from approaches such as Endo et al. [71] and Blanco et al.'s [29], where required number of test cases for coverage are much fewer than random, prove that they reduce the cost of testing not merely by automating the input generation process but also by reducing the number of required test cases to run on the service(s) under test.

Some of the model-based test case generation approaches [29, 71, 86, 93, 109, 110, 240, 251] described above have a single common feature. In traditional MBT, models are created using requirements or specifications separately from the executable code. However, almost all of the model-based approaches described above generate models from the executable code itself. This abstraction/translation process leads to a model that reflects the behaviour of the executable code rather than a model that reflects the expected behaviour of the system. Thus, testing using such a model will lead to testing of the translation process rather than testing of the actual system. By definition, the errors revealed using these approaches can only be those errors introduced by their translation process.

This problem does not affect the approaches using formal verification methods. In formal verification, logical properties that the SUT is checked against are not derived from the executable code. Other approaches in this section such as Felderer et al. [74] require the test model to be generated separately.

9. Interoperability Testing of Service-centric Systems

Interoperability is the ability of multiple components to work together. That is, to exchange information and to process the exchanged information. Interoperability is a very important issue in open platforms such as SOA. Even though web services must conform to standard protocols and service specifications, incompatibility issues might still arise.

The need for interoperability among service specifications is recognized by industry and the WS-I, an open industry organization, formed by the leading IT companies. The organization defined a WS-I Basic Profile [222] in order to enforce web service interoperability. WS-I organization provides interoperability scenarios that need to be tested and a number of tools to help testing process. Kumar et al. [187] describe the possible interoperability issues regarding core web service specifications such as SOAP, WSDL and UDDI and explain how the WS-I Basic Profile provides solutions for the interoperability issues with web service specifications.

There are also interoperability problems that might be caused by web service toolkits such as Java Axis and .Net. For example using dynamic data structures in services which use a certain toolkit might cause interoperability problems (due to message consumption errors) for the clients using other toolkits [191]. Interoperability problems do not occur only among different toolkits but might also occur in different versions of the same toolkit. This is also another important interoperability aspect that needs to be tested by both the developer and the certifier.

DOI: 10.1002/stvr

9.1. Perspectives in Interoperability Testing

Since the aim of the interoperability is to observe whether the services exchange messages as expected it can be performed by all the stakeholders in SOA. Interoperability testing of services (service protocols and interfaces) is very important for the provider and the certifier. Testing for interoperability must be included in reliability measurement. Even though most of the approaches in this section target services there are approaches such as Narita et al. [152] and Yu et al. [249] that target service compositions. These approaches can only be performed by the integrator.

9.2. Interoperability Testing Approaches

The need for testing the interoperability among services is also recognized by researchers. For example, Bertolino and Polini [25] propose a framework that tests the service interoperability using service's WSDL file along with a Protocol State Machine Diagram (PSM) [168] provided by the service provider. The PSM diagram carries information on the order of the operation invocations for a service. The proposed framework checks the order of the web service invocations among different web services and tries to point out possible interoperability problems.

Yu et al. [249] propose an ontology-based interoperability testing approach using the communication data among web services. The proposed approach captures communication data and stores it in an ontology library. The data in the library is analysed and reasoning rules for error analysis and communication data are generated in order to run with the JESS reasoning framework [103]. Using the rules from the JESS framework gives this approach the ability to adapt certain problems such as network failure or network delay. The framework can also replay the errors that have been identified by using a Petri Net graphic of the communicating web services.

Smythe [192] discusses the benefits of the model-driven approach in the SOA context and proposes an approach that can verify interoperability of services using UML models. The author points out the need for UML-profile for SOA that contains the interoperability specification in order to use with the proposed approach. Using the proposed the UML-profile, test cases can be generated for testing the conformance of the web service's interoperability.

Similarly, Bertolino et al. [25] propose a model-based approach for testing interoperability. In the proposed environment, web services are tested before registration. In this approach, service provider is expected to provide information on how to invoke a web service using an UML 2.0 behaviour diagram.

Ramsokul and Sowmya [174] propose the use of ASEHA framework to verify the service protocol's implementation against its specification. They claim that the ASEHA framework is capable of modelling complex protocols such as Web Services Atomic Transaction (WS-AtomicTransaction) [226] and Web Services Business Activity (WS-BusinessActivity) [227]. The proposed ASEHA framework captures the SOAP messages from services, maps them into ASEHA automata and verifies the protocol implementation against its specification.

Guermouche and Godart [87] propose a model-checking approach for verifying service interoperability. The approach uses UPPAAL model checker and includes timed properties

DOI: 10.1002/stvr

in order to check for timed conflicts among services. The approach is capable of handling asynchronous timed communications.

Betin-Can and Bultan [27] propose the use of a model, based on HSMs for specifying the behavioural interfaces of participating services in a composite service. Betin-Can and Bultan suggest that verification of the web services that are developed using the Peer Controller Pattern is easier to automate and propose the use of HSMs as the interface identifiers for web services in order to achieve interoperability. Betin-Can and Bultan propose a modular verification approach using Java PathFinder to perform interface verification and SPIN for behaviour verification and synchronizability analysis. The use of the proposed approach improves the efficiency of the interface verification significantly as claimed by the Betin-Can and Bultan.

Narita et al. [152] propose a framework for interoperability testing to verify web service protocols, especially aimed at reliable messaging protocols. Narita et al. claim that none of the existing testing tools aims to perform interoperability testing for communication protocols. They also highlight the need for a testing approach that covers the reliable messaging protocols, capable of executing erroneous test cases for these protocols. As a result their framework is capable of creating test cases containing erroneous messages by intercepting messaging across web services.

Passive testing is the process monitoring the behaviour of the SUT without predefining the input(s) [46]. The first passive testing approach for web services is proposed by Benharref et al. [20]. This EFSM-based approach introduces an online observer that is capable of analysing the traces and reporting faults. The observer also performs forward and backward walks on the EFSM of the WSUT in order to speed up the state recognition and variable assignment procedures.

Passive conformance testing of EFSMs was proposed by Cavalli et al. [46] where testing artefacts called 'invariants' enable testing for conformance. These invariants contain information on the expected behaviour of the SUT and used in testing traces. Several authors extended this research to web services domain.

For example, Andrés et al. [2] propose the use of passive testing for service compositions. The proposed invariants contain information on expected behaviour of services in the composition and their interaction properties. The proposed passive testing approach checks local logs against invariants in order to check for the absence of prescribed faults. Morales et al. [147] propose a set of formal invariants for passive testing. In this approach, time extended invariants are checked on collected traces. The approach uses a tool called TIPS that enables passive testing.

Cao et al. [43] also propose a passive testing approach for service compositions. The proposed approach enables both online and offline verification using constraints on data and events called security rules. The security rules are defined in the Nomad language. The authors also present a tool that automates the passive testing for behavioural conformance called RV4WS.

DOI: 10.1002/stvr

9.3. Experimental Results

Narita et al. [152] performed experiments on Reliable Messaging for Grid Services (RM4GS) version 1.1, an open source implementation of WS-Reliability 1.1. The framework performs coverage-based testing in order to check for conformance to its specifications. It also performs application-driven testing to check for interoperability. The errors introduced by the framework include losing a package, changing massage order and sending duplicate messages. During coverage testing, the framework tested 180 out of 212 WS-Reliability items and unable to find any errors but raised 3 warnings. During application-driven tesing 4 errors were revealed and 4 warnings were raised.

Betin-Can and Bultan [27] experimented on the travel agency and the order handling examples. For the travel agency different peers of the program took between 4.61 seconds to 9.72 seconds for interface verification. The resources used in this experiment is range between 3.95MB and 19.69MB of memory. Synchronizability analysis of the travel agency (which was 8,911 states) took 0.38 seconds and used 5.15 MB of memory. Order handling interface verification for peers took between 4.63 to 5.00 seconds and used 3.73MB to 7.69MB of memory. Synchronizability analysis of order handling (which was 1,562 states) took 0.08 seconds and used 2.01MB of memory.

9.4. Discussion

As stated, interoperability is one of the major potential of Web services. Web services must be tested for interoperability in order to achieve this potential. Interoperability issues that are caused by different versions of the protocols such as SOAP are addressed by industry in WS-I. However, other challenges requires approaches such as Tsai et al. [208] and Bertolino et al.'s [25] where interoperability is tested before service registration. Testing web services before the registration process can prevent many of the possible interoperability problems and this might increase the confidence in the registered services.

The approaches in this section are divided into three main groups. The first group aims to verify service protocols, such as the work of Narita et al. [152] and Ramsokul and Sowmya [174]. The second group verify interfaces and communication among services, such as the work of Betin-Can and Bultan [27], Smythe [192] and Yu et al. [249]. The third group are the passive testing approaches, such as the work of Andrés et al. [2], Morales et al. [147] and Cao et al. [43].

The approaches in this section can also be grouped in terms of their cost. The approaches that use formal verification and passive testing approaches will reduce the cost of testing for the integrator. Formal verification approaches cover service and protocol verification respectively. Using them together allows a complete offline interoperability testing for the integrator. The passive testing approaches will provide the integrator to ability to detect real-world usage faults. For the approaches where test cases are generated, the cost can be higher. The passive testing approaches increase the cost of testing for the integrator due the effort required to create the necessary invariants.

DOI: 10.1002/stvr

10. Integration Testing of Service-centric Systems

Integration testing is crucial in most fields of engineering to make sure all the components of a system work together as expected. The importance of performing integration testing is also well established in software engineering. Since the idea behind SOA is to have multiple loosely coupled and interoperable distributed services to form a software system, integration testing in SOA is at least as important. By performing integration testing, all the elements of a ScS can be tested including services, messages, interfaces, and the overall composition.

Bendetto [19] defined the difference between integration testing of traditional systems and ScS. Canfora and Di Penta [38] point out the challenges in integration testing in SOA. According to Bendetto and Canfora and Di Penta the challenges of integration testing in ScS are:

1. Integration testing must include the testing of services at the binding phase, workflows and business process connectivity. Business process testing must also include all possible bindings.
2. Low visibility, limited control and the stateless nature of SOA environment make integration testing harder.
3. Availability of services during testing might also be a problem.
4. Dynamic binding makes the testing expensive due to the number of required service calls.

10.1. Perspectives in Integration Testing

As might be expected, integration testing is only performed by the integrator. The rest of the stakeholders are not capable of performing integration-oriented approaches due to the lack of observability.

Most of the approaches in this section target service compositions using static binding. By contrast for dynamic SOA, performing integration testing can be very challenging due to ScS's configuration being available only at run-time (this problem is referred as the "run-time configuration issue" in the rest of this paper). In dynamic SOA, the integrator needs to test for all possible bindings, which can increase the cost of testing greatly.

10.2. Integration Testing Approaches

One of the earliest works on integration testing of web services is Tsai et al.'s [210] Coyote framework. Coyote is an XML-based object-oriented testing framework that can perform integration testing. Coyote is formed of two main components; a test master and a test engine. The test master is capable of mapping WSDL specifications into test scenarios, generating test cases for these scenarios and performing dependency analysis, completeness and consistency checking. The test engine on the other hand performs the tests and logs the results for these tests.

In software development there is a concept called Continuous Integration (CI) [67]. CI is performed by integrating the service under development frequently. CI also requires

DOI: 10.1002/stvr

continuous testing. Continuous Integration Testing (CIT) allows early detection of problems at the integration level. Huang et al. [97] propose a simulation framework that addresses the service availability problem using CIT. The proposed framework automates the testing by using a surrogate generator that generates platform specific code skeleton from service specifications and a surrogate engine that simulates the component behaviour according to skeleton code. Huang et al. claim that the proposed surrogates are more flexible than the common simulation methods such as stubs and mocks and the simulation is platform-independent.

Liu et al. [130] also propose a CIT approach with which executable test cases carry information on their behaviour and configuration. In the proposed approach, integration test cases are generated from sequence diagrams. The authors also introduce a test execution engine to support this approach.

Peyton et al. [167] propose a testing framework that can perform ”grey-box” integration testing of composite applications and their underlying services. The proposed framework is implemented in TTCN-3 [73], an European Telecommunications Standards Institute standard test specification and implementation language. It is capable of testing the composite application behaviour and interaction between participating web services. The framework increases the visibility and the control in testing by inserting test agents into a web service composition. These agents are used in analysing HTTP and SOAP messages between the participating services.

Mei et al. [143] address the integration issues that might be caused by XPath in BPEL processes such as extracting wrong data from an XML message. The proposed approach uses CFGs of BPEL processes along with another graph called XPath Rewriting Graph (XRG) that models XPath conceptually (models how XPath can be rewritten). Mei et al. create a model that combines these two graphs called X-WSBPEL. Data-flow testing criteria based on def-use associations in XRG are defined by Mei et al. and using these criteria, data-flow testing can be performed on the X-WSBPEL model.

Angelis et al. [3] propose a model-checking integration testing approach. Test cases are derived from both orchestration definition and specification of the expected behaviour for the candidate services. The authors also present a tool that supports this approach.

Tarhini et al. [202] address the issue of web service availability and the cost of testing. Tarhini et al. solve these problems by finding suitable services before the integration process and using only the previously selected services according to their availability. In this approach, testing to find suitable services is accomplished in four stages. The first stage is the ”find stage“ in which candidate web services from a service broker are found. In the second stage selected web services are tested for their correct functionality. At the third stage, each web service is tested for interaction as a stand-alone component and, if it passes this stage, it is tested for interactions with the rest of the components. When a web service passes all the required steps it is logged into the list of services to be invoked at runtime. The proposed framework uses a modified version of the Coyote framework for the automation of testing.

Yu et al. [248] address the interaction problems within OWL-S compositions. Yu et al.’s approach tests interaction among participating web services using interaction requirements. Yu et al. propose an extension to existing OWL-S models to carry these requirements.

DOI: 10.1002/stvr

There are also previously mentioned approaches that are capable of performing integration testing. For example, Tsai et al.'s ASTRAR framework [207] and the proposed Enhanced UDDI server [208] are also capable of performing integration testing. Similarly, Lenz et al.'s [118] model-driven testing approach can be used for integration testing.

10.3. Experimental Results

Huang et al. [97] experimented on an human resources system that is transformed into a web service. In this system, there are 17 components in the business layer with 58 interfaces and 22 components in data layer with 22 interfaces. During the pure simulation without real components 7 bugs are identified caused by issues such as reference to a wrong service, interface mismatch and missing service. During real component tests (includes surrogates as well) 3 bugs are identified for five components.

Mei et al. [143] experimented on the eight popular BPEL examples. The authors created mutants by injecting faults into three different layers in the composition BPEL, WSDL and XPath. Test sets created by the approach achieved almost 100% coverage in all test criteria considered. The authors also compared their fault detection rates with random testing. Overall the minimum detection rates for this approach are between 53% to 67% where as random only achieved 18%. Mean rates of fault detection rates for this approach are between 92% to 98% whereas random achieved 73%. The authors also investigated the performance of the approach. It took between 0.45 to 1.2 second for generating test sets with a 2.4 GHz processor and 512MB memory.

Liu et al. [130] experimented on two synthetic examples: an HR system and a meeting room management system. The approach revealed 22 bugs in the HR system and 7 in the meeting room system. The approach revealed faults in categories such as incorrect method calls, incorrect parameter passing, configuration problems and interface/function mismatches.

10.4. Discussion

Integration testing is one of the most important testing methodologies for SOA. The challenges that the integrator faces during integration testing are addressed by some of the approaches mentioned in this section such as Tarhini et al. [202], Huang et al.'s [97] and Liu et al.'s [130] frameworks.

Huang et al.'s [97] CI based integration testing can be very useful by starting testing early. The ability to use surrogate services can also help to reduce the cost of testing. Since surrogate services can be generated automatically using them does not increase the overall cost. The only handicap of this approach might be finding/generating suitable web service specifications to be used in surrogate generation. One other issue that can increase the cost of testing is the lack of automated test case generation within the framework.

Liu et al. [130] partly automate test case generation in CIT using sequence diagrams. This approach makes use of Huang et al.'s work and is able simulate unavailable components. As a result, it has the same restrictions as Huang's work regarding the service simulation. The approach's ability to verify execution traces using object comparison and expression verification is the main advantage of this approach.

DOI: 10.1002/stvr

Mei et al.'s [143] approach addresses a problem that is overlooked by many developers. Integration issues that can be caused by XPath are an important problem in service compositions that need to be tested. The results from their experiments prove the effectiveness of their approach in revealing these problems.

Almost all of the approaches discussed above will have problems adapting to dynamic environments. For example, Tarhini et al.'s [202] approach might be rendered inapplicable due to not being able to know the services available at run-time and not being able choose the service to bound at run-time. On the other hand, Huang et al. [97], Peyton et al. [167], Tsai et al. [210] and Mei et al.'s [143] approaches might become more expensive to perform.

11. Collaborative Testing of Service-centric Systems

Collaborative software testing is the testing concept where multiple stakeholders involved in a web service, such as developer, integrator, tester and user, participate in the testing process. Collaborative testing is generally used in testing techniques such as usability walk-through where correct functionality is tested with participation of different stakeholders.

Challenges involving testing ScS are identified by Canfora and Di Penta [38], some of which require collaborative solutions. These challenges that might require collaborative solutions are:

1. Users not having a realistic test set.
2. Users not having an interface to test web service systems.
3. The need for a third-party testing and QoS verification rather than testing by each service user.

11.1. Perspectives in Collaborative Testing

Collaborative testing requires collaboration among stakeholders. The proposed approaches described in this section seek to establish a collaboration between the developer and the integrator. Some of the approaches include a third-party in order to increase testability.

11.2. Collaborative Testing Approaches

Tsai. et al. [204] propose a Co-operative Validation and Verification (CV&V) model that addresses these challenges instead of the traditional Independent Validation and Verification (IV&V). One example of this collaborative testing approach is Tsai et al.'s proposed enhanced UDDI server [208]. This UDDI server further enhances the verification enhancements in UDDI version 3 [214]. These proposed enhancements include:

1. The UDDI server stores test scripts for the registered web services.
2. The UDDI server arranges test scripts in a hierarchical tree of domains and sub-domains.
3. The UDDI server has an enhanced registration mechanism called check-in. The Check-in mechanism registers a web service if it passes all test scripts for its related domain and sub-domain.

DOI: 10.1002/stvr