phi_f$, intuitively $\Phi = \bigcup \Phi_f$. It must be possible to map from any explicitly specified transition to a unique scenario, describing its behaviour; symmetrically, any specified scenario must correspond to a unique function labelling at least one explicit transition in the automaton.

Lemma 3
A scenario may be identified uniquely by a label pair: (req_f, rsp_{f_j}), where req is the name of an operation $f \in F$ and $resp$ is the name of a scenario $\phi_{f_j} \in \Phi_f$, the branches of operation f.

Proof
By definition, the label req_f uniquely identifies a distinct operation request $f \in F$ and within the set of scenarios Φ_f of f, the label rsp_{f_j} uniquely identifies a distinct scenario response, also by definition. □

Definition 15
We say that the protocol P_Z and the automaton A_Z of a DSXM Z are *congruent* if every explicitly labelled transition $\phi \in \Phi$ (before the completion of A_Z with trivial transitions) corresponds by name to exactly one scenario $\phi \in \bigcup \Phi_f$, and symmetrically, if every scenario corresponds by name to a function ϕ labelling one or more explicit transitions.

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

Protocol-automaton congruence is checked directly through model inspection, by assuring the equivalence of the two sets of labels: $labels(transitions(states(A_Z))) \equiv labels(scenarios(operations(P_Z)))$. We established earlier that transitions were labelled in the same binary style (req, rsp). Note that the same label may sometimes be used on more than one transition (denoting the same function, but targeting different states). Note also that trivial transitions play no part in this consideration, because the protocol P_Z only specifies valid requests.

4.3. *Memory initialization and test input completeness*

For the protocol to simulate correctly, the memory must be initialized and suitable test inputs must be found for every $\phi \in \bigcup \Phi_f$. In a departure from traditional DSXM approaches, which presume the existence of a *test function* to generate inputs, we require a test input binding constraint in the specification, a pragmatic choice to limit the search performed by the constraint solver. The binding constraint is expressed as a simple inequality, for example, $v_1 = c_1$ or $v_1 < v_2$ and is used by the constraint solver to bind v_1 to a suitable value.

Definition 16
If the memory type $M \subseteq T^*$ defines a product of constants and variables $c_i, v_j \in T$, then we say memory is *initialized* if an initial binding exists for every declared variable in initial memory, that is, $\forall v_i \in m_0 \cdot \exists c_j \in m_0 \cdot (v_i := c_j) \in init$, where $init$ is the set of initial bindings. The types must be compatible, but there may be fewer constants than variables.

Definition 17
Similarly, every scenario $\phi_{f_k} \in \Phi_f$ must accept a distinct input tuple $in_{f_k} : T^*_{fin}$ specified in the $test$ bindings. A test binding must exist for each input variable (state variables are already bound), that is, $\forall v_i \in in_{f_k} \cdot \exists c_j \cdot (v_i := c_j) \in test$, where $test$ is the set of test bindings and c_j is found by constraint satisfaction. Furthermore, we say that a test binding is *input complete* if, under suitable memory conditions m_i, the guard g_{f_k} for scenario ϕ_{f_k} may be true, that is, $\forall \phi_{f_k} \in \Phi_f \cdot \exists m_i : M, \exists in_{f_k} : T^*_{fin} \cdot g_{f_k}(m_i, in_{f_k}) = true$, where $g_{f_k} = guard(\phi_{f_k})$.

The existence of initial bindings of memory and test input bindings for each scenario is checked directly through model inspection. Test input completeness is initially assumed, after manually picking suitable test bindings that will satisfy the guards g_{f_k}. This assumption is later discharged by protocol simulation, when the test generation tool checks that every scenario was executed at least once, so satisfying the test *input completeness* condition. For functions with no inputs, no test input binding is needed, because any guards may only test memory.

4.4. *Operation completeness and determinism*

To satisfy the testing assumptions for the DSXM Z, the associated protocol P_Z must specify deterministic and complete behaviour over all inputs and memory. To verify this, we must show that every operation $f \in F$ individually specifies deterministic and complete behaviour over memory M and its own domain type T^*_{fin}.

Definition 18
Where an operation $f \in F$ consists of a set of scenarios $\phi_{f_i} \in \Phi_f$, we say that f is *complete* if, for all inputs and memory, at least one $\phi_{f_i} \in \Phi_f$ may be triggered; similarly, we say that f is *deterministic* if, for all inputs and memory, at most one $\phi_{f_i} \in \Phi_f$ may be triggered. This is equivalent to imposing a mutually exclusive and exhaustive condition on the guards g_{f_i} of each scenario, namely, $\forall in_f : T^*_{fin}, \forall m : M \cdot \exists! g_{f_i} \in G_f \cdot g_{f_i}(m, in_f) = true$, where $G_f = \{guard(\phi_{f_i}) \mid \phi_{f_i} \in \Phi_f\}$.

Verifying this property is the main task of the verification tool. We convert a possibly large state-space search problem into a finite symbolic checking problem. The key insight is that a guard is a predicate $g_{f_i} : M \times T^*_{fin} \rightarrow Boolean$ whose domain type can be divided completely into equivalence partitions, based on the partitioning effects of predicates used in the complete set of guards

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

DOI: 10.1002/stvr

$g_{f_i} \in G_f$ for the operation f. These equivalence partitions may be represented as conjunctions of atomic predicates, which may then be tested against the guards by symbolic subsumption.

In the sequel, we first deconstruct the guards $g_{f_i} \in G_f$ to reveal the *atomic predicates* acting on a subset of relevant variables; then we recombine the *atomic predicates* in all possible memory/input combinations; then we eliminate *inconsistent partitions* by constraint solving; and finally, we satisfy Definition 18 by *symbolic subsumption*.

4.4.1. Deconstruction of guards into atomic predicates. Guards are constructed from a finite collection of predicates available in the specification's expression language (Table I). Predicates fall into three basic meta-types $Comparison$, $Membership$ and $Proposition$, plus the degenerate $Atom$, denoting a Boolean value. Complex $Predicate$s are defined inductively over constants c, variables v, collections s and (recursively) predicates p:

$$\begin{aligned}
\text{Atom} &::= \text{c, v} \in \text{Boolean} \\
\text{Comparison} &::= \text{cmp(v, c)} \mid \text{cmp(c, v)} \mid \text{cmp(v, v)}, \text{ where v, c} \in \text{Num} \cup \text{Char}^* \\
\text{Membership} &::= \text{mbr(s)} \mid \text{mbr(v, s)} \mid \text{mbr(c, s)} \mid \text{mbr(s, s)}, \text{ where v, c} \in \text{T, s} \in \text{Set[T]} \cup \text{List[T]} \cup \text{Map[T, T]} \\
\text{Proposition} &::= \text{pro(p)} \mid \text{pro(p, p)}, \text{ where p} \in \text{Atom} \cup \text{Comparison} \cup \text{Membership} \cup \text{Proposition} \\
\text{Predicate} &::= \text{Proposition} \mid \text{Membership} \mid \text{Comparison} \mid \text{Atom.}
\end{aligned}$$

This inductive definition supports deconstruction of a guard into a set of irreducible constraints on relevant variables that affect its outcome. We define a structure-matching recursive function $atomic()$ that deconstructs all compound logical formulae $pro \in Proposition$:

$$\begin{aligned}
&\text{atomic : Predicate} \rightarrow \text{Set[Predicate]} \\
&\quad \text{atomic(pro(p))} = \text{atomic(p)} && \text{-- --unary proposition} \\
&\quad \text{atomic(pro(p, q))} = \text{atomic(p)} \cup \text{atomic(q)} && \text{-- --binary proposition} \\
&\quad \text{atomic(p)} = \{\text{p}\} && \text{-- --everything else}
\end{aligned}$$

The deconstructed set contains only $Atom$s and $Atomic$ predicates defined over scalar and set-theoretic types, where the meta-type $Atomic ::= Comparison \mid Membership$. $Atomic$ predicates denote irreducible scalar inequality or set-theoretic constraints over variables; $Atom$s are Boolean variables that may bind to $true$ or $false$.

Whereas the domain of each scenario $\phi_{f_i} \in \Phi_f$ is the whole type $M \times T^*_{fin}$, the domain of the guards $g_{f_i} \in G_f$ may be a (shorter) projection of this type. Guards are not obliged to test all variables but may choose to test a subset of *relevant variables*. In the sequel, we group atomic predicates by the *relevant variables* that they constrain.

4.4.2. Treating predicate partitions as value-spaces. Consider that a given atomic predicate $p(v, c) \in Atomic$ constrains the values of a variable $v : T$ in relation to some constant $c : T$. This expression may be construed as denoting a partition of the type T; that is, the predicate $p(v, c)$ stands for the set of values $v : T$ that satisfy p. By symbolic manipulation of $Atomic$ predicates, it is possible to construct further partitions of T, such that we obtain a set of predicate terms that completely partition T. Whereas predicates from $Membership$ generate two partitions, for example, $\{(e \in s), (e \notin s)\}$, predicates from $Comparison$ generate three partitions on total orders, for example, $\{(v < c), (v = c), (v > c)\}$. The construction of partitions is achieved by negating and splitting atomic predicates. Negation has the usual logical sense. The definition of $split()$ is sensitive to certain predicates $cmp(x, y) \in Comparison$ with a compound sense:

$$\begin{aligned}
&\text{split : Atomic} \rightarrow \text{Set[Atomic]} \\
&\quad \text{split (x <= y)} = \{\text{(x < y), (x = y)}\} \\
&\quad \text{split (x >= y)} = \{\text{(x > y), (x = y)}\} \\
&\quad \text{split (x != y)} = \{\text{(x < y), (x > y)}\} \\
&\quad \text{split (p)} = \{\text{p}\}.
\end{aligned}$$

The $partitions()$ function, constructed earlier, is sensitive to whether its input is an $Atomic$ predicate or an $Atom$. It maps an $Atomic$ predicate to a set of complementary predicates on the same relevant variables and maps an $Atom$ to a pair of Boolean values:

$$\begin{aligned}&\text{partitions : Atomic} \cup \text{Atom} \rightarrow \text{Set[Atomic} \cup \text{Atom]}\\ &\quad\text{partitions(p} \in \text{Atomic)} = \text{split(p)} \cup \text{split(negate(p))}\\ &\quad\text{partitions(v} \in \text{Atom)} = \{(\text{v} \leftarrow \text{true}), (\text{v} \leftarrow \text{false})\}.\end{aligned}$$

A single guard g may now be mapped to a set of sets, where each contained set denotes the partitions of one of its relevant variables (constraining it in relation to another variable or constant), defined by comprehension as

$$\begin{aligned}&\text{partition_sets : Predicate} \rightarrow \text{Set[Set[Atomic} \cup \text{Atom]]}\\ &\quad\text{partition_sets(g: Predicate)} = \{\text{partitions(p)} \mid \text{p} \in \text{atomic(g)}\}.\end{aligned}$$

Generalizing this to all guards $g_{f_i} \in G_f$ collected from all scenarios of operation f, we note that the guards for different scenarios may choose to test more or fewer relevant variables (being more, or less, selective). Therefore, the set of all partitions for an operation is the distributed union:

$$\begin{aligned}&\text{all_partitions : F} \rightarrow \text{Set[Set[Atomic} \cup \text{Atom]]}\\ &\quad\text{all_partitions(f} \in \text{F)} = \bigcup\{\text{partition_sets}(\text{g}_{\text{f}_\text{i}}) \mid \text{g}_{\text{f}_\text{i}} \in \text{G}_\text{f}\}, \text{ where}\\ &\qquad\quad \text{G}_\text{f} = \{\text{guard}(\phi_{\text{f}_\text{i}}) \mid \phi_{\text{f}_\text{i}} \in \Phi_\text{f}\}.\end{aligned}$$

This set returned by $all_partitions()$ contains, for each *relevant variable* tested in any of the guards $g_{f_i} \in G_f$, a set of symbolic partitions of that variable, used in the sequel to generate all the unique partitions of memory and state that could be presented to an operation.

4.4.3. Exhaustive partitions of relevant inputs and memory. The exhaustive partitions of relevant input and memory for a given operation $f \in F$ are constructed as the Cartesian product of the partition sets indexed by each relevant variable. The product of two partition sets is interpreted as a set of conjoined predicates. An example of this is

$$\begin{aligned}&\{(\text{e} \in \text{s}), (\text{e} \notin \text{s})\} \otimes \{(\text{v} < \text{c}), (\text{v} = \text{c}), (\text{v} > \text{c})\} =\\ &\quad\{(\text{e} \in \text{s}) \wedge (\text{v} < \text{c}), (\text{e} \in \text{s}) \wedge (\text{v} = \text{c}), (\text{e} \in \text{s}) \wedge (\text{v} > \text{c}),\\ &\quad\;(\text{e} \notin \text{s}) \wedge (\text{v} < \text{c}), (\text{e} \notin \text{s}) \wedge (\text{v} = \text{c}), (\text{e} \notin \text{s}) \wedge (\text{v} > \text{c})\},\end{aligned}$$

namely, ordered pairs created by the product are treated as conjunctions of atomic predicates (and atoms). We use the distributed version of the Cartesian product operator, because the set of partition sets may contain as many sets as there are relevant variables tested in guards:

$$\begin{aligned}&\prod : \text{Set[Set[Atomic} \cup \text{Atom]]} \rightarrow \text{Set[Proposition]}\\ &\qquad\text{X}_1 \otimes \text{X}_2 \cdots \otimes \text{X}_\text{n} = \{\text{x}_1 \wedge \text{x}_2 \cdots \wedge \text{x}_\text{n} \mid \text{x}_1 \in \text{X}_1, \text{x}_2 \in \text{X}_2, \ldots \text{x}_\text{n} \in \text{X}_\text{n}\}.\end{aligned}$$

This product represents every way in which one symbolic partition could be combined with every other symbolic partition of every relevant variable. This set represents all combinations of distinct bindings of relevant input and state variables that could be presented to an operation.

4.4.4. Preserving consistent partitions of inputs and memory. In fact, the set $\prod(all_partitions(f : F))$ is a conservative overestimate of the exhaustive partitions of relevant inputs and memory. This is because some conjunction terms, which apply constraints transitively to overlapping sets of variables, may be logically inconsistent. Consider how: $(x < y) \wedge (y = z) \wedge (z < x)$ is inconsistent, because no value may be found for y that is simultaneously greater and less than x.

The elimination of inconsistent conjunction terms is a constraint satisfaction problem. The approach taken is to populate the free variables of such terms with values that try to satisfy each predicate individually and then to evaluate the conjunction term as a whole. Conjunctions that do

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

DOI: 10.1002/stvr

not hold under any possible value assignments are deemed inconsistent. Existing constraint solvers for Java, such as Choco[†††] or Cream[‡‡‡], were explored but were found unsuitable for our needs, because they could only handle simple numeric and string types (whereas our specifications also have constructed *Set, List* and *Map* types). We therefore created our own constraint solver.

We assume the existence of a reliable binding operator $bind()$ that binds the free variables in a term with values that marginally satisfy the term's constraint (a tactic known as "crawling around the edges" of a specification; this approach has been used with success in the solvers used to find counter-examples in the Alloy tool [44]). The $bind()$ operator has no effect on bound variables and may choose to bind a left-hand or right-hand free variable in relation to a constant or bound variable mentioned in the same constraint. Some examples of this are

$$\text{bind (v: Integer} = 6) \Rightarrow \text{v} \leftarrow 6 \qquad \text{bind(v: Integer} > 6) \Rightarrow \text{v} \leftarrow 7$$
$$\text{bind}(6 > (\text{v: Integer} + 2)) \Rightarrow \text{v} \leftarrow 3 \qquad \text{bind}(7.4 > \text{v: Double}) \Rightarrow \text{v} \leftarrow 7.399$$
$$\text{bind(c: T} \in \text{s: Set[T]}) \Rightarrow \text{s} \leftarrow \{c\} \qquad \text{bind(c: T} \notin \text{s: Set[T]}) \Rightarrow \text{s} \leftarrow \{\}$$
$$\text{bind(isEmpty(s: Set[T]))} \Rightarrow \text{s} \leftarrow \{\} \qquad \text{bind(notEmpty(s: Set[T]))} \Rightarrow \exists \text{c: T} \cdot \text{s} \leftarrow \{\text{c}\}.$$

The $bind()$ operator is a meta-method defined for every operation in the expression language and for every type in *T*. For floating point inequalities, we assume that a small delta difference from a limit is adequate. For constructed $List$, Set and Map constraints, we assume that it is sufficient to satisfy the top-level constraint. For expressions in which the free variables are nested, $bind()$ propagates a derived constraint back into the expression, by forcing it to have a given result.

Given this binding operator, it is possible to populate all the free variables in conjunction terms, in some order. However, consider how the order of variable binding is significant. If $bind()$ were called to populate the term: $(x > 4) \wedge (x < z) \wedge (z > 7)$ in left-to-right order, then the variables will receive the following assignments: $\{x \leftarrow 5, z \leftarrow 6\}$ and the conjunction will evaluate to false. However, if the terms are populated in right-to-left order, $bind()$ will assign $\{z \leftarrow 8, x \leftarrow 7\}$ and the conjunction will evaluate to true. The objective is to find at least one consistent binding, which demonstrates that the conjunction term models a valid input and memory combination.

Treating a conjunction $conj$ as a sequence (cf. the tuple generated by the Cartesian product), we compute all permutations of this sequence $perm \in P(conj)$ and bind each permutation in indexed order of its terms, such that if at least one of these evaluates to true, the conjunction is valid. We conjecture that if a valid binding exists, then $bind()$ will always find it by "crawling around the edges." A sketch of the proof is by considering a conjunction of terms for which there is a unique solution: $(x > 5) \wedge (x < 7)$, for which $bind()$ will assign $\{x \leftarrow 6\}$, whereas any alternative binding algorithm that assigns to x values much greater than 5 or much less than 7 will fail to find this.

The consistent partitions of relevant input and memory are then determined as

$$\text{consistent_partitions}_\text{f} : \text{Set[Proposition]} = \{\text{conj} \in \prod(\text{all_partitions(f : F)}) \mid \exists\text{perm} \in \text{P(conj)} \cdot \text{bind(perm)} = \text{true}\},$$

where $P()$ computes all permutations of a conjunction term and $bind()$ binds all free variables of the permutation in order.

4.4.5. Determinism and completeness of operations. Having filtered the exhaustive partitions of relevant input and memory to preserve only those that are logically consistent, we may use these as symbolic exemplars of all relevant tuples that may be presented to the guards of an operation. For a given operation $f \in F$ and for the set of guards G_f of this operation, we determine the following:

$$\forall \text{c} \in \text{consistent_partitions}_\text{f} \cdot \exists! \text{g} \in \text{G}_\text{f} \cdot \text{g} \supseteq \text{c},$$

where $g \supseteq c$ denotes logical subsumption, in the sense that g is more general than c, such that if c holds for some input and memory tuple, g will also hold. This formulation in terms of logical subsumption is equivalent to the original proof obligation:

$$\forall \text{in}_\text{f} : \text{T}^*_\text{fin}, \forall \text{m} : \text{M} \cdot \exists! \text{g}_{\text{f}_\text{i}} \in \text{G}_\text{f} \cdot \text{g}_{\text{f}_\text{i}}(\text{m}, \text{in}_\text{f}) = \text{true},$$

[†††]Choco solver: an open-source Java library for constraint programming. http://www.choco-solver.org/
[‡‡‡]Cream: class library for constraint programming in Java. http://bach.istc.kobe-u.ac.jp/cream/

since the symbolic conjunction tuple c is a projection of $in_f \times m$ that is relevant to the guards G_f of operation f, which is agnostic about other values in $T^* \times M$. The subsumption check is equivalent to proving that the guard accepts all inputs and memory states that satisfy the conjunction tuple.

Symbolic subsumption involves matching terms indexed by common operands and proving that the guard-term is more general than or equal to the input-term; namely, the guard-term: $(x \leq y) \wedge (p \neq q)$ subsumes the input-term: $(x < y) \wedge (p > q) \wedge (e \in s)$, because the first two sub-terms of the input are pairwise more specific and the third term is irrelevant. This check is performed directly on model predicates, using subsumption-checking meta-methods, without any need to populate terms with values.

4.5. *Limitations of the approach*

The decidability of verification could be challenged by unbounded values. We mitigate this by ensuring that all types are finite and structures are of bounded length (all types are computational, so must fit within finite computer memory). While the symbolic subsumption algorithm is complete by design with respect to all predicates in the expression language, the constraint solver, whose purpose is to eliminate inconsistency, may possibly fail to find consistent bindings in deeply nested expressions that break the assumptions of the reliable binding operator. The effect of this would be to exclude a valid input term from the subsumption checker, which would proceed without it. Degrading gracefully, the verification tool will still report a result on the basis of all other permutations. The tool works well in practice, as demonstrated by the examples included in Section 6.

5. VERIFICATION AND TESTING TOOLS

The BrokerCloud Verification and Testing Tool Suite (VTTS) [25] includes two tools to assist the designer in writing a correct specification, suitable for model-based test generation. The designer develops a DSXM specification Z in the manner outlined in Section 2. The *validation tool* may then be used to check the desired state-transition behaviour of the associated automaton A_Z and the *verification tool* may be used to check the completeness and determinism of the associated protocol P_Z, using the approach described in Section 4. VTTS also includes two tools to assist the designer in generating tests. A correct specification may be submitted to the *test generation tool* to create complete functional test suites, using the optimizing approach described in Section 3. These technology-neutral test suites contain full information for code generation in any desired format. As a proof of concept, the *test grounding tool* can map these to executable JUnit tests for a variety of Java-wrapped web services.

The *validation* and *verification tools* accept as input an XML specification and produce as output an annotated version of the relevant part of the specification, flagging various issues by attaching extra XML elements of the kinds: *Notice*, *Analysis* or *Warning* describing the results of the analysis and highlighting any faults that need correction. The *test generation tool* also accepts an XML specification but outputs an abstract XML test suite, similarly annotated to describe test parameters, test optimizations and test coverage properties. The *test generation tool* accepts an abstract XML test suite as input and generates Java code for JUnit, in which all metadata is translated into Java comments.

5.1. *Validation tool*

The *validation tool* helps the designer to structure the state space of the service, by reflecting back the consequences of the state machine design. The tool analyses the completeness of the states, under all known events supported by the machine (the FSM's alphabet, the union of events on all transitions) by static analysis, and determines the reachability of states by dynamic simulation. Altogether, it checks that an initial state has been specified (well-formedness); it checks by exploration that every state is reachable from the initial state (reachability); for each state specified in the machine, it checks whether a transition exists for every event in the alphabet (completeness); and for each scenario specified in the protocol, it cross-checks whether this has a corresponding transition in the machine (consistency).

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

DOI: 10.1002/stvr

Feedback is presented to the designer by annotating the model of the *Machine*. The output XML file may be processed by client tools, to present this information to the designer in any desired format. Warnings are issued if any of the earlier properties are violated. The designer may then repair a faulty machine or decide whether any missing transition should be added. After validation, the machine is considered complete, and missing transitions are treated implicitly as trivial transitions of the form: *request/ignore*, where *ignore* indicates a mandatory null response.

5.1.1. Validation algorithm. The validation algorithm is straightforward, based on a static analysis of the specification model. The states of the machine and explicit transitions leaving each state are found by inspection in the specification model. In addition,

- exactly one state must be marked as the initial state;
- the alphabet of the machine is constructed from the set of all transition events used anywhere in the machine;
- missing transitions for each state are discovered by comparing that state's explicit transitions against the alphabet;
- bounded breadth-first search (with a 5 s time-out) explores the machine to determine whether all states are reachable; and
- consistency of names used for scenario labels and transition events is determined by comparing the two alphabet sets.

5.2. Verification tool

The *verification tool* helps the designer to complete the branching logic expressed in the protocol, by checking this for completeness and consistency. The tool ensures, by static analysis, that all specified parameters are bound to suitable values and then checks that every specified operation is deterministic and non-blocking, by symbolic execution and constraint satisfaction. Altogether, it checks that every specified variable in memory is assigned an initial value (initialization); for each scenario of each operation, it checks that every specified input has a test input binding (testability); for each operation, it determines the exhaustive equivalence partitions of inputs and memory and then checks that every equivalence partition triggers exactly one scenario of that operation (deterministic, non-blocking); and finally, for each transition specified in the machine, it cross-checks whether this has a corresponding scenario in the protocol (consistency).

The initialization and testability checks ensure that the protocol can be simulated, without raising null value exceptions. The checks for determinism and non-blocking ensure that operations are complete and behave consistently under all inputs and memory. These two conditions are important, not only for logically correctness but also for testability [15]. If these conditions were not met in the specification, testing could not distinguish between different nondeterministic behaviours and might not detect blocking in the implementation. Feedback is provided by annotating the model of the *Protocol*, which may be visualized and presented to the designer in any desired format.

5.2.1. Verification algorithm. The verification algorithm was presented in Section 4, and the stages of the algorithm are summarized hereafter. Referenced model elements, such as memory, operations and variables, are all available by inspection in the model. In addition,

- for each memory variable, a corresponding assignment is sought in the initial memory bindings; and for each input variable of each scenario, a corresponding test input binding is sought in the model;
- for each operation, all possible equivalence partitions of symbolic input and memory are calculated, by analysing the set of predicates collected from all of the scenarios of that operation;
- a constraint solver is used to eliminate any inconsistent partitions denoting impossible input/memory configurations; and

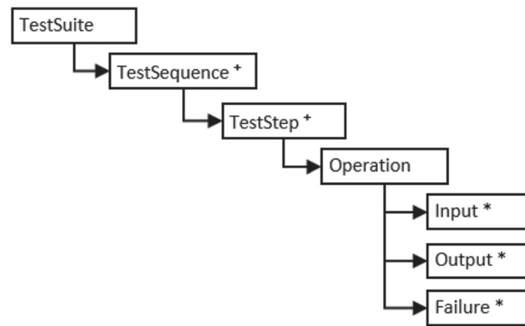


Figure 3. Compositional structure of a test suite.

- every consistent partition of input and memory is presented in turn to each of the guarded scenarios of that operation, to determine by symbolic subsumption whether that partition is accepted by exactly one guarded scenario.

5.3. *Test generation tool*

The *test generation tool* accepts a verified (complete, deterministic and non-blocking) specification and from this generates complete functional tests for the service under test. The specification serves both as a means to determine the minimum level of test coverage and also as the oracle for the generated tests, by supplying all transition labels, reached states and expected input–output correspondences for each test sequence. The test suite is output in a technology-agnostic, platform-independent XML format (Figure 3), which is later used as the input to different translators that ground the tests for particular service platforms, implementation styles and programming languages.

Altogether, the *test generation tool* checks by exploration that every state is reachable from the initial state, despite the blocking effects of guards (state coverage); it checks during simulation that every transition in the specification has been triggered at least once, despite the blocking effects of guards (transition coverage); it proposes higher coverage paths, such as all-pairs and all-triples of transitions, to cover a non-minimal implementation (super-minimal coverage); it filters proposed test paths through the memory and guarded operations, to eliminate paths that are infeasible in the specification (path feasibility optimization); it prunes all test paths that have proven empty cycles in their prefix, achieving a significant reduction in the test suite size (test redundancy optimization); it optionally creates multi-objective tests, merging shorter and longer test paths whose test goals can be jointly verified (test compression optimization); generated test sequences contain complete information about required test inputs, expected test outputs or failures, triggered transition labels and reached states (test observation completeness).

The output of the *test generation tool* is an XML file, whose schema is visualized as a tree-diagram in Figure 3. The tool gives feedback to the tester, by annotating this model with metadata, recording the chosen test parameters and the reductions to the test suite size achieved by successive optimizations. The tool also warns if states or transitions were not covered when simulating the Stream X-Machine, which might occur under the blocking effects of guards. The designer can then take different kinds of remedial action in response, such as increasing the maximum path length, or specifying different test values, or creating additional scenarios to ensure certain critical memory states are reached.

5.3.1. Test generation algorithm. The optimized DSXM test generation algorithm was presented in Section 3, and the stages of the algorithm are summarized hereafter. We simulate the complete Stream X-Machine, that is, the *Machine* and *Protocol* simultaneously, keeping track both of the current state and current memory values, in order to generate feasible test sequences by construction: this is a new achievement in Stream X-Machine testing. Altogether,

- bounded breadth-first search (with a 5 s time-out) explores the Stream X-Machine to determine the minimal feasible state cover, under the possible blocking effects of guards;

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

DOI: 10.1002/stvr

- a test suite is constructed from the concatenated product of the minimal feasible state cover and the bounded Kleene star language (which is *prefix closed* and *path complete*), consisting of all possible event sequences of lengths $0..n$, where n is the maximum path length; the size of this test suite is the baseline against which optimization is measured;
- candidate test sequences from the baseline test suite are presented in turn to the Stream X-Machine for simulation, after resetting the *Machine* and *Protocol* to their intial states, such that for each sequence
  - any sequence that executes completely in both the *Machine* and the *Protocol* is retained in the test suite;
  - any sequence that executes in the *Machine* but is blocked in the *Protocol* is discarded as infeasible;
  - any sequence that is blocked in the *Machine* is only retained if it ends with the empty cycle, otherwise it is discarded as redundant; and
  - assertion flags are added to the final step of each sequence to check the transition label, reached state and expected outputs; and
- if the tester requests multi-objective testing, the optimized test suite is compressed further, by merging shorter sequences with longer sequences of which they are a prefix.

5.3.2. Test optimization benefits. The *test generation tool* uses verified assumptions about trivial transitions and mutually excluding scenarios to make significant optimizations on the size of the test suite generated by the baseline DSXM test generation algorithm. The three main optimization steps were presented in Section 3.5 but are summarized hereafter, with an indication of each step's effectiveness (see also Section 6).

The precursor step is to use a reliable state oracle instead of extended characterization sequences from W. This directly reduces the size of the test set by a factor of $card(W)$, because in the classic W-method, every exploratory sequence would be extended by each of the sequences in W, to identify the reached state. Test sequences are also shorter without the suffix from W. Whereas executing sequences from W would force the tested system through additional states and transitions, the state oracle function is side-effect free, which we exploit hereafter in the third optimization.

The first optimization filters out all paths that contain trivial prefix cycles. If a path ends with a trivial cycle, this is confirmed in the implementation by testing for an explicit *nullop*; thereafter, all future paths that extend it are redundant, because any such path is equivalent to a shorter path without the trivial cycle. The benefit gained by removing redundant paths grows in proportion to the elaboration of states with missing transitions in the specification but typically reduces the size of the test suite by 60–70% in a system with three to five states. This huge gain is achieved relative to the exponential growth in the size of the test suite as the maximum path length increases.

The second optimization filters out infeasible paths that cannot be executed. Previous SXM testing approaches [15–17] recognize that simulating (only) the SXM's associated automaton may generate infeasible paths, blocked by guards. The hypothetical test function converts abstract paths into sequences of concrete test inputs, such that if no input can be found to trigger a given transition, the input sequence is truncated. In our approach, the maximally feasible prefix is already in the *prefix closed* test suite, so we may safely discard the infeasible sequence. Negative testing (to prove blocking) is accomplished by positive testing for a mutually exclusive path. The benefit gained by removing infeasible paths grows in proportion to the number of guards used in the specification but typically reduces the size of the test suite by 20–30%. Furthermore, we guarantee detection of missing coverage due to blocking, because we simulate the whole DSXM. This cannot be detected by simulating the automaton alone (unless the system is artificially forced to be *input complete*).

The third optimization compresses the test suite by merging sequences that share test objectives. This can only be achieved if all test-final assertion checks are side-effect free. We may then embed shorter sequences inside longer multi-objective sequences that contain them as a prefix, executing assertion checks after one or more prefix steps, as well as after the final step. We check the triggered transition (cf. *output distinguishability*), the reached state (cf. sequences from W) and the result

of each operation. The benefit of compressing the test suite grows in proportion to the number of sequences that are also prefixes but typically reduces the size of the test set by 5–15%.

Altogether, the combination of these different optimization strategies may reduce the size of the generated test suite to under 10% of the theoretical size of test suites generated by the W-method [19]. Examples of performance are reported for five services in Section 6 hereafter.

5.4. *Test grounding tool*

In general, the translation of test suites into executable tests is the responsibility of the service provider, who controls the platform and programming languages used to build the service. The *test grounding tool* is provided as a proof of concept that the generated tests can be translated into different service technologies. The tool performs a model-to-code transformation, accepting an XML model produced by the *test generation tool,* and from this generates JUnit tests suitable for a JAX-WS Java client for a WSDL/SOAP web service or a JAX-RS Java client for a REST/JSON web service. A grounding to plain Java is also provided. Like other code generators, it follows a number of standard design patterns [45], in particular the *Visitor* pattern to walk through the model test suite (Figure 3) and the *AbstractFactory* pattern to synthesize Java types and values from models expressed in the standard expression language (see also Section 2).

Standard groundings all assume that the service client has a Java API and that the service may be tested through this API. Test grounding generates the source code for a JUnit test-driver class, whose @*Test*-annotated methods are the individual test sequences. The @*Before*-annotated method initializes the service on first invocation and resets it subsequently. Each test consists of prefix steps designed to reach a given memory state, followed by the final verified step. Every aspect may be checked by JUnit assertions, to verify which scenario was executed, what result was returned and what state was reached. In multi-objective testing, intermediate steps may also be verified by assertion.

The different concrete groundings adopt slightly different strategies (through different realizations of the *Grounding* visitor). The *JaxWsGrounding* generates a JAX-WS Java client, which converts regular Java method-calls into SOAP requests and then converts the SOAP responses back into Java objects. The *JaxRsGrounding* generates a JAX-RS Java client, which dispatches RESTful-style HTTP requests that include the operation-name and any inputs as part of the URL and uses a JSON parser to convert the responses back into Java objects. Both of these groundings use JUnit to execute the concrete tests.

However, this is not the only possibility. A custom grounding was created that translates operation inputs and outputs directly into SOAP requests and responses [28], in the format accepted by the SOAP-UI test engine. This invokes a sequence of SOAP requests on a web service and compares the actual SOAP responses against expected responses. This approach can be used to test any service with a WSDL interface and can test full state and transition behaviour if the service exposes these in the manner described in the design-for-test conditions (Section 1).

Our partners at SAP have also created a custom grounding that outputs tabular HTML instructions for the Selenium test engine [27], which drives the tested service through a rich-client, manipulating and comparing the HTML Document Object Model (DOM) on the client-side. This is useful for styles of service with mixtures of client-side and server-side business logic (typically implemented using JavaScript), or for clients which communicate via AJAX with the server, rather than through a functional web service API. However, this kind of grounding can only be created by the service-provider with full knowledge of the client-side DOM.

6. CASE STUDY AND EVALUATION OF RESULTS

During the course of the BrokerCloud project [4], seven case studies were developed, ranging from simple micro-services to complete service applications, including a shopping cart, a data warehouse and a VAT clearance application. These were deliberately chosen to vary the numbers of states, guarded or trivial transitions and dependency on memory. The specifications are available online, accessible via public URL [25] and so may be used as input to the validation, verification, test

generation and test grounding tools. In the succeeding texts, we describe the evolution of one of these examples, the *DocumentStore* data warehouse, through the various stages of specification, validation, verification and testing. The aim is to give a substantial example of service specification, showing the kinds of issue encountered at different stages during its creation. We follow this with a summary of our experience in applying the proposed approach to the seven case studies.

6.1. *Requirements for the DocumentStore*

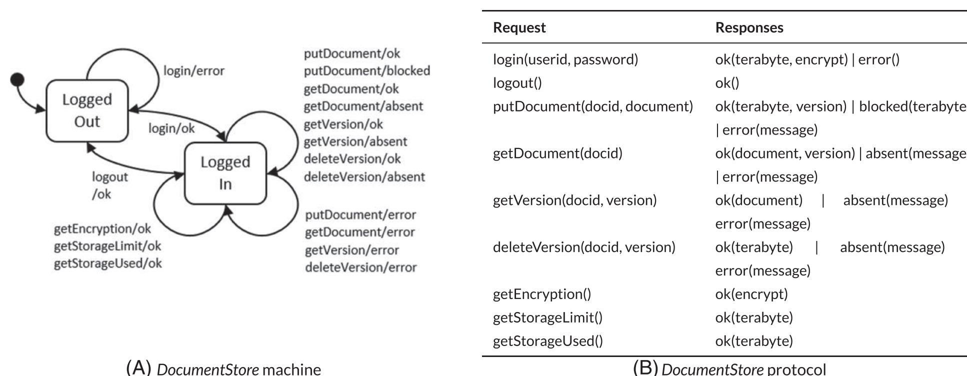
Imagine that a platform provider wishes to offer bulk data storage services at different levels of assured quality. Non-functional requirements include availability, reputation and cost, assured by other components of the BrokerCloud framework [6]. Functional requirements include offering three different storage capacities (up to 10, 100 or 1000 terabytes), three possible AES encryption key lengths (128, 192 or 256 bits), full document versioning and login-based authentication.

Figure 4(a) shows the state machine for the *DocumentStore*, as the designer first imagined it (note that this is not the final version). The service has two states: *LoggedOut* and *LoggedIn*. In the *LoggedOut* state, the only action possible is to login. In the *LoggedIn* state, all actions are possible apart from login. Signed-up users will already have been given accounts corresponding to the levels of service agreed with the provider; this includes the encryption level, which is constant for each user, and reported only at login. Users must first login with their *userid* and *password*. Subsequently, they may check in, or check out, versioned documents, up to an agreed storage allocation (the whole *document* is stored each time rather than delta increments.) The system must ensure that users cannot exceed their agreed allocation. Figure 4(b) sketches the protocol to support the operations shown in Figure 4(a). Requests are shown with their alternative responses, and parameters have the following meanings: *terabyte* is the allocated storage, *version* is the version number, *encrypt* is the encryption level and *message* is an error message.

6.2. *Specification for the DocumentStore*

Figure 5 shows the memory abstraction for the *DocumentStore*. Constants represent valid and invalid users and passwords, an encryption and storage limit associated with the valid user, and documents of different sizes. Variables represent the total storage used, a document ID counter, a version-list of documents and a repository of document versions. These are suitably initialized. The versioning system is modelled as a map, which stores the version-list of a document against its *docid*. Documents are represented as uniquely identifiable instances of an otherwise uninterpreted type, *Document*, whose sizes in terabytes are recorded against their identifiers in a map. One of these deliberately exceeds the storage limit.

The operations require at least one scenario to cover each expected branch in the protocol. Figure 6 shows an excerpt of this protocol, describing the *putDocument* operation. Initially, the designer expected this to have three responses: {*ok, blocked, error*} (Figure 4(a)). However, while developing the *ok* response, the designer realized that different effects should happen when adding



Request	Responses
login(userid, password)	ok(terabyte, encrypt) | error()
logout()	ok()
putDocument(docid, document)	ok(terabyte, version) | blocked(terabyte) | error(message)
getDocument(docid)	ok(document, version) | absent(message) | error(message)
getVersion(docid, version)	ok(document) | absent(message) | error(message)
deleteVersion(docid, version)	ok(terabyte) | absent(message) | error(message)
getEncryption()	ok(encrypt)
getStorageLimit()	ok(terabyte)
getStorageUsed()	ok(terabyte)

(A) *DocumentStore* machine (B) *DocumentStore* protocol

Figure 4. Initial conception of the *DocumentStore* machine and protocol.

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

a new document or a subsequent version of this document. Figure 6 shows how the old *ok* response has been refined into distinct *new* and *update* responses, which behave differently in the way that the *docVersions* variable is handled. We assume that the designer forgets to revisit the state machine to adjust the transition labels. Unknown to the designer at this stage, there are two further errors buried in Figure 6.

```
const encryption : Integer = 192
const storageLimit : Integer = 100
const knownUser : String = “Jane Good”
const knownPassword : String = “serendipity”
const badUser : String = “Jim Hacker”
const badPassword : String = “backdoor”
const badDocid : String = "bad document identifier"
const badVerid : String = "bad version identifier"
const smallDoc : Document = Document(d1)
const largeDoc : Document = Document(d2)
const docSizes : Map[Document, Integer] = {d1=20, d2=100}
const emptyVersions : List[Document] = []
const emptyRepository : Map[Integer,List[Document]] = {}

var storageUsed : Integer := 0
var docCounter : Integer := 0
var docVersions : List[Document] := emptyVersions
var docRepository : Map[Integer,List[Document]] := emptyRepository
```

Figure 5. Memory abstraction for the DocumentStore.

```
operation putDocument (docid: Integer, document: Document) ->
  (terabyte : Integer, version : Integer) | (terabyte: Integer) | (error: String)
scenario putDocument/new test docid = docCounter + 1, document = smallDoc
 if docid = docCounter + 1 and storageUsed < storageLimit – searchAt(docsizes, document)
 then
  storageUsed := storageUsed + searchAt(docsizes, document),
  docCounter := docid,
  docVersions := insert(emptyVersions, document),
  docRepository := replaceAt(docRepository, docid, docVersions),
  terabyte := storageLimit – storageUsed,
  version := size(docVersions)
scenario putDocument/update test docid = docCounter, document = smallDoc
 if docid  > 0 and docid <= docCounter + 1 and
  storageUsed < storageLimit – searchAt(docsizes, document)
 then
  storageUsed := storageUsed + searchAt(docsizes, document),
  docVersions := insert(searchAt(docRepository, docid), document),
  docRepository := replaceAt(docRepository, docid, docVersions),
  terabyte := storageLimit – storageUsed,
  version := size(docVersions)
scenario putDocument/blocked test docid = docCounter, document = largeDoc
 if docid  > 0 and docid <= docCounter and
  storageUsed >= storageLimit – searchAt(docsizes, document)
 then
  terabyte := storageLimit – storageUsed
scenario putDocument/error test docid = 0, document = largeDoc
 if docid  <= 0 or docid >= docCounter + 1 then
  error := badDocid
end
```

Figure 6. Protocol of the *DocumentStore::putDocument* operation.

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

DOI: 10.1002/stvr

Symbolic inputs revealing nondeterminism (a):

docid = docCounter + 1 **and** storageUsed < storageLimit – searchAt(docSizes, document)
and docid = 0 **and** docid > docCounter

docid = docCounter + 1 **and** storageUsed < storageLimit – searchAt(docSizes, document)
and docid < 0 **and** docid > docCounter

Symbolic inputs revealing blocking (b):

docid > 0 **and** docid = docCounter + 1 **and** storageUsed = storageLimit – searchAt(docSizes, document)
and docid > docCounter

docid > 0 **and** docid = docCounter + 1 **and** storageUsed > storageLimit – searchAt(docSizes, document)
and docid > docCounter

Figure 7. Symbolic inputs revealing blocking and nondeterminism.

6.3. *Validation and verification of the DocumentStore*

Assuming the designer did not repair inconsistencies in the specification, then validation follows next. The *validation tool* exercises due diligence, reporting that the *LoggedIn* state has no transitions for *login/ok* and *login/error* and that these are the only transitions available in the *LoggedOut* state, whereas all other transitions are missing. This is as intended. However, the tool also reports warnings that two transitions: *putDocument/new* and *putDocument/update* were present in the protocol but were absent from the machine. The designer fixes the state machine specification by replacing the old *putDocument/ok* transition (in Figure 4(a)) with these refinements (from Figure 6).

Next, the specification is passed through the *verification tool*, which agrees that all four variables in memory are correctly initialized (Figure 5), but then discovers faults in the specification of the operation *putDocument*. This is found initially to be nondeterministic, due to the fact that a pair of symbolic inputs that were generated by the tool and which are shown in Figure 7(a), both trigger multiple scenarios, namely, *putDocument/new* and *putDocument/error*. The fault is found in the guard *for putDocument/new* (Figure 6) which should have the additional conjunct: $docid > 0$ to distinguish its acceptance of positive *docids* from the guard of *putDocument/error*, which accepts *docids* less than or equal to zero. Formerly, they overlapped on the value zero. After fixing this fault, the designer passes the specification through the tool again, which now reveals that *putDocument* is blocking, due to the fact that another pair of generated symbolic inputs shown in Figure 7(b) are not accepted by any scenario. This fault is traced to an off-by-one boundary error in the guard condition for the scenario *putDocument/blocked*. The upper bound for *docid* should be raised by one: $docid \leq docCounter + 1$. The designer fixes this fault. The final version of this specification is available to view online [25].

As an indication of how useful automatic verification can be, the operation *putDocument* is found to have 27 equivalence partitions, of which one triggers the scenario *putDocument/new*, one triggers *putDocument/update*, four trigger *putDocument/blocked* and the rest trigger *putDocument/error*. However, the operation *getVersion* is found to have 81 equivalence partitions, of which four trigger *getVersion/ok*, fourteen trigger *getVersion/absent* and the rest trigger *getVersion/error*. This would be too many to check by hand; or rather, the designer might never think of all such cases.

6.4. *Test generation for the DocumentStore*

Tests are initially generated for the *DocumentStore* specification with the path length set to 1. While this is sufficient to cover all states and transitions of the associated state machine, it is not adequate to cover all the transitions of the Stream X-Machine, due to the restrictions imposed by guards. The tool reports generating 39 unique sequences of events, of which 23 paths are executable and 16 paths are infeasible. The infeasible paths are sequences that attempt to update or access a document before any document has been deposited. The tool issues warnings that eight transitions are never executed: *getDocument/ok, getDocument/absent*, *putDocument/update, putDocument/blocked , getVersion/ok, getVersion/absent, deleteVersion/ok* and *deleteVersion/absent*, none of which are enabled until at least one document is present.

Because coverage of the specification is incomplete, tests are generated again with the maximum path length set to 2. The resulting test suite includes the previous test suite, plus longer paths. This time, the only generated warning is for one uncovered transition: *getDocument/absent* (only testable after a document inserted at a given *docID* has been deleted); all other transitions are exercised at least once. The tool reports 742 unique theoretical sequences, of which 394 paths are deemed infeasible (blocked by guards), and a further 209 paths are deemed redundant (with trivial prefix cycles). Only 139 paths are retained as usefully discriminating, executable tests. This corresponds to 18.7% of the original test suite.

To ensure that every transition of the specification is covered at least once, tests are generated again with the maximum path length set to 3. This time, there are 14,099 unique theoretical sequences, because the test suite grows exponentially. Of these, 8,102 paths are deemed infeasible, 4,862 paths are deemed redundant, leaving 1,171 paths that are usefully discriminating, executable tests. This corresponds to 8.3% of the original test suite. If the tester selects multi-objective test generation, in order to merge paths which share test objectives, the executable test suite is optimized further to 1,077 paths, which is 7.6% of the original test suite.

The resulting test suite has powerful conformance-testing properties. It tests every scenario from the specification at least once, and for all but one scenario, *getDocument/absent* covers more than the minimal FSM of the specification. Testing with the maximum path length set to *n* covers all 1-step reachable duplicates of implementation states that were covered in the specification with path length set to *n-1*.

6.5. Evaluation across multiple services

We present summary results for all seven of the service case studies developed during the course of the BrokerCloud project [4], which may be examined online [25]. The *Login* service is a single-sign-in micro-service. The *Account* service is a payment micro-service. *ContactList* is an address book micro-service. *HolidayBooking* is a mobile app for booking periods of vacation leave, proposed by partner SAP SE, developed in SAP OpenUI5 and executing on the HANA platform [27]. We developed a *ShoppingCart* app for online shopping, accessed via a SOAP/WSDL API on an Apache Tomcat web service. *DocumentStore* is a data repository app, developed by partner SEERC and accessed via a REST/JSON API on a cloud Apache web service. *VatClearance* is a business app for self-employed clients to calculate their annual VAT returns, proposed by partner SingularLogic SA and accessed via a REST/JSON API on the SingularLogic Orbi cloud platform.

Table II shows summary verification characteristics for each of these service specifications. For each service, the columns indicate the number of operations in the service's interface; the total number of scenarios describing distinct paths through these operations; the total number of equivalence partitions generated from guard expressions to validate the scenarios; the number of valid, logically consistent partitions used in verification; the number of invalid, logically inconsistent partitions that were excluded; the maximum number of partitions found for any operation; and the maximum number of valid partitions found for any operation.

From this, it is clear that the *DocumentStore* example presented the most challenging task for verification, with two operations each having 81 equivalence partitions, leading to 220 equivalence partitions overall. The cases of 81 partitions arose from guards for these operations consisting of four conjoined terms, each of which had three partitions individually. The *ContactList* example was

Table II. Summary verification characteristics.

Service	#Operations	#Scenarios	#Partitions	#Valid	#Invalid	MaxPart	MaxValid
Login	3	4	11	11	0	9	9
Account	6	11	19	19	0	9	9
ContactList	4	11	54	28	26	27	12
HolidayBooking	6	13	39	32	7	18	18
ShoppingCart	9	13	16	16	0	3	3
DocumentStore	9	19	220	220	0	81	81
VatClearance	6	11	12	12	0	3	3

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

Table III. Summary test generation characteristics.

Service	#States	#Trans.	Cover path	#Total tests	#Infeas. tests	#Redun. tests	#Single tests	#MultiTests	%Prune
Login	2	8	1	9	0	0	9	7	22.2%
Account	2	22	2	254	63	99	92	82	67.7%
ContactList	3	33	3	4,126	3,501	0	625	520	87.4%
HolidayBooking	3	39	3	6,956	3,475	2,990	491	433	93.8%
ShoppingCart	4	52	2	703	108	377	218	199	71.7%
DocumentStore	2	38	3	14,099	8,102	4,826	1,171	1,077	92.4%
VatClearance	4	44	4	61,359	24,514	36,190	655	571	99.1%

also intriguing, because of its high number of detected invalid partitions. These arose from a higher occurrence of variable-sharing across the conjoined terms of a guard (e.g. conjuncts having three terms, pairwise sharing two variables). This service also had a higher ratio of scenarios per operation, to capture memory-contingent branching behaviour. In other examples, memory-contingency was less important, for example, the *ShoppingCart* is strongly state-contingent and its scenarios mostly have single-term guards resulting in fewer partitions (almost one per scenario).

Table III shows summary test generation characteristics for each of these service specifications. For each service, the columns indicate the number of states in the associated automaton; the total number of transitions (including trivial circular transitions); the transition path-length necessary to cover the Stream X-Machine (starting from each state); the baseline number of tests required to meet this test obligation; the number of culled tests deemed infeasible; the number of culled tests deemed redundant; the remaining number of useful single-objective tests; the optimized number of merged, multi-objective tests; and an efficiency expressed as the ratio of pruned (culled and merged) tests to baseline tests.

From this, it is clear that the specifications offered vastly different testing challenges. They ranged from *Login*, whose SXM was already covered by the transition cover of the associated automaton, to *VatClearance*, which needed to exercise longer paths up to length 4 (from each state) to reach memory conditions that would allow certain guards to be triggered. While the size of the baseline test suite increases exponentially in the length of the transition path, there is also greater opportunity to reduce the test suite size by optimization, where pruning ranged from 22.2% in *Login* to 99.1% in *VatClearance*. For median coverage paths of length 3, on average around 90% of the theoretically possible baseline tests may be eliminated, leaving behind the 10% most effective tests (effective in the sense: non-redundant, executable and discriminating).

Different specifications benefited more or less from different optimizations. *Login* showed how a path length greater than one is needed before the culling of redundant tests is possible but still benefited from path merging. Services with greater memory-contingency, having a higher branching factor (viz. more scenarios per operation), such as *ContactList, HolidayBooking* or *DocumentStore*, offered greater opportunity to cull infeasible tests. The extreme example was *ContactList*, whose states were entirely characterized by memory conditions and for which infeasible paths were pruned before empty cycles were considered. Services with greater state-contingency, having a larger state space, such as *ShoppingCart* or *VatClearance*, offered greater opportunity to cull redundant tests, due to the increased presence of trivial prefix cycles.

In terms of time complexity, the test generation algorithm is exponential in the maximum length of paths explored, the verification algorithm is proportional to the product of the number of scenarios and their input/memory partitions and other algorithms are linear in the size of their inputs. However, in timing experiments conducted on the examples, we found that execution times were dominated by other factors, such as data transmission via HTTP, when invoking the tools as cloud services via URL.

The biggest time penalties were incurred by client web browsers (or GUI tools) which rendered the XML or Java code using syntax highlighting. We established this by invoking the test generation service via a remote internet connection and rendering results on a laptop with an Intel Core i5 1.8GHz processor with 6GB RAM. We compared the timing for the *VatClearance* and *DocumentStore* examples. Approximately speaking, the former generated 61K tests, optimized to 0.5K

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

Table IV. Micro-benchmarking test generation performance.

Service	#States	#Trans.	Cover path	#Total tests	#MultiTests	t-Gen	t-Save	t-Ground	t-Total
Login	2	8	1	9	7	7 ms	4 ms	4 ms	15 ms
Account	2	22	2	254	82	15 ms	18 ms	20 ms	53 ms
ContactList	3	33	3	4126	520	68 ms	94 ms	71 ms	233 ms
HolidayBooking	3	39	3	6956	433	100 ms	85 ms	62 ms	247 ms
ShoppingCart	4	52	2	703	199	37 ms	42 ms	38 ms	117 ms
DocumentStore	2	38	3	14 099	1077	121 ms	183 ms	166 ms	470 ms
VatClearance	4	44	4	61 359	571	364 ms	87 ms	102 ms	553 ms

tests, while the latter generated only 14K tests, but optimized to 1K tests (twice as many). The former rendered the XML in 12 s, whereas the latter took 25 s, indicating that response times were determined by the size of the rendered XML file rather than by the complexity of test generation. In a second comparison, we remotely piped the results of test generation into test grounding to Java (an additional processing stage) but returned the results without any syntax highlighting. The response times were an order of magnitude faster at respectively 1.5 and 2 s, indicating that local rendering was the dominant cost.

Table IV gives a better indication of the raw performance of the (most costly) test generation algorithm. We conducted Java micro-benchmarking experiments on a laptop with an Intel Core i7 1.8GHz processor with 16GB RAM. The test program took time-checkpoints after test generation (with all optimizations), after saving the XML file and after generating the Java grounding for each of the examples. The average timings shown in Table IV were obtained over ten runs for each set of measurements, to ensure that the Java code had been properly loaded and exercised, but not to the point of invoking the Just-in-Time peephole optimizer. The various split timings are as expected for the complexity of each task.

6.6. *Threats to validity*

Results are reported for the seven specifications and the described partner platforms that were made available during the BrokerCloud project [4]. It was not reasonable to create a larger base of test examples in the time, due to the individual nature of partner services and the need to develop a specific specification for each. Otherwise, we identified the following threats to validity [46].

The threat to *internal validity* concerns the possibility that the BrokerCloud VTTS tools are faulty. To mitigate this, we used code inspections and modular testing during development and then exercised the tools on a wide variety of specifications having different extreme properties. Verification and testing results were compared with expected theoretical results. The source code has been publicly available, along with a user manual and example specifications, allowing others to reproduce our results.

The threat to *construct validity* concerns the possibility we did not measure properties that are of interest. To counter this, we included execution times and metrics describing the size of the DSXM (numbers of states, transitions, operations and guarded scenarios) relating these to metrics describing the cost of testing (path length to achieve DSXM coverage, baseline number of tests, numbers of culled infeasible and redundant tests, retained single-objective and multi-objective tests), with cost savings expressed as a percentage reduction of the baseline test suite.

The threat to *external validity* concerns the possible failure to generalize from the results of the seven case studies. This is inevitable because we have no way of sampling from the population of real case studies in a uniform manner. However, we did seek to use real case studies provided by industrial partners, which covered a spread of types of web service, including traditional client/server, modern rich-client services and a mixture of SOAP/WSDL, REST/JSON and HTML/Selenium implementations. The specifications also reflected widely varying designs, some state-contingent and others memory-contingent, showing the related benefits of different optimizations. We are therefore confident that the approach is applicable in a wide domain of applications.

7. COMPARISON WITH RELATED WORK

Surveys relevant to model-based testing in the cloud include [30, 31, 47–49]. While some are more general reviews of state-based [47] or model-based [49] testing tools, contrasting their approaches to specification, test generation and overall support [49], others consider service-oriented architectures explicitly [30, 31] looking at the adequacy of web-specifications for testing, or the merits of active testing (with explicit test suites) versus passive testing (non-interventionist observation of traces) [48].

7.1. Testing cloud and service-oriented systems

The systematic reviews of state-based testing tools [47, 49] analyse several tools from different perspectives: what test criteria are supported , what scaffolding criteria are supported (for the generation of test drivers, test stubs and test oracles) and what support is given for related activities (such as model creation, verification, debugging, and test execution). The authors concluded that, while a majority of the surveyed tools can generate abstract tests to satisfy simple criteria (e.g. state and transition coverage), very few support more complex criteria (like round-trip path coverage, or full predicate exploration), and data criteria are seldom supported. Many of the surveyed tools only offer partial support for test scaffolding construction, namely, they create test stubs that the tester must fill out by hand. One exception is Weissleder's tool ParTeG[§§§], which creates adapters and oracles for Java, as complete JUnit test cases. None of these surveyed tools were directed towards generating concrete tests for cloud-based services.

The survey by Cavalli et al. [48] acknowledges FSM-based testing as a sound basis for conformance testing in the cloud. The authors review the many variations of Chow's W-method [19] that infer system states from distinguishable input/output sequences but note that unique input/output sequences may not always exist. In passive testing, they survey approaches that use EFSMs as oracles working in parallel, during the normal execution of services, or offline, analysing collected traces post hoc. Passive approaches are less invasive but are unfortunately not complete, because actual service usage may only cover a subset of states and transitions. One such approach is Núñez and Hierons' cloud-centric adaptation of metamorphic testing [50], which infers faults from discrepancies between successive observations across multiple VM instances.

Canfora and Di Penta [30] provide an excellent survey on verification and testing approaches suitable for service-oriented architectures. They cover both functional and non-functional issues, highlighting how some traditional testing assumptions are violated by dynamic service discovery and late binding. This poses problems for performance testing (the context of service consumption cannot be guaranteed) and integration testing (the integration may not be known until late). For functional testing, they highlight the lack of observability as a problem, also the cost of repeated execution of test sequences. They survey a number of approaches for unit testing, using WSDL to inform the selections of test inputs and using BPEL (Business Process Execution Language) to extract models of component service behaviour but conclude that existing web service languages need augmentation with testing facets (to suggest test cases), functional descriptions (to express behaviour in more detail) and dependency analysis (to relate input/output pairs).

Bozkurt et al. [31] cover model-based testing (MBT) for service-oriented systems in more detail. They endorse the findings of Canfora and Di Penta, highlighting the inadequacy of WSDL when used as a model for test data generation. Early service-oriented testing approaches considered only WSDL interfaces [51] or REST protocols [52] as the basis for test generation, which lacked semantic content. They survey approaches that augment WSDL with semantic information, such as OWL-S (Web Service Ontology Language) in order to describe the semantic effects of executing operations but otherwise conclude that while most of the surveyed approaches successfully automate test execution and the generation of test data, few of them can automatically generate adequate test oracles (from the impoverished models).

[§§§]ParTeG (partition test generator) http://parteg.sourceforge.net

Workarounds that provide oracle information include Heckel and Lohmann's [53] augmentation with pre-generated design contracts, whereas Tsai et al. [54] require multiple implementations of the same service in order to detect delta differences (cf. metamorphic testing [50]). Many of the surveyed MBT approaches synthesized the models directly from the services, so were more likely to test the consistency of models generated by translation than test whether the code actually obeyed an independent specification. One exception was Sinha and Paradkar's [32] creation of an extended finite state machine (EFSM) specification from WSDL-S descriptions, which facilitated a number of test generation approaches, including symbolic execution for full predicate coverage and projection-coverage using user-specified coverage goals.

Other recent work with the same goal of testing cloud services from abstract specifications includes an automated approach that generates useful tests from algebraic specifications [55] but does not especially address test-coverage issues and a high-level method for specifying different kinds of robustness tests for a cloud platform, based on input validation, or state-space exploration, or concurrent access stress-testing [56], which does not address full test automation.

7.2. Extended finite-state machine testing

Our own approach follows in the tradition of using EFSM specifications to define the test coverage criteria; but we also seek to generate full concrete test data and oracle assertions automatically. Other authors have modelled service-oriented applications as some kind of finite state machine (FSM) for the sake of test generation but have not always achieved all the three aforementioned goals. We discuss some of these approaches hereafter.

The simple FSM approach by Endo and da Silva Simão [57] abstracts strongly over the service, in that it models the states and transitions of the FSM, but elides any more detailed functional description. We prefer the EFSM, in particular Laycock's Stream-X Machine [14], for its ability to model complex memory datatypes and realistic functional processes. While Wu and Huang recognize the superiority of EFSMs to handle memory [33], they only generate symbolic paths through the automaton, rather than concrete test cases with real inputs. Bertolino et al. [34] use the UML protocol state machine diagram and convert this into a Symbolic Transition System (an extension of Labelled Transition Systems with guards on symbolic memory), in order to generate symbolic test paths. Dranidis et al. specify systems directly as Stream X-Machines [40], whose transitions are modelled as functions operating on inputs, outputs and memory. Their tool JSXM generates fully executable JUnit tests for Java classes (POJOs); but to achieve this, the specification must include snippets of executable Java code for the functions, which are then pasted into the generated tests, whose coverage is motivated by the SXM specification.

The starting point of Ma et al. [58] is slightly different, in that their goal is to find suitable inputs to test services specified using BPEL. They describe a detailed manual transformation to convert BPEL descriptions into an SXM, whose desirable properties for test generation they acknowledge [14], but this is as far as the work goes. In a similar vein, Sun et al. [59] seek to attach EFSMs to WSDL descriptions, in order to use the power of EFSMs to motivate test generation with suitable coverage. They describe a procedure for partially converting WSDL descriptions but find unsurprisingly that the relevant semantic information has to be added by hand.

Sinha and Paradkar [32] first suggested augmenting WSDL-S interfaces with functional semantics expressed in the IOPE style (input, output, precondition and effect). They chose SWRL (Semantic Web Rule Language) to express the IOPE conditions and described a procedure to generate a testable EFSM. However, they were only able to derive an EFSM with one state and many guarded transitions. Ramollari et al. [26] were more successful in synthesizing a multi-state Stream X-Machine from semantically annotated WSDL (SAWSDL). They chose RIF-PRD (Rule Interchange Format - Production Rule Dialect) to describe the IOPE semantics of the system's functions. Their algorithm generated all the high-level states of an SXM automatically by observing the domain partitioning effect of operation guards on state variables. This approach came the closest to extrapolating the complete SXM automatically (they lacked a reasoning component to relate the prior and posterior states of memory).

Endo et al. [60, 61] adopt a related approach to test coverage using event sequence graphs (ESG). Their tool ESG4WSC (ESG for web service compositions) successfully generates positive and negative test cases [60], and they report results from industrial web service testing [61]. Endo et al. plan to investigate how to evolve ESG4WSC models in the direction of state machines, so that they can take states into account as well as events, and also handle more complex behaviour. Bertolino et al. [34] continue to use Symbolic Transition Systems in their PLASTIC framework for testing service-oriented applications [62], which share many similarities with the state machine model.

Others have used UML as the basis for state-based testing methods but do not address service-oriented issues. Holt et al. [63] use the model-based testing tool *TRUST* and evaluate six state-based coverage criteria in an industrial case study. Khalil and Labiche [64] generate test trees from UML state machine diagrams, extract test cases and compose test suites. Hasanan and Labiche [65] test real-time embedded systems, using random generation from RTEdge models combined with SPIN model-checking to discover test cases missed by random generation.

7.3. Improvements over similar approaches

Our approach is closest to Dranidis et al. [40], in that we also write our specifications directly as Stream X-Machines and generate concrete executable tests. Their JSXM tool (Java Stream X-Machine) automates DSXM testing for Java classes. A JSXM specification is created in a mixture of technology-neutral XML describing the state machine, and executable Java describing the functions, whereas we follow a fully abstract approach to specifying functions. Like Sinha and Paradkar [32] and Ramollari et al. [26], we consider IOPE semantics to be the right level of abstraction at which to express functional behaviour, because it is minimal and elegant. This contrasts, for example, with Tsai et al. [36], who extended WSDL with paired input–output dependencies, invocation sequences, hierarchical function descriptions and sequence specifications, which in our view detracts from the desirable abstraction of a specification.

Another difference between our approach and Dranidis et al. [40] is that we are able to verify that the specification is non-blocking and deterministic, before it is submitted for test generation; that is, we check that the specification satisfies the assumptions of the testing method. Our verification tool performs symbolic reasoning upon the IOPE specification, for which having explicit models of the functions is essential. (Dranidis et. al. are working on a separate reasoning component for JSXM – pers. comm.)

Furthermore, our test generator is able to reason fully about the whole DSXM specification, which solves many of the issues that arise with infeasible sequences, when these are generated from the automaton alone. JSXM requires the designer to be more careful in choosing a *realizable state cover* and *feasible separating sets* to avoid blocking issues when confirming the identity of reached states [40]. Our test generator simply alerts the designer if the blocking effects of guards cause loss of model coverage. We finesse the issue of state verification through a reliable oracle. Generally, our approach provides more support and requires less expertise to use.

Our DSXM test generator probably has one of the most ambitious test-pruning algorithms. Other SXM-based testing approaches [15–17] avoid discussion of how to treat infeasible sequences, assuming that the test generators must produce them but that they block during execution. We are able to detect infeasible sequences early by simulating the whole DSXM (including memory states and guards). Our pruning of idempotent paths containing trivial prefix cycles is new, based on an assumption that is discharged through testing. Our compression of test sequences by merging into multi-objective sequences is only possible because all state, function and output assertions are side-effect free. We do not force the DSXM through further state-distinguishing sequences, which produce divergent test paths offering far fewer opportunities to merge sequences.

Recent developments in Stream X-Machines include Ipate's combined method for testing component machines and their system-level integration in parallel [41]; the TXStates domain-specific language for multi-agent systems [66], which supports specifying agents as Stream X-Machines in NetLogo; and a timed extension to the SXM formalism [67], for which we are not yet aware of any testing method.

8. CONCLUSIONS AND FUTURE WORK

The first innovation of this work is the creation of a Deterministic Stream X-Machine (DSXM) specification model that exposes not only the abstract state-transition behaviour of the automaton but also the concrete input, output, precondition and effect (IOPE) behaviour of operations acting upon memory. Our tools reason explicitly about the blocking effects of guards on memory as well as blocking due to missing transitions in the automaton. The specification model also meets our goals for a web-transmissible format that may be used by different verification and testing tools at distributed locations in the cloud.

The second theoretical innovation is that simulation of the whole DSXM supports generation of complete test suites that are known to be feasible by construction. This overcomes the earlier problems found with test suites generated from the automaton alone. It means that the designer no longer has to worry about manually selecting a *realizable state cover* or *separating sets* to identify states and reduces the need for special inputs to ensure *controllability*. The tool will simply report if states or transitions are not eventually covered. This is better than approaches that determine test coverage from the automaton alone and later filter out infeasible tests, without any guarantee that coverage is still complete.

A third theoretical innovation is the verification algorithm for determinism and completeness, which converts a potentially large partition-finding problem in the state space of input and memory variables into a compact, finite symbolic checking problem, using conjunction terms to represent input and memory spaces and symbolic subsumption in lieu of concrete execution. Furthermore, we believe that this is the first alliance of such a verification tool with DSXM test generation, which verifies the testing method's assumption of a non-blocking and deterministic specification. This tool also meets our goals of aiding engineers to write correct specifications.

The fourth set of theoretical and engineering innovations are the three optimizations in test generation, which mitigate the problem of exponential growth in the size of test suites as tested paths grow longer. In particular, the *prefix complete* and *path complete* properties of the test generator support pruning of infeasible sequences and sequences with trivial prefix cycles. The verification method ensures that a witness remains for paths ending with blocking guards. The assumed idempotence of sequences with and without trivial prefix cycles is discharged by testing. These optimizations attack different kinds of test growth in systems with more guards or more states. The merging of test sequences with shared test objectives is only made possible by eliminating the mutating nature of W to determine states.

A fifth innovation lies in the novel solution to the hypothetical test function that maps sequences onto test inputs. The test input constraint, supplied as part of the specification, guides the constraint solver to provide test inputs that, under suitable memory conditions, will cause a transition eventually to fire. Abstract test inputs are then converted by test grounding into any desired execution format.

The different tools were evaluated in practice by members of the BrokerCloud consortium. During the development of the case study examples reported in Section 6, we collected informal feedback from industry partners SAP SE (Karlsruhe) and SingularLogic SA (Athens). Developers were surprised by their tendency to write incomplete specifications, either missing scenarios or making mistakes in the guards for the default "otherwise" scenario. In this respect, the verification tool proved useful in helping them to think more rigorously about equivalence partitions of input and memory. Sometimes, deciding how to represent the problem-state abstractly using *Set, List* and *Map* types was a challenge. Sometimes, choosing the best operation decomposition into scenarios was difficult: for example, the *ContactList* service needed four scenarios for removing an entry, to capture all possible selection states in the GUI after a deletion.

The same respondents agreed that test generation produced comprehensive test suites. Automatically generated tests detected subtle faults that their in-house QA procedures had not found. The most common kinds of extra fault detected were incorrect state transfer, leaving the application in the wrong state [27], and wrongly exposing internal variable information after a trivial cycle transition, which was a security vulnerability. In general, the exhaustive testing capability of the tool was far superior to in-house testing approaches that were based on walking through a finite number

of end-user scenarios. For the sake of test generation, developers found it intuitive to specify test input constraints as part of the specification. For services with greater memory-contingency, it was important to create enough data-entry scenarios that would load memory in all the distinct ways that would allow guards to be triggered. When creating a bespoke grounding to Selenium [27], the abstract test suite provided all necessary information to combine with an external DOM to generate the Selenium test script.

Future work arising from this project includes the further development of user-oriented tools for creating and editing specifications; the wider public offering of Testing as a Service to help increase the quality of brokered software services in the cloud; and also research into new areas enabled by having a completely modelled cloud service specification. One attractive future research area is to investigate the testability of service compositions. The current flat state model could be extended to permit decomposed states containing sub-state machines and composed models could be mapped by model transformation to equivalent flat models using UML or Statechart semantics. In this way, a single sign-in process for a given cloud platform could be wrapped around an arbitrary provided service and the additional test obligations determined automatically from the model composition. This could eventually support an incremental test generation approach for composed software services in the cloud.

ACKNOWLEDGMENTS

We would especially like to thank Wolfgang Schwach and Andreas Friesen at SAP SE (Karlsruhe) for supplying the *ContactList* and *HolidayBooking* case studies and for developing the bespoke Selenium grounding engine [27]. We also thank Panagiotis Gouvas at SingularLogic SA (Athens) for supplying the *VatClearance* case study and for co-ordinating developer responses. We also thank Simos Veloudis at SEERC (Thessaloniki) for supplying the *DocumentStore* case study. We finally thank Antonia Schwichtenberg and Simone Braun at CAS Software (Karlsruhe) and the anonymous reviewers for their comments. The research leading to these results was funded by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 328392 and the BrokerCloud project [4].

REFERENCES

1. Simons AJH, Bratanis K, Kourtesis D, Paraskakis I, Veloudis S, Verginadis Y, Mentzas G, Braun S, Rossini A. Advanced service brokerage capabilities as the catalyst for future cloud service ecosystems. *Proceedings of the 2nd InternationalWorkshop on CrossCloud Systems, CCB@Middleware 2014*: Bordeaux, France, 2014; 7:1–7:6. December 8, 2014.
2. Liu F, Tong J, Mao J, Bohn R, Messina J, Badger L, Leaf D. *NIST Cloud Computing Reference Architecture: Recommendations of the National Institute of Standards and Technology (Special Publication 500-292)*. CreateSpace Independent Publishing Platform: USA, 2011.
3. Bratanis K, Kourtesis D, Paraskakis I, Verginadis Y, Mentzas G, Simons AJH, Friesen A, Braun S. *A research roadmap for bringing continuous quality assurance and optimization to cloud service brokers*, 2013. eChallenges e-2013, ref 73.
4. BrokerCloud Consortium. *BrokerCloud: continuous quality assurance and optimisation for cloud brokers*, 2016. Last accessed, March 2019. https://cordis.europa.eu/project/rcn/105609/factsheet/en.
5. Lefticaru R, Simons AJ. X-machine based testing for cloud services. In *Advances in Service-Oriented and Cloud Computing*, Communications in Computer and Information Science, vol. 508. Springer International Publishing: Manchester, UK, 2015; 175–189.
6. Veloudis S, Paraskakis I, Petsos C. Cloud service brokerage: enhancing resilience in virtual enterprises through service governance and quality assurance. *Serv Oriented Comput Appl* 2017; **11**(4):445–458.
7. Kritikos K, Domaschka J, Rossini A. SRL: ascalability rule language for multi-cloud environments. *IEEE 6th International Conference on Cloud Computing Technology and Science, CloudCom 2014*: Singapore, 2014; 1–9. December 15-18, 2014.
8. Patiniotakis I, Verginadis Y, Mentzas G. Pulsar: preference-based cloud service selection for cloud service brokers. *J Internet Serv Appl* 2015; **6**(1):26:1–26:14.
9. Papazoglou MP, Pohl K, Parkin M, Metzger A. Service Research Challenges and Solutions for the Future Internet. In - *S-Cube - Towards Engineering, Managing and Adapting Service-Based Systems*, vol. 6500, Lecture Notes in Computer Science. Springer: Verlag Berlin Heidelberg, 2010; 372.

10. Bratanis K, Dranidis D, Simons AJH. SLAs for cross-layer adaptation and monitoring of service-based applications: A case study. In *Proceedings of the International Workshop on Quality Assurance for Service-Based Applications*, QASBA '11. ACM: New York, NY, USA, 2011; 28–32.
11. Rana OF, Warnier M, Quillinan TB, Brazier F, Cojocarasu D. *Managing violations in service level agreements*, Grid Middleware and Services. Springer: US, 2008.
12. Heckel R, Mariani L. Automatic conformance testing of web services. *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005*: Edinburgh, UK, 2005; 34–48. April 4-8, 2005, Proceedings.
13. Holcombe M. X-machines as a basis for dynamic system specification. *Softw Eng J* 1988; **3**(2):69–76.
14. Laycock GT. The theory and practice of specification based software testing. *PhD Thesis*, University of Sheffield, 1993.
15. Holcombe M, Ipate F. *Correct Systems - Building a Business Process Solution*, Applied Computing Series. Springer: London, New York, 1998.
16. Ipate F, Holcombe M. An integration testing method that is proved to find all faults. *Int J Comput Math* 1997; **63**:159–178.
17. Ipate F, Holcombe M. A method for refining and testing generalised machine specifications. *Int J Comput Math* 1998; **68**(3-4):197–219.
18. Ipate F. Complete Deterministic Stream X-Machine testing. *Form Asp Comput* 2004; **16**(4):374–386.
19. Chow TS. Testing software design modeled by finite-state machines. *IEEE Trans Software Eng* 1978; **4**(3):178–187.
20. Ipate F. Testing against a non-controllable Stream X-Machine using state counting. *Theor Comput Sci* 2006; **353**(1-3):291–316.
21. Ipate F, Holcombe M. Testing data processing-oriented systems from Stream X-Machine models. *Theor Comput Sci* 2008; **403**(2-3):176–191.
22. Naito S, Tsunoyama M. Fault detection for sequential machines by transition-tours. *Proceedings of the 11th IEEE Fault-Tolerant Computing Conference (FTCS 1981)*, IEEE Computer Society Press, 1981; 238–243.
23. Sabnani KK, Dahbura AT. A protocol test generation procedure. *Comput Netw* 1988; **15**:285–297.
24. Briand LC, Penta MD, Labiche Y. Assessing and improving state-based class testing: a series of experiments. *IEEE Trans Software Eng* 2004; **30**(11):770–793.
25. Simons AJH. *BrokerCloud Verification and Testing Tool Suite*, 2015. Last accessed, March 2019. http://staffwww.dcs.shef.ac.uk/people/A.Simons/broker/.
26. Ramollari E, Kourtesis D, Dranidis D, Simons AJH. Leveraging semantic web service descriptions for validation by automated functional testing. *The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC 2009*: Heraklion, Crete, Greece, 2009; 593–607. May 31-June 4, 2009, Proceedings.
27. Kiran M, Friesen A, Simons AJH, Schwach WKR. Model-based testing in cloud brokerage scenarios. In *Service-Oriented Computing - ICSOC 2013 Workshops, Lecture Notes in Computer Science, vol. 8377*. Springer: Berlin, Heidelberg, 2014; 192–208.
28. Hu S. SOAP testing for web services. *Master's thesis*, University of Sheffield, UK, 2015.
29. Cardoso JS, Pedrinaci C. Evolution and overview of linked USDL. *Exploring Services Science - 6th International Conference*: IESS 2015: Porto, Portugal, 2015; 50–64. February 4-6, 2015, Proceedings.
30. Canfora G, Penta MD. Service-oriented architectures testing: a survey. In *Software Engineering, International Summer School (ISSE 2006-2008), Lecture Notes in Computer Science, vol. 5413*. Springer: Berlin, Heidelberg, 2008; 78–105.
31. Bozkurt M, Harman M, Hassoun Y. Testing and verification in service-oriented architecture: a survey. *Softw Test, Verif Reliab* 2013; **23**(4):261–313.
32. Sinha A, Paradkar AM. Model-based functional conformance testing of web services operating on persistent data. *Proceedings of the 2006 Workshop on Testing, Analysis, and Verification of Web Services and Applications, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), TAV-WEB 2006*: Portland, Maine, USA, 2006; 17–22. July 17, 2006.
33. Wu C, Huang C. The web services composition testing based on extended finite state machine and UML model. *Fifth International Conference on Service Science and Innovation, ICSSI 2013*: Kaohsiung, Taiwan, 2013; 215–222. 29-31 May, 2013.
34. Bertolino A, Frantzen L, Polini A, Tretmans J. Audition of web services for testing conformance to open specified protocols. In *Architecting Systems with Trustworthy Components, Lecture Notes in Computer Science, vol. 3938*. Springer: Berlin, Heidelberg, 2004; 1–25.
35. Noikajana S, Suwannasart T. An improved test case generation method for web service testing from WSDL-S and OCL with pairwise testing technique. *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, COMPSAC 2009*, Vol. 1: Seattle, Washington, USA, 2009; 115–123. July 20-24, 2009.
36. Tsai W, Paul RA, Wang Y, Fan C, Wang D. Extending WSDL to facilitate web services testing. *7th IEEE International Symposium on High-Assurance Systems Engineering (HASE 2002)*: Tokyo, Japan, 2002; 171–172. 23-25 October 2002.
37. Kopecký J, Vitvar T, Bournez C, Farrell J. SAWSDL: semantic annotations for WSDL and XML schema. *IEEE Internet Computing* 2007; **11**(6):60–67.
38. Eilenberg S. *Automata, Languages, and Machines*. Academic Press, Inc: Orlando, FL, USA, 1974.
39. Spivey JM. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc: Upper Saddle River, NJ, USA, 1989.

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.

40. Dranidis D, Bratanis K, Ipate F. JSXM: A tool for automated test generation. In *Software Engineering and Formal Methods - SEFM 2012, Lecture Notes in Computer Science, vol. 7504*. Springer: Berlin, Heidelberg, 2012; 352–366.
41. Ipate F, Dranidis D. A unified integration and component testing approach from Deterministic Stream X-Machine specifications. *Form Asp Comput* 2016; **28**(1):1–20.
42. Balanescu T, Gheorghe M, Ipate F, Holcombe M. Formal black box testing for partially specified deterministic finite state machines. *Found Comput Decis Systems* 2003; **28**(1):17–28.
43. Bogdanov K, Holcombe M, Ipate F, Seed L, Vanak SK. Testing methods for X-machines: a review. *Form Asp Comput* 2006; **18**(1):3–30.
44. Jackson D. *Software Abstractions: Logic, Language, and Analysis*, Revised. The MIT Press: Cambridge MA, USA, 2011.
45. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co, Inc: Boston, MA, USA, 1995.
46. Runeson P, Höst M. Guidelines for conducting and reporting case study research in software engineering. *Empir Softw Eng* 2009; **14**(2):131–164.
47. Shafique M, Labiche Y. A systematic review of state-based test tools. *Int J Softw Tools Technol Transf* 2015; **17**(1):59–76.
48. Cavalli AR, Higashino T, Núñez M. A survey on formal active and passive testing with applications to the cloud. *Ann Télécommun* 2015; **70**(3-4):85–93.
49. Li W, Le Gall F, Spaseski N. A survey on model-based testing tools for test case generation. In *Tools and Methods of Program Analysis*, VItsykson, AScedrov, VZakharov (eds). Springer International Publishing: Cham, 2018; 77–89.
50. Nuñez A, Hierons RM. A methodology for validating cloud models using metamorphic testing. *Ann Télécommun* 2015; **70**(3-4):127–135.
51. Bai X, Dong W, Tsai W-T, Chen Y. WSDL-based automatic test case generation for web services testing. In *Proceedings of the IEEE InternationalWorkshop*, SOSE '05. IEEE Computer Society: Washington, DC, USA, 2005; 215–220.
52. Chakrabarti SK, Kumar P. Test-the-REST: an approach to testing RESTful web services. *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, 2009; 302–308.
53. Heckel R, Lohmann M. Towards contract-based testing of web services. *Electr Notes Theor Comput Sci* 2005; **116**:145–156.
54. Tsai W, Chen Y, Zhang D, Huang H. Voting multi-dimensional data with deviations for web services under group testing. *25th International Conference on Distributed Computing SystemsWorkshops (ICDCS 2005Workshops)*: Columbus, OH, USA, 2005; 65–71. 6-10 June 2005.
55. Liu D, Liu Y, Zhang X, Zhu H, Bayley I. Automated testing of web services based on algebraic specifications. *2015 IEEE Symposium on Service-Oriented System Engineering, SOSE 2015*: San Francisco Bay, CA, USA, 2015; 143–152. March 30 - April 3, 2015.
56. Cotroneo D, Frattini F, Pietrantuono R, Russo S. State-based robustness testing of IaaS cloud platforms. *Proceedings of the 5th InternationalWorkshop on Cloud Data and Platforms*: *CloudDP@EuroSys* 2015: Bordeaux, France, 2015; 3:1-3:6. April 21-24, 2015.
57. Endo AT, Simão A. Model-based testing of service-oriented applications via state models. *IEEE International Conference on Services Computing (SCC 2011)*, 2011; 432–439.
58. Ma C, Wu J, Zhang T, Zhang Y, Cai X. Testing BPEL with Stream X-Machine. *International Symposium on Information Science and Engineering (ISISE '08)*, Vol. 1, 2008; 578–582.
59. Sun F, Liao L, Zhang L. Model-based testing of web service with EFSM. In *Practical Applications of Intelligent Systems, Advances in Intelligent Systems and Computing*, Vol. 279. Springer: Berlin, Heidelberg, 2014; 91–100.
60. Belli F, Endo AT, Linschulte M, da Silva Simão A. A holistic approach to model-based testing of web service compositions. *Softw Pract Exper* 2014; **44**(2):201–234.
61. Endo AT, Bernardino M, Rodrigues EM, da Silva Simão A, de Oliveira FM, Zorzo AF, Saad RS. An industrial experience on using models to test web service-oriented applications. *The 15th International Conference on Information Integration and Web-based Applications & Services*, IIWAS '13: Vienna, Austria, 2013; 240. December 2-4, 2013.
62. Bertolino A, Angelis GD, Frantzen L, Polini A. The PLASTIC framework and tools for testing service-oriented applications. In *Software Engineering, International Summer School (ISSE 2006-2008), Lecture Notes in Computer Science, vol. 5413*. Springer: Berlin, Heidelberg, 2008; 106–139.
63. Holt NE, Briand LC, Torkar R. Empirical evaluations on the cost-effectiveness of state-based testing: an industrial case study. *Inform Software Tech* 2014; **56**(8):890–910.
64. Khalil H, Labiche Y. State-based tests suites automatic generation tool (STAGE-1). *41st IEEE Annual Computer Software and Applications Conference, COMPSAC 2017*, Vol. 1: Turin, Italy, 2017; 357–362. July 4-8, 2017.
65. Hasanain W, Labiche Y, Gheorghe S. Automated state-based online testing real-time embedded software with RTEdgeMODELSWARD 2015. *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2015), Vol. 1*: Angers, Loire Valley, France, 2015; 294–302. 9-11 February, 2015.
66. Sakellariou I, Dranidis D, Ntika M, Kefalas P. From formal modelling to agent simulation execution and testing. *ICAART 2015 - Proceedings of the International Conference on Agents and Artificial Intelligence*, Vol. 1: Lisbon, Portugal, 2015; 87–98. 10-12 January, 2015.
67. Merayo MG, Núñez M, Hierons RM. Testing timed systems modeled by Stream X-Machines. *Softw System Model* 2011; **10**(2):201–217.

©2020 The Authors. *Software Testing, Verification & Reliability* published by John Wiley & Sons Ltd.