An Automatic Test Case Generation Framework for Web Services

Yongyan Zheng, Jiong Zhou, Paul Krause
Department of Computing, University of Surrey, Guildford, UK
Email: {y.zheng,j.zhou,p.krause}@surrey.ac.uk

Abstract— **BPEL (Business Process Execution Language) as a de-facto standard for web service orchestration has drawn particularly attention from researchers and industries. BPEL is a semi-formal flow language with complex features such as concurrency and hierarchy. To test a model thoroughly, we need to cover different execution scenarios. As is well known, it is tedious, time-consuming, and error prone to design test cases manually, especially for complex modelling languages. Hence, it is desirable to apply existing model-based-testing techniques in the domain of web services. We proposed WSA (Web Service Automata) to be the operational semantics for BPEL. Based on WSA, we propose a model checking based test case generation framework for BPEL. The SPIN and NuSMV model checkers are used as the test generation engine, and the conventional structural test coverage criteria are encoded into LTL and CTL temporal logic. State coverage and transition coverage are used for BPEL control flow testing, and all-du-path coverage is used for BPEL data flow testing. Two levels of test cases can be generated to test whether the implementation of web services conforms to the BPEL behaviour and WSDL interface models. The generated test cases are executed on the JUnit test execution engine.**

Index Terms— **web services, finite state machine, model checking, test coverage, test case generation**

I. Introduction

Web services is an emerging paradigm which provides a flexible, re-usable, and loosely coupled model for distributed computing. BPEL is the de-facto industry standard language to model the behaviour of web service compositions. BPEL is a semi-formal flow-based language with complex features, which may thus include fault behaviours. It becomes essential to verify a web service design before publishing it, and to test whether the published service conforms to the design model. However, the manual writing and verification of test cases for complex models is tedious, time-consuming and error prone. Hence, it is vital to automate this process.

A number of proposals have been made that use model checking for the rigorous verification of BPEL [1]. We identify two problems of the existing approaches. First, BPEL activity relationships can be categorised into control-flow and data-flow. Since BPEL is a semi-formal flow language, various formal semantics have been proposed, so that BPEL models can be verified rigorously. However, most current formal models only focus on modelling BPEL control flow, and do not cover the BPEL data flow analysis. Second, there exist two kinds of interactions of BPEL: internal and external. The external interactions between BPEL models are by message passing. The internal interactions between activities of a BPEL model, are modelled explicitly by control dependencies and implicitly by data sharing. Those internal interactions caused by data sharing will be omitted if an approach does not cover the BPEL data flow analysis. Furthermore, there is less material in the literature on automatic test case generation from BPEL models. From the model-driven-testing point of view, existing model checking tools can be reused for the purpose of verification and testing of BPEL. With model checking, a BPEL model can not only be a design model for verification, but also be a test model for deriving test cases.

The formal semantics proposed to date for BPEL can be categorised as process algebra based, Petri-net based, and automata based. Since the verification tools for process algebras are less mature, and using Petri-nets as the formal models for BPEL has scalability problems, we choose an automata-based approach, so that a rich range of mature model checking tools can be applied. Our formal model is designed to be used by the verification tools. We propose a Web Service Automaton (WSA), an extension of Mealy machines, which covers data, supports message passing communication, and adapts the asynchronous interleaving semantics. In this paper, we justify the suitability of WSA for BPEL on two counts: (1) our model supports separate analyses of BPEL control and data flows; (2) its message passing communication provides a uniform semantics for both BPEL internal and external interactions. Based on WSA, we provide a model checking based test case generation framework for BPEL. We support the application of both SPIN and NuSMV model checkers as the test generation engines, and we encode the conventional structural test coverage criteria into LTL and CTL temporal logic. State coverage and transition coverage are used for BPEL control flow testing, and all-du-path coverage is used for BPEL data flow testing. Test cases can be generated to test whether the implementation of web services conforms to the BPEL behaviour and WSDL interface models. To our knowledge, none of the prior research studies

This paper is based on "A Model Checking based Test Case Generation Framework for Web Services," by Y. Zheng, J. Zhou, and P. Krause, which appeared in the Proceedings of the International Conference on Information Technology (ITNG), Las Vegas, Nevada, USA, April 2007.

This work was supported by the EU FP6 funded project Digital Business Ecosystem.

the verification and test case generation of both BPEL control-flows and data-flows in a unified approach. In summary, our main contributions lie in the two aspects: (1) proposing a web service automaton (WSA) as the operational semantics for BPEL, which is suitable for model checking; (2) analysing BPEL control and data dependencies in WSA; (3) presenting an automatic test framework for BPEL web services.

The rest of the paper is organized as follows. Section II introduces the background of BPEL language, the semantics of our WSA, and the modelling of BPEL in WSA. Section III presents our test framework in details. Section IV provides a case study and tool support to demonstrate our approach. Section V reviews the relevant literature. Finally, section VI concludes the paper and outlines future work.

II. BACKGROUND

In this section, we briefly describe BPEL features, present our web service automata (WSA) together with how to capture BPEL control and data dependencies in WSA, and model checking in testing. In the following sections, we use *machine* as shorthand for a web service automaton, and call the machine associated with BPEL x activity as x machine. In state machine diagrams, an initial state is pointed by an arrow started with a filled black circle, and a final state is shaded.

A. BPEL Language

BPEL is a flow-based language in XML format. BPEL consists of two categories of activities: basic and structured activities. Basic activities are atomic actions, including *receive, reply, assign, invoke, throw, terminate, empty*, and *wait*. As with programming languages, the structured activities impose control flow dependency constrains on the executions of either the basic or structured activities within them. A structured activity can contain an arbitrary depth of sub-activities. The structured activities include *pick, switch, while, sequence, flow, scope, eventHandlers, faultHandlers, compensationHandler*. For data handling, BPEL uses the *blackboard* approach, where a set of variables is shared by all activities within the same scope.

Since every web service shall have its own business logic, a web service should be associated with a behavioral model in addition to the web service interface description. As a well known orchestration language for the interactions of multiple web services, BPEL is rich enough to also describe the behaviours of single services. So, we suppose that a web service is associated with two models: a BPEL process as the behavioural model, and a WSDL description as the interface. From the testing point of view, when more than one BPEL process is considered, the system boundary needs to be included. The process within the system boundary are called *SUT* (service under test), and a process outside the system boundary is *tester*. We use the loan approval service from the BPEL standard [2] as the discussion example. The loan approval service includes four web services: customer, approval, assessor, and approver. The approval service acts as the orchestration service to interact with other services. The BPEL standard provides an *approval* BPEL process for the loan approval example.

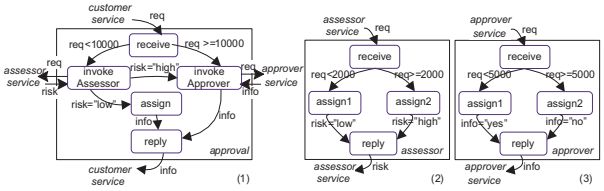

Figure 1. The loan approval business scenarios

The business requirement of the approval service is illustrated in part (1) of Fig 1. The approval service starts by receiving a request from the customer. If the request is less than 10000, it forwards the requests to the assessor service; otherwise it forwards the request to the approver service. The assessor evaluates the request and returns a risk result. The approver evaluates the request and returns an approved result. When the returned risk from the assessor is low then the approval service replies the customer by a granted loan; otherwise if the returned risk from the assessor is high, the approval service calls the approver. After getting response from the approver, the approval service replies the customer of the loan decision. In addition to the approval service, the assessor and approver services must have some business logics to decide whether an incoming request has high risk or low risk, and whether a high risk request can be granted. We add BPEL processes for the assessor and approver services. Their business requirements are shown in parts (2) and (3) of Fig 1.

B. Web Service Automata

The static semantics of a web service automaton extends a finite state machine with signature, data structure, and message storage schema. Asynchronous execution of web services is achieved by using queues for message processing. The default queuing protocol in WSA is to associated an FIFO queue for each message. We assume that each WSA is equipped with a **finite multi-set** buffer to store the incoming messages. WSA communicate by message passing.

Definition 1: A **Web Service Automaton** (WSA) M is a sextuple $M = (I_M, S_M, s_{0M}, S_{fM}, T_M, \delta_M)$. As a convention, we omit the subscript of M so that $M = (I, S, s_0, S_f, T, \delta)$.

1) I is the signature of M, denoted as a triple $I = (E, L, O)$, where E, L, O are pair-wise disjoint and represent sets of input, internal, and output events, respectively. Let $Msg = (L \cup E \cup O)$ be the set of events, we assume that L is the disjoint union of a set L_{in} of internal input events and a set L_{out} of internal output events, and the elements of $(E \cup O)$ will be called external events.

2) S is a set of states, $s_0 \in S$ is the initial state, $S_f \subseteq S$ is a set of final states.
3) $T \subseteq (EX \cup \{\Omega\}) \times BX \times (\wp(AX \cup O \cup L_{out}) \cup \{\Omega\})$ is a set of transitions, where:
   - EX is the set of Boolean expressions over input event sets $E \cup L_{in}$, linked by logical operators AND, OR, and NOT, denoted as $\wedge, \vee$, and $\neg$ respectively. Let V be a countable infinite set of variables of M. AX is the set of assignments over V. BX is the set of Boolean expressions over V.
   - For each transition $t = (ex, g, a) \in T$ (graphically denoted as $ex[g]/a$), $ex \in EX \cup \{\Omega\}$ is the input event expression, $g \in BX$ is the guard predicate, and $a \subseteq \wp(AX \cup O \cup L_{out}) \cup \{\Omega\}$ is the action set composed of assignments and output events. Ω indicates the omission of an input event expression or an output event. The components of transition t are denoted as $t.ex = ex, t.g = g, t.a = a$.
   - If there exist two statements $st_1, st_2 \in t.a \cap AX$ where $def(st_1) = def(st_2)$, then $st_1 \equiv st_2$ (see the definition of def below).
4) $\delta \subseteq S \times T \times S$ is the transition relation (graphically denoted as $s \xrightarrow{t} s'$). If $s \xrightarrow{t} s'$ with $t = (ex, g, a)$, then if the machine is in state s, the $t.ex$ and $t.g$ are evaluated to true, then the machine executes the set of instructions a and change state to s'.

We use symbols $?, !, @$ as a convention in diagrams to indicate whether an event is input, output, or internal event, denoted as $?e \in E, !e \in O, @e \in L$, respectively.

Let st denotes a statement that represents an input event of machine M, output event of M, assignment, or Boolean expression. First, we define three functions:

- $def : (AX \cup E_M) \rightarrow \wp(V)$, where $def(st) \subseteq \wp(V)$ returns the assigned variables of a statement, i.e. the variable on the left hand side of an assignment, and the input event parameters of M.
- $cuses : (AX \cup O_M) \rightarrow \wp(V)$, where $cuses(st) \subseteq V$ returns the variables on the right hand side of an assignment, and the output event parameters of M.
- $puses : BX \rightarrow \wp(V)$, where $puses(st) \subseteq V$ returns the variables in the Boolean expression over variables.

Definition 2: The **data structure** of machine M is a triple $(V_M, AX_M \cup E_M \cup O_M, BX_M)$, where AX_M, BX_M can be retrieved from the transition set T_M. $AX_M = \{st \in AX | \exists t \in T.st \in t.a\}$ and $BX_M = \{st \in BX | \exists t \in T.st \in t.g\}$. V_M is the union of $\bigcup_{st \in (AX_M \cup E_M \cup O_M)} (def(st) \cup cuses(st))$ and $\bigcup_{st \in BX_M} puses(st)$.

C. Modelling BPEL Control Dependencies

Since a WSA has no hierarchy, we simulate the hierarchical control dependencies of BPEL activities by the parent and child relationships between machines. A machine M_A is a parent of machine M_B if M_A sends a start message to M_B. A child machine can optionally send a done message to its parent machine when reaching one of its final states. Each machine has $0..1$ parent machines, and $0..*$ child machines. Since the BPEL basic activity is atomic and a BPEL structured activity is hierarchical, the machine for a BPEL basic activity has no child, and the machine for a BPEL structured activity has $0..*$ children. Fig 2 shows the machine relationships of the *approval* BPEL process. The dark arrows denote the *start* messages sent from parents to children.

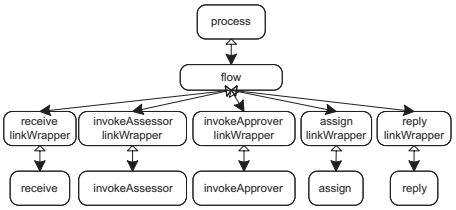

Figure 2. An example of machine hierarchy

Common Machine Layout. With consideration of fault and interruption, the machines for BPEL structured activities have a common layout, shown in Fig 3. A machine can have one or more child machines. Each machine has a *stopStatus* as a local variable. Suppose M is a machine, we can derive three scenarios from this common layout: 1) when M receives a fault from its children and no stop message arrives, it forwards the fault to its parent ($t_{i.3}$), and the 2) scenario is followed; 2) when M is interrupted by receiving a stop message, it propagates the message to its children ($t_{i.0}$) and updates the stopStatus to true. Upon receiving the child machines' done messages, if the current stopStatus is true, then the machine enters an abnormal final state($t_{i.1}$); 3) when M receives the children's done messages and no fault or stop message arrives, it ends normally ($t_{i.2}$).

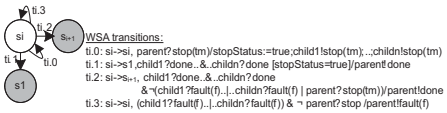

Figure 3. The common machine layout

It can be observed that the priority of the incoming events from high to low is: stop, fault, and done messages. In transition $t_{i.2}$, we do not use the predicate $stopStatus! = true$ to guard the transition. In the case that both done and stop messages have arrived, the stopStatus is updated to true only when the stop message is consumed, so the predicte $stopStatus! = true$ has no impact on the selection of consuming done or stop message. Without using the priority constraints on events, the machine only consumes one of them randomly. So, we introduce multiple-input events to the transition.

In the following, we show that a machine's input events with logical AND, OR, NOT can capture various BPEL features.

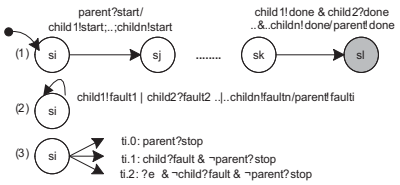

Figure 4. Multiple-input events

Concurrency. BPEL flow, scope, and eventHandler activities allow the enclosed activities to perform concurrently. We use flow activity as an illustration here. When the flow enters, all the enclosed activities start. The flow ends when all the enclosed activities end. We model this by two transitions, shown in (1) Fig 4. On the left of (1), the flow machine starts all its children as a transition action, so that all child machines will start at the same time. On the right of (1), a logical-AND operator is added to the transition input events, so that the flow machine will not end until all its children end by sending done messages.

Fault Propagation. When a structured machine receives a fault message from its children, it forwards the fault message to its parent. Suppose the structured activity encloses more than one activity. The fault is propagated as long as one of the enclosed activities raises a fault. We model this by adding a logical-OR operator to the transition input events, shown in (2) of Fig 4. Instead of using a queue for each fault, we use one FIFO queue to store all fault messages, so the fault message sent from the activity machine to its parent depends on which child's fault comes first.

Interruption. BPEL has two kinds of Interruptions. First, when a termination message is thrown when a *terminate* activity is reached, the process machine ends abnormally, and a stop message is propagated downstream from the process machine. Second, when a fault is thrown by a *throw* activity or an *invoke* activity, the fault will be propagated upstream until it can be caught by a *scope* or *process* activity that has the faultHandlers to handle this fault. The scope or process activity will stop its normal activities before enabling the faultHandlers. The stop message is propagated downstream from the scope or process machine. When a structured activity is stopped, all its children need to be stopped first. This is modelled by propagating a stop message downstream. The priority of a stop message is captured by adding logical-AND together with logical-NOT to transition input events. A stop message has higher priority than a fault message, which in turn has higher priority than a normal message. In (3) of Fig 4, transition $t_{i.0}$ is triggered when a stop message arrives. The transition $t_{i.1}$ will be triggered when it receives a fault message from its child, and only when no stop message arrives. It indicates that a fault will not be propagated when the machine is asked to stop. The transition $t_{i.2}$ indicates that a fault or interruption message has higher priority than a normal incoming message.

Synchronisation of Activities and Dead-Path-Elimination. A set of *links* can be declared in the flow construct to express the synchronisation dependencies between activities within a flow. A link is a Boolean variable, and each link is associated with a pair of source activity and target activity. For instance, if M_A and M_B are source and target activities of a link l_1, respectively, then l_1 is M_A's outgoing link with *source* tag, and M_B's incoming link with *target* tag.

The synchronisation between source and target activities is realised by setting and getting the link value. The source activity sets the link to be true or false, and the target activity gets the link value. The target activity can start when 1) all the incoming links' values are defined by the source activities, and 2) its associated *join-condition* is satisfied, which is a user-defined logical constraint on link values. The default logical constraint is OR. If the join-condition is false, the target activity will not be executed and this effect will be propagated downstream in the flow model. This is called *Dead-Path-Elimination* in BPEL. We capture the dead-path-elimination feature by updating the related links to false, and sending the link-related data exchange messages to the target activity machines.

The *target* tag and *source* tag are standard elements of BPEL constructs, indicating every BPEL activity may or may not have incoming links and outgoing links. It would be too complicate to consider handling for each activity, so we use a supporting linkWrapper machine to handle links. When an activity has incoming or outgoing links, it will associate with a linkWrapper machine and a core machine. The linkWrapper will be the core machine's parent. When an activity has no link, it is only associated with a core machine. This separation simplifies the structure of a machine, and allows BPEL activities to share a common machine structure for link handling. Fig 5 shows the linkWrapper machine structures, which covers the cases when an activity 1) has source links but no target link, 2) has target links but no source link, 3) has target links and source links.

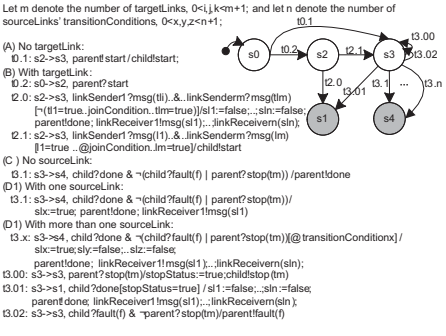

Figure 5. linkWrapper machine

Suppose M_B is the core machine and M_A is the linkWrapper machine for an activity, several scenarios can be derived from the machine structure. For case 1), when an activity has source links but no target link, two normal scenarios follow the paths $\langle t_{0.1}, t_{3.1}\rangle$, $\langle t_{0.1}, t_{3.1}, .., t_{3.n}\rangle$. The machine M_A starts by receiving a start message from its parent, and it starts the child machine M_B ($t_{0.1}$). After

M_B has finished, M_A send a done message to its parent. If the activity has one source link l_1, M_A sets the link l_1 to true and sends the link exchange message $msg(sl_1)$ to the machine of the link's target activity. If the activity has more than one source link $sl_1, ..sl_n$, for the transitionCondition of link sl_x is evaluated to true, only one link sl_x is set to true and the rest links are set to false. Thereafter, M_A sends the link messages $msg(sl_1); ..; msg(sl_n)$ to the machines of the links' target activities. For case 2), when an activity has target links but no source link, a normal scenario follows the path $\langle t_{0.2}, t_{2.1}, t_{3.1} \rangle$. M_A receives link messages $msg(tl_1), ..msg(tl_m)$ from the machines of the links' source activities. M_A waits for all the link messages to arrive and check whether the links satisfy the joinCondition. For case 3), when an activity has target links and source links, machine M_A handles the target links in the same way as the case 2), and the machine handles the source links the same as the case 1).

When all the link messages have arrived and the links do not satisfy the joinCondition, a *joinFailure* occurs and M_A ends abnormally by updating all the source links to false and sending the link messages to the machines of links' target activities ($t_{2.0}$). Alternatively, when M_A receives a fault from M_B ($t_{3.02}$), M_A propagates the fault to its parent. As long as M_A receives a stop message from its parent ($t_{3.00}$), M_A propagates the stop to M_B. After having received M_B's done message, M_A ends abnormally by setting all source links to false and sending the link messages $msg(sl_1); ..; msg(sl_n)$ to the machines of the links' target activities ($t_{3.01}$).

D. Modelling BPEL Data Dependencies

Data flow captures the relations between inputs and outputs of BPEL activities. In BPEL, variables and flow-links may affect the control flow; variables may appear in the condition expressions of *switch* and *while* activities, and may also be used in the conditions to fire particular flow-links in the source element. So taking into account variables is essential in the formal model. There are two types of variables in BPEL: BPEL variables and links. BPEL variables are declared in the variables tag of either process or scope activity. The flow-links are Boolean variables declared in the links tag of the *flow* activity. BPEL handles data by a *blackboard* approach. BPEL variables and flow-links can be used and defined by the process or scope enclosed activities, and the flow enclosed activities, respectively. We analyse BPEL activities to discover data dependencies among activities. In the following, for a message $msg(x) \in E_M \cup O_M$, msg and x denote the message name and input/output parameter, respectively.

Let $\{M_m..M_n\}$ be the set of machines selected as SUT, a message $msg(v)$ sent from machine M_1 to machine M_2, and a transition t associated with variable x, we have:

- t is annotated with $df(x)$ if a) x is defined in an assignment action of t, i.e. $\{x \in def(exp) | exp \in t.a\}$; or b) $x = v$ is the input parameter of M_2 where M_1 and M_2 are tester and SUT respectively, i.e. $\{x \in def(exp) | t \in T_{M_2} \wedge exp \in t.ex\}$, $M_2 \in \{M_m..M_n\}$, $M_1 \notin \{M_m..M_n\}$.
- t is annotated with $us(x)$ if a) x is used in an assignment action or guard of t, i.e.$\{x \in cuses(exp) | exp \in t.a\}$ or $\{x \in puses(exp) | exp \in t.g\}$; or b) $x = v$ is the output parameter of M_1 where M_1 and M_2 are tester and SUT respectively, i.e. $\{x \in cuses(exp) | t \in T_{M_1} \wedge exp \in t.a \cap O_{M_1}\}$, $M_2 \in \{M_m..M_n\}$, $M_1 \notin \{M_m..M_n\}$.

The BPEL data dependencies are captured by three data exchange models: (1) An *internal-data-exchange model* is used for a single BPEL process to specify the relation between inputs and outputs of BPEL activities. An *external-data-exchange model* is used to capture how messages are transferred from one BPEL process to other BPEL processes. When a single BPEL process is selected as SUT, an internal-data-exchange model is enough to capture the BPEL data semantics. When multiple BPEL processes are selected as SUT, a *global-data-exchange model* which is the union of the internal and external data exchange models is required to capture the BPEL data semantics. The internal-data-exchange model is captured by deriving a set of machine sequences for each BPEL variables and flow-links. The external data exchange model can be easily constructed from BPEL activities by identifying which partnerLink a message is sent to or received from. The detailed analysis of BPEL data dependencies can be found in [3]. Fig 6 shows the global-data-exchange model of the loan approval example.

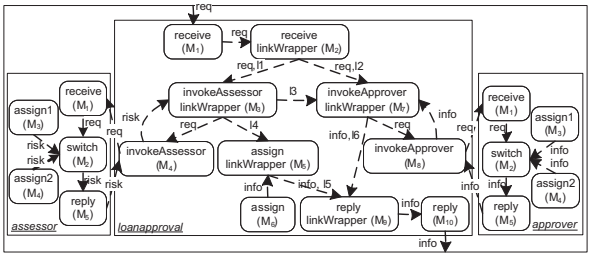

Figure 6. Global data exchange model

E. Model Checking in Testing

The idea behind model checking is to check whether a given model satisfies a given property, by exploring all possible alternatives of the given model. There are two inputs to a model checker: the model, based on a finite state machine; and, the property, expressed as a temporal logical formula. When the given property is not satisfied, the model checker outputs a set of counterexamples. A *counterexample* is an execution path that will take the finite state model from its initial state to a state where the violation occurs.

The SPIN model checker supports LTL temporal logic, and the NuSMV model checker supports both LTL and CTL temporal logic. LTL (Linear Time Temporal Logic) views time as a sequence of states with no choice as

to which state is next. The choice of next state is deterministic. CTL (Computation Tree Logic) views time as branching, so from a given branch alternative states may be reached. In our framework, we will use the LTL temporal operators [] (always), $\diamond$ (eventually), X (next), and U (until); and the CTL temporal operators A (for all paths), E (there exists some path), G (always), F (finally), X (next), and U (until).

The attraction of applying model checking in testing is that model checking can automatically produce counterexamples, which can be the basis for test cases. Proposals for applying model checking in coverage-based testing were made by [4], [5], [6]. The idea is to use a model checker to find test cases by formulating test purposes as *trap properties*. An example of a test purpose is 'a state is reachable'. A *trap property* is the negation of the original desired property, such that counterexamples can be generated for a non-error test model. The test case generation process is summarised as three steps: 1) a given test purpose is encoded into a trap property; 2) the model checker checks the given model against the trap property, and generates counterexamples; 3) test cases can be retrieved from the counterexamples. Test coverage can be achieved, by repeating such process for each test purpose for the given model. For instance, suppose the test criterion is state coverage for a machine M with states s_1, s_2, s_3, s_4, this requires four test purposes where each corresponds to 'state s_i is reachable'. By encoding the test purposes into trap properties, the model checker will search for counterexamples for each test purpose.

III. Test Case Generation Framework

In this section, we will elaborate our proposed test framework for BPEL test case generation. First, we will introduce how to simplify WSA for the purpose of improving the performance of model checking. Second, we will briefly describe the mapping from WSA to Promela and from WSA to SMV. Afterwards, we will present how to apply model checking in test case generation.

A. Overview

Fig 7 gives an overview of the test framework. The user selects one or more verified BPEL models as the SUT, picks a pre-defined test coverage criterion, and chooses a model checker. Inside the framework, the selected test coverage criterion is encoded into a set of trap properties. The BPEL processes are analysed and transformed into our proposed WSA, which in turn are transformed into Promela or SMV models (Promela and SMV are the input languages of the SPIN and NuSMV model checkers, respectively). In order to tackle the state space explosion problem of model checking and to speed up the checking performance, some model simplification techniques will be introduced for WSA. After the model transformations, the model is model-checked against the trap properties (see section III-D). A set of counterexamples will be generated. The transition IDs can be retrieved from the counterexamples. By the transition IDs, we can get the inputs, guards, and outputs of the corresponding transitions from WSA. Also, the message types of the inputs and outputs can be extracted from the WSDL interface. As a consequence, the test framework will produce BPEL based test cases that enable the user to input test data. After executing the test cases, the user can verify whether the responses from the BPEL model are operating as expected. The test cases can check whether the composed web service conforms to functional requirements.

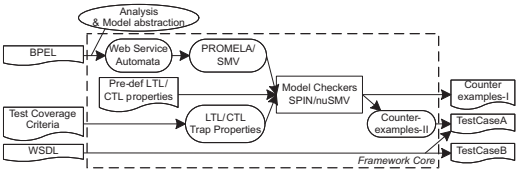

Figure 7. Framework architecture

B. Model Simplification in WSA

The abstraction is important for formal analysis with model checking techniques. In WSA, the complex data type and the concrete data value of BPEL variables will be abstracted; also the BPEL predicates will be abstracted. The abstraction will not hinder the model checking but will help to speed up the model checking. To further simplify the model, those redundant transitions and states related to faults and interruptions will be removed. The model reduction can alleviate the state explosion problem inherent of model checking techniques.

Abstraction of BPEL Predicates. We introduce a symbolic predicate for those decision points where the no concrete value of message is available by the time of analysis. From the point of view of model verification and testing, such a decision point could be considered a symbolic predicate which may equally be true or false. For example, in the *approval* BPEL process, the receive activity has two guarded outgoing links, where the guards are $request.amount < 10000$ and $request.amount \geq 10000$. Here $request$ is a BPEL variable, which is assigned a value by a message from an external service. Since the actual value of $request$ is not determined by the time of static analysis, we use symbolic predicates $pred_1, pred_2$ to abstract the actual guards. Since the values of symbolic predicates would be either true or false will equal probability, the values of $pred_1, pred_2$ can be $(1,0),(0,1),(1,1),(0,0)$ where $1,0$ denote true and false respectively. However, $pred_1, pred_2$ are not allowed to be true or false simultaneously, since the guards of outgoing flow-links of an activity should be mutually exclusive. Therefore, some logical constraints need to be added to the symbolic predicates. In the example, the logical constraint is $pred_1 \neq pred_2$.

In BPEL, a symbolic predicate will be introduced at the control decision points where the condition expressions are explicitly modelled. The symbolic predicates and logical constraints can be introduced in the following:

(1) For the *transitionConditions* of sourceLinks in an activity, a symbolic predicate is introduced for each transitionCondition. These predicates should be mutually exclusive. Suppose $p1, p2, p3$ are the predicates, the logical constraint should be $(p1 \wedge \neg(p2 \vee p3)) \vee (p2 \wedge \neg(p1 \vee p3)) \vee (p3 \wedge \neg(p1 \vee p2))$.
(2) For the *condition* of a *while* activity, two mutual exclusive symbolic predicates are introduced. Let $p1, p2$ be the predicates. The logical constraint should be $(p1 \wedge \neg p2) \vee (p2 \wedge \neg p1)$.
(3) For the condition expressions in the *case* and *otherwise* constructs of a *switch* activity, a symbolic predicate is introduced for each of them. Let pi denote a condition expression, where $1 \leq pi \leq n-1$ and $p_i = n$ are the condition expressions of *case* and *otherwise*, respectively. The logical constraint should be $(\neg p1 \wedge p2) \vee (\neg(p1 \vee p2) \wedge p3) \vee (\neg(p1 \vee p2 \vee p3) \wedge p4) .. \vee .. \neg(p_{n-3} \vee p_{n-2} \vee p_n)$.

In order to derive the logical constraints precisely for all predicates in a BPEL process, a specific tool is required.

Abstraction of BPEL Variable Types and Values. A BPEL variable can be declared as one of the three types: WSDL message type, XML Schema element, and XML Schema type. A WSDL message type and a XML element must be a complex type, while a XML Schema type can be either a simple or complex type. A complex type is a type with enclosed elements. If the variable is a WSDL message type, we need to search the WSDL model for the declaration of such a message type, so that such a variable can be declared as a complex type in Promela or SMV. In SMV modules, for a variable with a complex type in an assignment expression or in a message sending, all the enclosed elements need to be parsed and handled one by one. Since such additional modelling heavily complicates the Promela and SMV models and slows down the model checking, we abstract the complex types into abstract types without enclosed elements. An abstract type corresponds to the BPEL variable's type name itself. For instance, if a BPEL variable is declared as a WSDL message type *creditInformationMessage*, the corresponding abstract type is simply named *creditInformationMessage*. So the WSDL model does not need to include data type definitions in the mappings from BPEL-to-WSA, WSA-to-PROMELA, or WSA-to-SMV.

Because the data type is abstracted, the data value also needs to be abstracted. For instance, a BPEL variable *request* has WSDL message type *creditInformationMessage*, which includes an *amount* part. There may exist *request.amount* in the model, we abstract it into the BPEL variable itself *request*. In the case that a BPEL variable is assigned a value, we abstract the actual data value into *defined*. This abstraction will influence the predicate evaluation. However, with the above symbolic predicates, the abstraction of data values has no side-effect on the model checking.

Removal of Fault and Stop Propagations. The machines with consideration of fault and interruptions are significantly larger than the ones with only normal scenarios. If no fault can be thrown in a BPEL process, any fault propagation related transitions and states of compound machines can be left out. Furthermore, if no fault is thrown and no *terminate* activity exists in a BPEL process, the transitions and states related either to fault or stop propagation can be left out. In this way, the model size can be drastically reduced.

C. Model Transformation

From BPEL to formal model WSA. BPEL has complex features such as hierarchy, interruption, concurrency, synchronization, scoping, compensation, fault handling, and multi-threads handling. Each BPEL activity is mapped to a WSA, or a core machine and a machine for link handling. A BPEL structural machine can start and stop its enclosed machines. The Boolean expression over multiple input events of machine transitions can capture various BPEL features. The logical-AND operator can capture the synchronization of *end*. BPEL data flow is analyzed explicitly, so that interactions of BPEL activities can be modelled by message passing. Details of how WSA model various BPEL features can be found in [7]. It is essential to provide an intermediate model between BPEL and model checkers. Without such a layer, every model checker needs to consider how to model BPEL features in its input language, which complicates the process. Instead, since BPEL features have been modelled in WSA, which is a Mealy-machine based model without hierarchy, WSA can be easily transformed to the automata-based input models of most model checkers.

From WSA to Promela or SMV models. Promela is the input language of the SPIN model checker. A Promela model consists of a set of processes and channels for process communication. The states, transition IDs, and local variables of a WSA are captured in the process's variable declaration part. The transition relations are captured in the process's behavioural modelling part, enclosed within a *do* loop. Since Promela supports message communication via channels, it is straightforward to transform WSA to Promela. SMV is the input language of NuSMV model checker. A SMV model is composed of a set of modules. The states, transition IDs, transition input events, transition output events, and local variables of a WSA are declared in the module's VAR section. The transition relations are captured in a module's ASSIGN section. Since the SMV language has no support for channels, the input queues of WSA need to be modelled explicitly. We model a queue for each message type as a SMV module, and the actual input queues are instantiated in the SMV module corresponding to a WSA. We implemented a queue structure that supports FIFO manipulation, but the state space increases dramatically with such models. In order to reduce the state space, a simple queue model holding only one message is used instead.

D. BPEL Test Case Generation

We apply the structural test coverage criteria to multiple machines. State and transition coverages are used for

BPEL control flow testing, and all-du-path coverage is used for BPEL data flow testing. The variables and flow-links declared in BPEL models will be considered in data flow testing. We are interested in testing the whole BPEL model. According to the machine hierarchy of the previous section, a test case should start and end with the BPEL *process* machine M_{proc}. Following the propagation of the *start* and *done* messages between parent machines and child machines, we may assume without loss of generality that in the machine hierarchical graph, every machine is reachable from the BPEL *process* machine, and that the BPEL *process* machine is reachable from every machine.

In the following definitions, let a BPEL process P associated with a set of machines $\{M_1, .., M_n, M_{proc}\}$.

Definition 3: A **test case** (or test path) of a BPEL process P starts from the initial state of M_{proc}, and ends at one of the final states of M_{proc}.

Definition 4: A test suite covers a state s if there is at least one test case in the test suite that executes s. A test suite is said to achieve **state coverage** of a BPEL process P if it covers each state $s \in S_{M_i}.M_i \in \{M_1, .., M_n, M_{proc}\}$.

Definition 5: A test suite covers a transition t if there is at least one test case in the test suite that executes t. A test suite is said to achieve **transition coverage** of a BPEL process P if it covers each transition $t \in T_{M_i}.M_i \in \{M_1, .., M_n, M_{proc}\}$.

Data flow testing is interesting because it stimulates the sequences of operations which define and subsequently use variable values. It is an effective systematic method for exposing faults. For the *du-path coverage*, we adopt the definition from [8]. Let x be a variable. A du-pair of x is a transition pair (t_i, t_j) where t_i is with $df(x)$ and t_j is with $us(x)$. A *def-clear path* with respect to x is a transition sequence $\langle t_1, t_2, .., t_n \rangle$ where there is no $df(x)$ in any of the transitions $t_2, t_3.., t_{n-1}$. A *data flow* (or *du-path*) of x is a transition sequence such that (t_i, t_j) is a du-pair and there is a def-clear path from t_i to t_j for x.

Note that we are only interested in the BPEL variables explicitly declared in a BPEL model (denoted as V_{bpel}), and the data dependency between BPEL activities. Therefore, in our all-du-path coverage criterion, we only consider du-pairs $\{(t_i, t_j) | t_i \in M_i, t_j \in M_j, i \neq j\}$ with respect to v where $v \in V_{bpel}$.

Definition 6: A test suite covers a du-path ph of a variable $v \in V_{bpel}$ if there is at least one test case in the test suite that executes ph. A test suite is said to achieve **all-du-path coverage** of a BPEL process P if it covers each du-path of each $v \in V_{bpel}$.

Test Coverage Criteria in Trap Properties. Now we can encode the test coverage criteria into CTL and LTL temporal logic. [4] gives a detailed study of encoding various structural test coverage criteria into CTL. Based on their work, the negation of state, transition, and all-du-path coverage criteria are encoded into the CTL formulas as follows. Here M is a web service automaton.

- $\{\neg EF(s_i \wedge EF\ s_f)\}$ where $s_i \in S_M, s_f \in S_{fMproc}$.
- $\{\neg EF(t_i \wedge EF\ s_f)\}$ where $t_i \in T_M, s_f \in S_{fMproc}$.
- $\{\neg EF(t_i \wedge EX\ E\,[\neg d(v) U (t_j \wedge EF\ s_f)])\}$ where $v \in V_M, t_i \in d(v), t_j \in u(v), s_f \in S_{fMproc}$.

Suppose M is a web service automaton. The negation of state, transition, and all-du-path coverage criteria are encoded into the LTL formulas as follows.

- $\{\neg\Diamond(s_i \wedge \Diamond s_f)\}$ where $s_i \in S_M, s_f \in S_{fMproc}$.
- $\{\neg\Diamond(t_i \wedge \Diamond s_f)\}$ where $t_i \in T_M, s_f \in S_{fMproc}$.
- $\{\neg\Diamond(t_i \wedge X(\neg d(v) U t_j) \wedge \Diamond s_f)\}$ where $v \in V_M$, $t_i \in d(v), t_j \in u(v), s_f \in S_{fMproc}$.

In SPIN model checker, each LTL formula needs to be converted into a *Buchi Automaton* enclosed in a *never claim*. Since a never claim is to negate the enclosed Buchi automata, the input LTL formula for SPIN model checker should be the original property (the non-negated one). In Promela, we attach a never claim, corresponding to the user selected test coverage criterion, to the Promela model generated from WSA, and use *#define* to declare the elements required in the never claim. Fig 8 (a) shows a never claim for covering a state and a final state of the process machine. $\Diamond(p \wedge \Diamond q)$ is the LTL formula for the enclosed Buchi Automaton. p denotes a state of a machine and q denotes a final state of the process machine. According to the above LTL state coverage, a set of the negations of such LTL formulas can provide state coverage of the BPEL model. Since SPIN can only verify one property in a run, SPIN needs to run n times for n pairs of (p, q).

NuSMV model checker accepts either LTL or CTL formulas, and a formula starts with the keyword *SPEC* in SMV models. Fig 8 (b) shows a CTL formula declaration for covering a state and a final state of the process machine. According to the above CTL state coverage, a set of such CTL formulas can provide state coverage of the BPEL model. Since NuSMV can verify more than a single property in a run, SMV only needs to run once for a set of CTL formulas.

```
    #define p (loanapproval_flow_receive1:state == s2)
    #define q (loanapproval:state == s3 || loanapproval:state == s1)
(a) never {    /* <>(p && q) */
    T0_init:
        if
        :: ((p) && (q)) -> goto accept_all
        :: (1) -> goto T0_init
        fi;
    accept_all:
        skip
    }
(b) SPEC !EF(loanapproval_flow_receive1.state = s2
    & EF loanapproval.state = s3 | loanapproval.state=s1)
```

Figure 8. An example of state coverage

Symbolic Test Case Generation. In a state machine with symbolic predicates (see section III-B), in order to enforce a model checker to explore alternative paths, the predicates of different paths need to be true alternatively. This requires the values of symbolic predicates to be equally true or false. We apply *Gray codes* (e.g. [9]) to compute the combinations of predicates, where two successive values differ in only one digit. With all the combinations, the model checkers can explore all the paths of a model. For two symbolic

predicates $pred_1, pred_2$, the two-bit Gray code matrix is $(pred_1, pred_2) : (0,0),(0,1),(1,1),(1,0)$. Here $1,0$ denote Boolean true and false, respectively. After all the possible combinations of predicates have been calculated, we can apply the logical constraints introduced in section III-B to remove illegal combinations. In the case that $pred_1, pred_2$ are mutually exclusive, the combinations $(0,0),(1,1)$ can be removed.

The corresponding Promela code is shown below on the left of Fig 9. First, each symbolic predicate $pred_i$ is declared as a global Boolean variable. Second, a Gray code matrix is constructed based on the declared symbolic predicates, the logical constraints on the predicates, if any, can be applied to the Gray code matrix. Third, two processes will be inserted into the Promela model: a *runner* process that assigns predicate values based on the above Gray code matrix, and a *chooser* process that chooses the current combination of predicates. The *chooser* process will be started by the *init* process.

```
bool pred1
bool pred2
/* Predicate Relationships
pred1 != pred2  */
proctype runner(byte type) {
 if
 :: type == 1-> atomic {pred1 = 1 ; pred2 = 0 ; }
 :: type == 2-> atomic {pred1 = 0 ; pred2 = 1 ; }
}
proctype chooser() {
  run runner(1) ;
  run runner(2) ;
}
```

```
--Predicate Relationships
--pred1 != pred2
MODULE runner
VAR
 pred1 : boolean;
 pred2 : boolean;
 i : 1..2;
ASSIGN
 next(pred1) := case
                 i = 1          : 1;
                 i = 2          : 0;    esac;
 next(pred2) := case
                 i = 1          : 0;
                 i = 2          : 1;    esac;
 next(i) := case
                 i = 2          : 1;
                 1              : i + 1; esac;
FAIRNESS running
```

Figure 9. An example of predicate handling

Similarly, the SMV code is shown on the right of Fig 9. A *runner* module declares the symbolic predicates. After the Gray code matrix has been constructed based on the declared symbolic predicates. The logical constraints on the predicates, if any, can be applied to the Gray code matrix. Thereafter, the *ASSIGN* section will be inserted into the runner module, so that the runner can choose the current combination of predicates. The *runner* process will be started by the *main* module.

From Counterexamples to Test Cases. A test case consists of a set of execution paths of the BPEL. The BPEL test generation is a kind of white-box testing that analyses the internal process behaviour. The transition IDs are modelled explicitly in Promela and SMV models, so that a transition ID list can be retrieved from the generated counterexample. A test case can be derived from the transition ID list, by extracting the corresponding transition input events, guards, actions, and output events from the associated WSA model, and getting the message types of the inputs and outputs from the WSDL interface.

IV. Case Study and Tool Support

In this section, we will use the loan approval service as case study to illustrate the modelling of BPEL processes in web service automata, and to evaluate the effectiveness of our test framework. For simplicity, we omit the fault-Handling feature in the example. Then, we will provide a brief description of the tool support.

A. Case Study

Single BPEL Process as SUT. When the *approval* BPEL process is selected as SUT, the customer, assessor, and approver become testers. The graphical view of the internal data exchange model for the approval process is shown on the middle of Fig 6. The machine sequences of the internal-data-exchange model can be derived as follows:

- For variable req, the machine du-pairs are $(M_1, M_4),(M_1, M_8)$, and the machine sequences are $\langle M_1, M_2, M_3, M_4\rangle$, the machine sequence is: $\langle M_1, M_2, M_7, M_8\rangle$.
- For variable $risk$, the machine du-pair are (M_4, M_3), and the machine sequence is $\langle M_4, M_3\rangle$.
- For variable app, the machine sequences are: $\langle M_8, M_7, M_9, M_{10}\rangle$, $\langle M_6, M_5, M_9, M_{10}\rangle$.
- For links $l_1, l_2, l_3, l_4, l_5, l_6$, the machine du-pairs are the same as the machine sequences, which are $\langle M_2, M_3\rangle, \langle M_2, M_7\rangle, \langle M_3, M_7\rangle$, $\langle M_3, M_5\rangle$,$\langle M_5, M_9\rangle$, $\langle M_7, M_9\rangle$, respectively.

According to the du-pairs of BPEL variables and links, the assertions for BPEL variables flow links will be added to transitions where the variables are used.

The machines for the approval process is shown in Fig 10. In the diagram, we use LW as short for linkWrapper. In the *receive, invokeAssessor*, *invokeApprover*, and *reply* machines, when it interacts with a machine of the external BPEL process and that process is not a part of the SUT, then the partner machine is simply named $tester$. In WSA machines, four symbolic predicates are introduced: the receive linkWrapper has predicates $pred_1, pred_2$ where $pred_1 \neq pred_2$, and the invokeAssessor linkWrapper has predicates $pred_3, pred_4$ where $pred_3 \neq pred_4$.

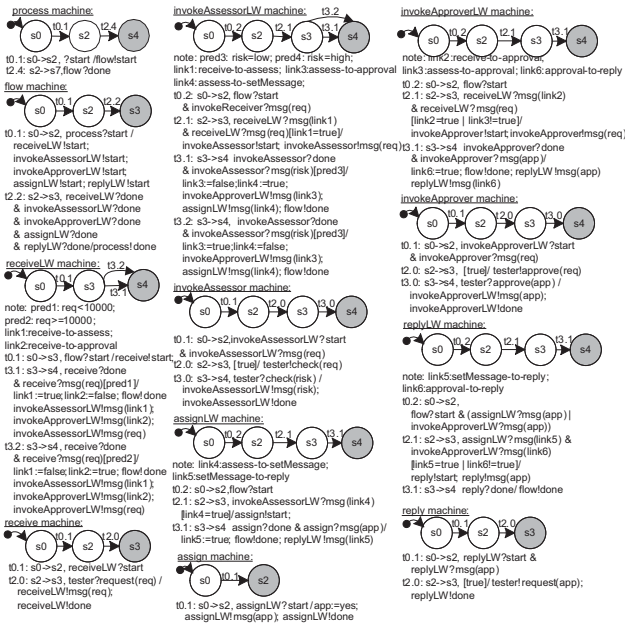

Figure 10. The machines for the approval BPEL model

When the *assessor* process is chosen as SUT, the graphical view of the internal data exchange model for the approval process is shown on the left of Fig 6. For req: $\langle M_1, M_2\rangle$. For $risk$: $\langle M_3, M_2, M_5\rangle$ and $\langle M_4, M_2, M_5\rangle$. According to the internal-data-exchange machine sequences and the mapping rules of BPEL to WSA, the *assessor* process can be transformed into the state machines in Fig 11. Similarly, the machine can be derived in the same way when the *approver* process is chosen as SUT. Since all the BPEL features are analysed and captured in WSA, the transformation from WSA to Promela or SMV are straight away.

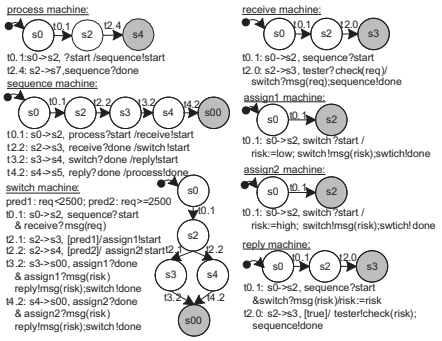

Figure 11. The machines for the assessor or approver BPEL model

Next, we can enter the stage of test case generation. Fig 12 shows three scenarios when the approval, assessor, and approver are selected as SUT respectively. The edge with arrow from a tester to the SUT denotes a test input to the SUT, and the edge with arrow from the SUT to a tester denotes a test output from SUT. The test framework can generate a set of test cases based on the selected SUT, where each test case corresponds to an execution scenario of a BPEL process. The message types of the test inputs and test outputs of a SUT are based on the message type declared in WSDL. In a test case, each test output from SUT is asserted to be not null, so that a test case fails if not an assertion is violated.

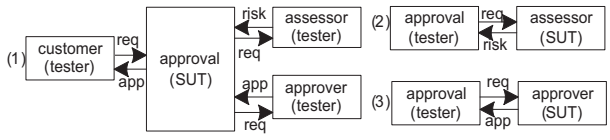

Figure 12. A single BPEL process as SUT

Since a test case covers a sequence of BPEL activities, in the following, we use a sequence of BPEL activities to illustrate a test case. First, when the *approval* BPEL process is selected as SUT, the test inputs to the SUT would be $req, risk, app$, and the test outputs from the SUT would be req, app. When choosing a control-flow based test coverage criterion, three test cases will be generated:

- $tc1$: <*receive, invokeAssessor, assign, reply*>. The alllowed data ranges for test inputs $req, risk$ are $req < 1000, risk = low$, respectively.
- $tc2$: <*receive, invokeAssessor, invokeApprover, reply*>. The allowed data ranges are $req < 10000$, $risk = high$, and $app = yes|app = no$.
- $tc3$: <*receive, invokeApprover, reply*>. The allowed data ranges are $req >= 10000$ and $app = yes|app = no$.

For example, the test input and output sequence of test case $tc1$ is: 1) the tester customer inputs a req to the approval, 2) the approval outputs the req to the tester assessor, 3) the tester assessor inputs a $risk$ to the approval, 4) the approval outputs an app to the customer tester.

When choosing the du-path test coverage criterion for BPEL variables, the paths for each BPEL variable are as follows. For req: <*receive, invokeAssessor, assign, reply*>,<*receive, invokeAssessor, invokeApprover, reply*>, and <*receive, invokeApprover, reply*>; For $risk$: <*receive, invokeAssessor, assign, reply*> and <*receive, invokeAssessor, invokeApprover, reply*>; For app: <*receive, invokeAssessor, invokeApprover, reply*> and <*receive, invokeApprover, reply*>. After merging these paths, the generated test cases are still the $tc1, tc2, tc3$ as above.

Multiple BPEL Processes as SUT. Since the internal-data-exchange model of a BPEL process is fixed, when more than one BPEL process is selected as SUT, the state machines associated with approval, assessor, and approver services are similar to the ones in Fig 10 and Fig 11. The difference is that if an activity A interacts with an external service's activity B and the external service is a part of SUT, then in A its partner machine is named B (instead of naming it as $tester$). For instance, if approval and approver services are selected as SUT, then the *invokeApprover* machine of approval service will interact with the *receive* and *reply* machines of the approver service. In the testing phase, Fig 13 shows an example when the approval, assessor, and approver are selected as SUT.

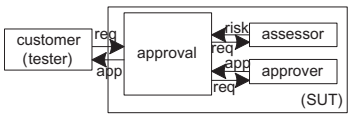

Figure 13. Multiple BPEL processes as SUT

There exist a test input req to SUT and a test output app from SUT. The interactions between approval, assessor, and approver are internal and not observable by the tester. Let S, L, A denote the shorthands for assessor, approval, and approver services. When choosing a control-flow based test coverage criterion, five test cases will be generated:

- $tc1$: <*L.receive, L.invokeAssessor, S.receive, S.switch, S.assign1, S.reply, L.invokeAssessor, L.assign, L.reply*>.
- $tc2$: <*L.receive, L.invokeAssessor, S.receive, S.switch, S.assign2, S.reply, L.invokeAssessor, L.invokeApprover, A.receive, A.switch, A.assign1, A.reply, L.invokeApprover, L.reply*>.

- $tc3$: *<L.receive, L.invokeAssessor, S.receive, S.switch, S.assign2, S.reply, L.invokeAssessor, L.invokeApprover, A.receive, A.switch, A.assign2, A.reply, L.invokeApprover, L.reply>*.
- $tc4$: *<L.receive, L.invokeApprover, A.receive, A.switch, A.assign1, A.reply, L.invokeApprover, L.reply>*.
- $tc5$: *<L.receive, L.invokeApprover, A.receive, A.switch, A.assign2, A.reply, L.invokeApprover, L.reply>*.

When choosing the du-path test coverage criterion for BPEL variables, the req is defined by the tester and userd by *S.switch*; the $risk$ is defined by $S.assign1$ or $S.assign3$, and used in $L.invokeAssessor$; the app is defined by $L.assign, A.assign1$ or $A.assign2$, and it is used by $L.reply$. After merging the test pathes for $req, risk, app$, the generated test cases are the same as above $tc1, tc2, tc3, tc4, tc5$.

Fig 14 shows the summary when choosing different BPEL processes as SUT.

BPEL	Loan approval service				
SUTs	Approval	Assessor/ Approver	Approval, Assessor	Approval, Approver	Approval, Assessor, Approver
State Space	203	91	296	367	473
Predicates (combination)	4 (4)	2 (2)	6 (8)	6 (8)	8 (16)
Def-Use Pairs	11	3	14	14	17
Num. of test paths	3	2	3	4	3

Figure 14. Multiple BPEL processes as SUT

Since state space explosion is the main issue of model checking techniques, in order to improve the performance of a model-checking based tool, it is essential to reduce the model state space. We have shown by case studies that our web service automaton can alleviate the state space explosion problem by introducing multiple input and output events to machine transitions. Also, our abstraction of message type, message value, and predicates helps to speed up the model checking.

B. Tool Support

Our test framework has been implemented as an Eclipse plugin, the *BPEL test case wizard*, which is a component of the DBEStudio for the EU project DBE [10]. The BPEL test case wizard consists of three components: the user interface, the transformation engine, and the model checker manager. The wizard allows the user to choose a BPEL process as the service under test (SUT), and choose the WSDL interface model. The partner web services of the selected BPEL process can be chosen as a component of the SUT. Then a model checker can be selected to verify the general properties and to generate test cases. Inside the test framework, the XSLT transformation engine manages the mappings between models, which is using Saxon's XSLT 2.0 processor. For pre-processing, the engine manages the mappings for the model checkers, including BPEL-to-WSA, WSA-to-PROMELA, WSA-to-SMV. For post-processing, the engine manages the mapping from counterexamples to transition sequences, and the mapping from transition sequences to test cases. The mapping to test cases is based on the transition sequence, the web service automata, and the WSDL description. The test cases are in JUnit test case format. Hence, the test cases can be run against the JUnit execution engine directly. The user allows entering the test input (data) via a user interface.

V. Related Work

The current formal semantics proposed for BPEL can be categorised under three branches: Petri-net, Process Algebra, and Automata.

A. Petri Nets based approaches

Web service algebra is proposed in [11] to define a set of web service composition operators. The authors use Petri nets as the formal semantics for the proposed web service algebra. The works of [12], [13] present Petri net semantics for the control flow of BPEL, with consideration of BPEL advanced features such as fault handling, event handling, and compensation handling. In [13], the tool BPEL2PN is developed to map BPEL code to Petri nets, and model checker LoLA is used to verify CTL temporal logic. The author of [14] extends the work of [13] by using Petri net to capture the global interactions between BPEL processes. In [12], the tool BPEL2PNML is developed to map BPEL to Petri nets, and a verification tool WofBPEL is used as the analysis engine. The paper discusses how to verify the activity reachability and some pre-defined BPEL constraints. As a summary, the above works abstract from data. As shown in our motivation example, it is important to consider BPEL data dependencies.

In [15], they claim to capture both BPEL control and data dependencies in CP-nets, and CPN tools can be used to verify the process. However, the paper only shows how to map a core subset of BPEL to CP-nets. There is no discussion of how to capture BPEL data dependencies, and no concern of modelling faults or compensations.

In [16], they use CP-nets as the process composition models, and apply CPN tools as the verification engine. BPEL skeleton code can be generated from the process composition model. In the CP-nets models, messages (events) and process variables are represented by tokens. Abstract colour sets are declared for the messages and variables such that each colour set is kept small to speed up the analysis. They also use an algorithm to automatically derive the conversation protocol, which is also CP-net based, from the process composition models. The conversation protocol in their context only models the interactions between the service consumer and the service provider, and hides the internal process details such as those providing data manipulation and interaction with other service partners. Instead of verifying the BPEL process, their work focuses on designing a correct CP-net based model and generating a BPEL skeleton process.

B. Process Algebras based approaches

Process algebraic service composition aims to introduce much simpler descriptions than other approaches. In [17], a two-way mapping is defined between BPEL and LOTOS, and the model checking toolbox CADP is used as the verification engine. The mapping from LOTOS to BPEL does not preserve the structure of a process, due to the expressive and flexible structure of LOTOS. The *disabling operator* is used to capture the BPEL interruptions. In LOTOS, the processes communicate synchronously by rendezvous.

In [18], the process algebra FSP (Finite State Processes) developed by [19] is used for the BPEL semantics and the model checker LTSA [19] as the verification engine. The web service composition specification is modelled in an MSC (Message Sequence Chart), and the implementation is modelled in BPEL. Both MSC models and BPEL processes are translated into FSPs, such that the BPEL implementation can be verified against the MSC specification by trace equivalence checking. The work of [20] extends the earlier work to verify the interacting BPEL processes and checks their compatibility. A tool LTSA-WS was implemented as an Eclipse plug-in. FSP is abstract from data, so their mapping does not cover BPEL data dependencies. Also, FSP supports synchronous communication.

In [21], they use Pi-calculus as the BPEL formal model and NuSMV model checker as the verification engine. A tool, OPAL, is developed to automate the mapping from BPEL to Pi-calculus, and from Pi-calculus the input language SMV of the NuSMV model checker. It points out that there exist two approaches to model check Pi-calculus. One is to analyse Pi-calculus processes based on a proof system, and the other is to transform Pi-calculus into automata. The authors follow the second approach.

In [22], a language named CDL is proposed to extend WSDL to model the behaviour of individual web services. A composition language is also proposed, which can support both centralised and distributed orchestrations. The formal semantics of these two languages are based on Pi-calculus. They point out that the use of shared variables in BPEL makes it difficult to coordinate the execution in a distributed manner. Their composition language has two core concepts: a task and a process. A task is equal to a BPEL activity. For inter-task dependency, they explicitly consider control dependencies and data dependencies. They further point out that the current tool support for verification of Pi-calculus is immature. Most do not support the complete language and require a complex and error prone input syntax. A solution is to map Pi-calculus to the input languages of mature model checkers such as SPIN. Since the input languages of most mature model checkers are automaton-based, we believe an automaton-based formal model is more suitable for those input languages.

In the above Process Algebra approaches, they consider both the core BPEL activities and the advanced BPEL features with fault handling, compensation handling, and event handling. However, since the BPEL scope based fault and compensation handling mechanism is complex, only a few works give it in-depth analysis. In [23], Pu *et al.* propose a BPEL0 language to capture the BPEL scope based compensation handling features. They propose a n-bisimulation relation, which reflects the scope-based compensation mechanism, to define the equivalence between BPEL0 programs. An execution engine for BPEL0 is developed. In [24], Bulter *et al.* formalise the notions of compensation in a StAC language. They model the BPEL control flow using StAC and the model the BPEL data manipulation in B notation. The semantics is clean and precise, but it is not clear how to verify or test such models in an automatic way.

C. Automata based approaches

Automata provide a well-known formalism for system specifications. There are many different kinds of automata, including finite state machines (FSMs) such as Mealy and Moore machines. Label Transition systems are automata, which are generally infinite state.

In [25], BPEL models are mapped into deterministic finite state automata for the matchmaking of web service composition. The STSs (State Transition Systems) are used in [26] to be the BPEL formal semantics, and a tool is developed as a part of the ASTRO toolset [27]. Both of these formal models are abstracted from data.

In [28], they propose guarded automata (GA) to be the formal models for both BPEL and the conversation protocol. GA extends Mealy machines with data, and every transition is equipped with a guard in the form of an XPath expression. The model checker SPIN is used as the verification engine. BPEL processes communicate by sending asynchronous messages, and each process has a queue. A global watcher keeps track of all messages. The conversation is introduced as a sequence of messages. They propose a set of sufficient conditions so that asynchronous communication can be replaced with synchronous communication. A tool WSAT is developed to map BPEL to guarded automata, and map guarded automata to Promela (the input language of SPIN). In their approach, each BPEL activity is mapped to a GA, so the BPEL process as a whole is a composition of a set of GAs. The interleaving semantics of concurrency is used. However, they omit the inter activity data dependencies. Also, in their models, the GAs representing BPEL processes communicate by message passing, but it is not clear how the GAs represents the BPEL activities communications. In our formal model, we believe it is clearer from the theoretical view to provide a single communication mechanism for both external and internal interactions, where machines communicate by either message passing or by data sharing, but not both. In their mapping from GA to Promela, the XPath expressions in the GA transition guards are also translated into Promela, so that the data manipulation can be verified. We believe this will decrease the speed of model checkers, and that

symbolic transition guards can reach the same verification result.

Similar to [28], the author of [29] transforms BPEL into an extended FSM called EFA. A tool is developed to automate the mappings from BPEL to EFA, and from EFA to Promela. EFA extends a Mealy machine with data, and it adopts asynchronous interleaving semantics of concurrency. They also do not consider the interactions among BPEL activities due to data dependencies. Furthermore, it is not clear which communication mechanism is used when they are modelling read and write data by BPEL activities.

The above automata based approaches, only cover core subsets of BPEL activities and do not consider fault handling, compensation handling, or event handling.

D. Testing of BPEL based web services

As we can see from the previous section, there is intensive research on providing precise semantics for BPEL and verification of BPEL models. However, there is less effort on using BPEL as the test models for deriving test cases. A framework is proposed in [30] to augment WSDL with a UML2.0 PSM (Process State Machine) for modelling web service interactions. After transforming PSM to a Symbolic Transition System, existing *ioco-conformance* testing tools can be applied. In [31], they use Graph Transformation Rules along with WSDL to generate test cases. WSDL-S is proposed in [32] to be the service behaviour model, which extends WSDL by adding a pre-condition and post-condition to each WSDL operation. The WSDL-S is mapped to EFSM so that the existing test techniques for EFSM can be applied. Yuan et al. [33], [34] propose a XCFG (extended control flow graph) to represent a BPEL process. First, they propose a DFS (depth first search) based algorithm to generate sequential test paths from the XCFG, according to branch coverage criterion. Second, the sequential test paths will be combined into concurrent test paths based on various BPEL structures. Finally, a constraint solver is used to remove the un-executable test paths, and to generate test data. They assume that the BPEL eventHandlers can only have on message thread, and assume that there is no interruption due to either fault propagation or process termination.

VI. Conclusion and Future Work

In this paper we have defined an operational semantics for the BPEL language, and presented an automatic test generation framework for BPEL models. We addressed the research question of applying model-based-testing techniques to the domain of BPEL-based web service orchestration, so that the functionality of the design model can be systematically verified.

The web service automaton (WSA) has been proposed with two purposes: 1) to define the operational semantics for BPEL models; and, 2) to clarify the BPEL verification and test case generation problem at hand. The Boolean expression over input events is introduced for the purpose of modelling concurrency, fault propagation, and interruption features of BPEL language in a more natural way, and also as means to reducing unnecessary state space. A model checking based test framework has been presented. Since NuSMV and SPIN model checkers are already used on a regular basis for the verification of real-world applications, they are used as two alternative test engines in our framework. The advantage of using model checking for test case generation is that it is automatic, but the state space explosion problem is a well known inherent problem to model checking techniques. In order to alleviate the state explosion problem, we abstracted certain parts of the BPEL processes in WSA. The structural test coverage criteria are applied to multiple machines. The state coverage and transition coverage are used for BPEL control flow testing, and all-du-path coverage is used for BPEL data flow testing. Because we abstract the BPEL predicates into symbolic predicates, the predicate combinations will be constructed to enforce the model checker to explore all the alternative paths. The generated test cases can check the behaviour conformance of web service interactions and various business scenarios.

An open issue is to prove the correctness of the model transformation. One solution is to transform the two kinds of models into a third common formalism, and check the equivalence between the models in the common formalism. This solution needs the assumption that the further transformations are correct. Currently, we adopt the simplest inspection approach. This is a trade-off between the soundness of a formal approach and the simplicity of an informal approach. Also, given a set of BPEL processes that are known as correct, we can partially check the transformation correctness, by running the model checker to verify the general properties, such as deadlock-freeness and absence of non-instantiated data, of the target models.

Currently, we applied the conventional state and transition test coverage criteria to multiple machines. An extension of this work is to define additional test coverage criteria which are more interesting for integration testing, so that the functionality is checked by less model checking time and effort. Another interesting future work is to support on-the-fly test criteria without the need of writing temporal logic manually by the user. Furthermore, currently the framework provides a GUI for the user to input test data. The question of how to automatically generate appropriate input data as parameters in the test case remains a topic for further research.

Acknowledgment

We thank Dr. Mike Shield for discussions concerning this work, and the reviewers for their detailed comments.

References

[1] N. Milanovic and M. Malek, "Current solutions for web service composition," *IEEE Internet Computing*, vol. 08, no. 6, pp. 51–59, 2004.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, S. Thatte, and S. Weerawarana, "Business process execution language for web services version 1.1," May 2003.
[3] Y. Zheng and P. Krause, "Analysis of bpel data dependencies," in *Proc. of EUROMICRO*. IEEE Computer Society, 2007.
[4] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural, "Data flow testing as model checking," in *Proc. of ICSE*. IEEE Computer Society, 2003, pp. 232–242.
[5] M. P. E. Heimdahl, D. George, and R. Weber, "Specification test coverage adequacy criteria = specification test generation inadequacy criteria?" in *Proc. of HASE*. IEEE Computer Society, 2004, pp. 178–186.
[6] S. Rayadurgam and M. P. E. Heimdahl, "Coverage based test case generation using model checkers," in *Proc. of ECBS*. IEEE Computer Society, 2001, p. 0083.
[7] Y. Zheng, "An automatic test framework for bpel web services," in *PhD Thesis*. Dept of Computing, University of Surrey, 2007.
[8] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 367–375, 1985.
[9] "Gray code," http://en.wikipedia.org/wiki/Gray_code, 2007.
[10] "Digital business ecosystem," http://www.digital-ecosystem.org, 2007.
[11] R. Hamadi and B. Benatallah, "A petri net-based model for web service composition," in *Proc. of ADC*. Australian Computer Society, Inc., 2003, pp. 191–200.
[12] C. Ouyang, W. van der Aalst, S. Breutel, M. Dumas, A. ter Hofstede, and H. Verbeek., "Formal semantics and analysis of control flow in ws-bpel," in *BPM Center Report*. BPMcenter.org, 2005.
[13] C. Stahl, "A petri net semantics for bpel," Humboldt-Universitt zu Berlin, Informatik-Berichte 188, July 2005.
[14] N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg, "Analyzing interacting bpel processes," in *Proc. of BPM*, ser. Lecture Notes in Computer Science, vol. 4102. Springer-Verlag, 2006, pp. 17–32.
[15] Y. Yang, Q. Tan, J. Yu, and F. Liu, "Transformation bpel to cp-nets for verifying web services composition," in *Proc. of NWeSP*. IEEE Computer Society, 2005.
[16] X. Yi and K. J. Kochut, "A cp-nets-based design and verification framework for web services composition," in *Proc. of ICWS*. IEEE Computer Society, 2004, pp. 756–760.
[17] A. Ferrara, "Web services:a process algebra approach," in *Proc. of ICSOC*. ACM Press, 2004, p. 242.
[18] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Model based verification of web service compositions," in *Proc. of ASE*. IEEE Computer Society, 2003, p. 152.
[19] J. Magee and J. Kramer, *Concurrency: state models and Java programs*. New York, NY, USA: John Wiley and Sons, Inc., 1999.
[20] H. Fostre, S. Uchitel, J. Magee, and J. Kramer, "Model based analysis of obligations in web service choreography," in *Proc. of AICT-ICIW*. IEEE Computer Society, 2006, p. 149.
[21] K. Xu, Y. Liu, and G. Pu, "Formalization, verification and restructuring of bpel models with pi calculus and model checking," IBM, Tech. Rep., 2006.
[22] S. J. Woodman, D. J. Palmer, S. K. Shrivastava, and S. M. Wheater, "Notations for the specification and verification of composite web services," in *Proc. of EDOC*. IEEE Computer Society, 2004, pp. 35–46.
[23] G. Pu, H. Zhu, Z. Qiu, S. Wang, X. Zhao, and J. He, "Theoretical foundations of scope-based compensable flow language for web service." in *Proc. of FMOODS*, 2006, pp. 251–266.
[24] M. J. Butler, C. Ferreira, and M. Y. Ng, "Precise modelling of compensating business transactions and its application to bpel." *The Journal of Universal Computer Science*, vol. 11, no. 5, pp. 712–743, 2005.
[25] A. Wombacher, P. Fankhauser, and E. Neuhold, "Transforming bpel into annotated deterministic finite state automata for service discovery," in *Proc. of ICWS*. IEEE Computer Society, 2004, p. 316.
[26] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi, "Automated synthesis of composite bpel4ws web services," in *Proc. of ICWS*. IEEE Computer Society, 2005, pp. 293–301.
[27] "Astro toolset," http://www.astroproject.org/.
[28] X. Fu, T. Bultan, and J. Su, "Synchronizability of conversations among web services," *IEEE Transactions on Software Engineering*, vol. 31, no. 12, pp. 1042–1055, 2005.
[29] S. Nakajima, "Lightweight formal analysis of web service flows," National Institute of Informatics, Tech. Rep., 2005, progress in Informatics.
[30] A. Bertolino and A. Polini, "The audition framework for testing web services interoperability," in *Proc. of EUROMICRO*. IEEE Computer Society, 2005, pp. 134–142.
[31] R. Heckel and L. Mariani, "Automatic conformance testing of web services," in *Proc. of FASE*. Springer, 2005, pp. 34–48.
[32] A. Sinha and A. Paradkar, "Model based functional conformance testing of web services operating on persistent data," in *Proc. of TAV-WEB*. ACM Press, 2006, pp. 17–22.
[33] Y. Yuan, Z. Li, and W. Sun, "A graph-search based approach to bpel4ws test generation," in *Proc. of ICSEA*. IEEE Computer Society, 2006, p. 14.
[34] J. Yan, Z. Li, Y. Yuan, W. Sun, and J. Zhang, "Bpel4ws unit testing: Test case generation using a concurrent path analysis approach," in *Proc. of ISSRE*. IEEE Computer Society, 2006, pp. 75–84.

Yongyan Zheng is currently a Ph.D. candidate at the University of Surrey in the Computing Department. Her research interest is in web services and test automation.

Jiong Zhou is currently a Ph.D. candidate at the University of Surrey in the Computing Department. His research interest is in analysis of software components and formal methods.

Paul Krause is Professor of Software Engineering at the University of Surrey in the Computing Department. His research interest is in developing processes and methods to develop and assess high quality software. Specific interests include: requirements management; analysis of emergent properties of software components; automatic test case generation and execution; statistical models for software quality assessment.