A Model Checking based Test Case Generation Framework for Web Services

Yongyan Zheng, Jiong Zhou, Paul Krause
Department of Computing, University of Surrey
Guildford, GU2 7XH, UK
{y.zheng,j.zhou,p.krause}@surrey.ac.uk

Abstract

BPEL (Business Process Execution Language) is an emerging standard language to describe web service composition behaviour. The advanced features of BPEL such as concurrency and hierarchy make it challenging to verify BPEL models. Previously, we proposed WSA (web service automata) to be the formal models for BPEL. Based on WSA, this paper presents a model checking based test case generation framework for BPEL. We apply SPIN and NuSMV model checkers as the test generation engine, 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. 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.

1. 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 standard language to model the behaviour of web service compositions. As is well known, it is tedious and time-consuming to create test cases manually, especially for large and complex models. BPEL is a semi-formal flow-based language with complex features like activity hierarchy, concurrency, and dead-path-elimination. Hence, it is desirable to introduce automatic test case generation tools for BPEL. In order to verify BPEL rigorously, there are a number of proposals for applying model checking to verify BPEL, by transforming BPEL models into formal models such as process algebras, Petri nets, and automata [9]. From the testing point of view, we use BPEL as the test model to derive test cases.

Based on our previously proposed web service automata (WSA) [14], this paper presents an automatic test case generation framework for BPEL. It is based on model checking and the test criteria are coverage oriented. Since NuSMV [3] and SPIN [7] model checkers are already used on a regular basis for the verification of real-world applications, they are used as two alternative test generation engines in our framework. 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 links declared in BPEL models will be considered in data flow testing. Two-levels of test cases will be generated in our framework. WSDL test cases are for unit testing to check the interface conformance between the implementation and the WSDL of individual services. BPEL test cases are for integration testing to check the behavioural conformance between the web service interactions and the various behavioural scenarios in BPEL models.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 gives a review of model based testing, the semantics of our WSA, and the machine hierarchical graph of WSAs. Section 4 presents our test generation framework in detail. Finally, section 5 concludes the paper and outlines future work.

1.1. Background

In this section, we introduce the application of model checking to testing, WSA semantics, the hierarchical modelling and data modelling. We use *machine* as shorthand for a web service automaton, and call the machine associated with BPEL x activity as x machine.

- **Model Checking in Testing**

Model checking is a formal verification technique for determining whether a system model satisfies certain properties. Proposals of applying model checking in coverage-based testing were made in [8, 6, 11]. The idea is to use a model checker to find test cases by formulating test criteria as a *trap properties* to be verified. A trap property is the negation of the original property. The process consists of four steps. First, the design models are mapped to finite

International Conference on Information Technology (ITNG'07)
0-7695-2776-0/07 $20.00 © 2007 IEEE

state automata suitable for model checkers. Second, the test criteria are encoded into temporal logic, such as CTL or LTL formulae. Third, a counterexample is generated if the model does not satisfy the temporal logic formula. A counterexample is an execution trace that will take the finite state model from its initial state to a state where the violation occurs. Finally, a test case can be retrieved from the counterexample. To achieve test coverage, the test criterion will be encoded into a set of trap properties, so that test cases satisfying the test criterion can be retrieved from a set of counterexamples.

- **Web Service Automaton**

Definition 1. We assume that we have available an enumerable infinite set V of variables and sets AX,BX of assignment expressions and Boolean expressions respectively, together with a set D of values. We also assume that we have a set of functions Env where $\epsilon \in Env : V \rightarrow D$ assigns variables of V with values from D. Given an expression exp, we need three functions: 1) $def : AX \rightarrow V$, where $def(exp) \in V$ returns the assigned variable on the left hand side of the assignment. 2) $cuses : AX \rightarrow \wp(V)$, where $\wp(V)$ is the power set of V and $cuses(exp) \subseteq V$ returns the variables on the right hand side of the assignment. 3) $puses : BX \rightarrow \wp(V)$, where $puses(exp) \subseteq V$ returns the variables in the Boolean expression.

Definition 2. A **Web Service Automaton** (WSA) M is a finite state machine, consisting of $WSA_M = (I_M, S_M, s_{0M}, S_{fM}, T_M, \delta_M)$. As a convention, we omit the subscript of M such that $M = (I, S, s_0, S_f, T, \delta)$.

1) I is the signature of M, denoted as a three tuple $I = (E, L, O)$, where E, L, O are pair-wise disjoint and represent a set of input events, internal events, and output events, respectively. Let $Msg = (L \cup E \cup O)$ be the set of events. We refer to the elements of $L = L_{in} \cup L_{out}$ as internal input events and internal output events, and to those of $Msg = (E \cup O)$ as 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) $T \subseteq IN \times BX \times (\wp(AX) \cup OUT)$ is a set of transitions, where $IN = (E \cup L_{in} \cup \{\Omega\})$ and $OUT = (O \cup L_{out} \cup \{\Omega\})$. For each $t = (m, g, a) \in T$ (graphically denoted as $m[g]/a$), $m \subseteq IN$ is a set of triggering events, $g \in BX$ is the guard predicate, and $a \subseteq (\wp(AX) \cup OUT)$ is the action set composed of assignments and output events. Ω indicates the omission of an event. We would represent a transition by $\Omega[g]/\Omega$ which simply determines a state change and nothing else. The elements of transition t are denoted as $t.m = m, t.g = g, t.a = a$.

- The events of the transition input event set $t.m \subseteq IN$ are linked by logical operator conjunction, disjunction, or negation, denoted as $AND : e_1 \wedge e_2, .., \wedge e_n$, $OR : e_1 \vee e_2, .., \vee e_n$, and $NOT : \neg(e_i)$, respectively.
- The *data structure* of machine M is the form of (V_M, AX_M, BX_M), which can be retrieved from T. Since $T \subseteq IN \times BX \times (\wp(AX) \cup OUT)$, we can retrieve $AX_M = \{exp \in AX | \exists t \in T \wedge exp \in t.a\}$, $BX_M = \{exp \in BX | \exists t \in T \wedge exp \in t.g\}$, and V_M which is the disjoin union of $\bigcup_{exp \in AX}(\{def(exp)\} \cup cuses(exp))$ and $\bigcup_{exp \in BX}\{puses(exp)\}$.

4) $\delta \subseteq S \times T \times S$ is the transition relation.

Asynchronous execution of web services is achieved by using queues for message processing. The default queuing protocol in WSA is to associated a FIFO queue for each message. WSA communicate by message passing.

- **Hierarchy Modelling**

Since WSA has no hierarchy, we simulate the hierarchical relationships of BPEL activities by adding *start* and *done* as common administration messages between machines. A machine can play the role of parent or child. For a machine M_j, if M_i sends a start message to M_j, then M_i is the parent machine of M_j and M_i is the child machine of M_i. A child machine will send a done message to its parent machine when reaching one of its final states. Each machine has zero or one parent machines, and zero or many child machines. Since the BPEL basic activity is atomic and a BPEL structured activity is hierarchical, the machine for BPEL basic activity has no child, and the machine for BPEL structured activity has $0..*$ children.

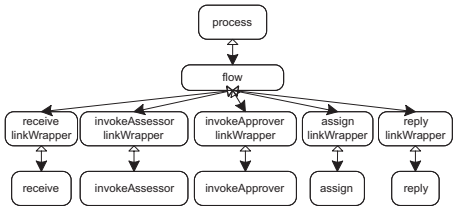

Figure 1. An example of machine hierarchy

Fig 1 shows the hierarchical relationships of the classic loanapproval example [1]. The dark arrows denote the *start* messages sent from parents to children, and the hollow arrows denote the *done* messages returned from children to parents. The machine without receiving a start message is the BPEL *process* machine, denoted as M_{proc}. The machine without receiving any done message is a BPEL *basic* machine. In Fig 1, the *process* machine is the parent

of *flow* machine, and the *flow* machine is the parent of *receivelinkWrapper* machine, which in turn is the parent of *receive* machine.

- **Data Flow Modelling**

Data flow captures the relations between inputs and outputs of BPEL activities. In BPEL, variables and links may affect the control flow; variables may appear in expressions in switch and while conditions, and may also be used in the conditions to fire particular 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. Links are Boolean variables declared in the links tag of the flow activity. BPEL handles data by a 'blackboard' approach. BPEL variables and links can be used and defined by the process or scope enclosed activities, and the flow enclosed activities, respectively. By message passing, we analyse BPEL activities to discover data dependencies among activities. Fig 2 shows the data-flow model of the loanapproval example.

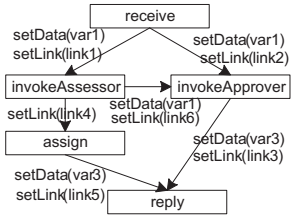

Figure 2. Data flow model

2. Test Coverage Criteria in Temporal Logic

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. 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, each criteria includes $s \in S_{fMproc}$, which forces the model checkers to execute a BPEL model to its end.

Definition 3. Suppose a BPEL model is associated with a set of WSAs $\{M_1, .., M_n\}$, a **test case** of the BPEL model starts from the initial state of M_{proc}, and ends at one of the final states of M_{proc}.

Definition 4. A test suite is said to achieve **state coverage** of a set of WSAs $\{M_1, .., M_n\}$, if each state $s \in S_{Mi}$ and one of the final states $s_f \in S_{fMproc}$ can be executed at least once.

Definition 5. A test suite is said to achieve **transition coverage** of a set of WSAs $\{M_1, .., M_n\}$, if each transition $t \in T_{Mi}$ and one of the final states $s_f \in S_{fMproc}$ can be executed at least once.

Data flow testing is interesting because stimulating the sequences of operations which define and subsequently use variable values is an effective systematic method for exposing faults. For the *du-path coverage*, we adopt the definition from [10]. According to definitions 1 and 2, given a variable v and a transition t, v is *def* in t if $v = def(exp)$ where $exp \in t.a$, v is *computation-use* in t if $v = cuses(exp)$ where $exp \in t.a$, and v is predicate-use in t if $v = puses(exp)$ where $exp \in t.g$. We use $d(v)$ and $u(v)$ to denote the sets of transitions where v is defined and computation-used (or predicate-used), respectively. A transition sequence $\langle t_1, t_2, .., t_n \rangle$ is a *def-clear-path* with respect to variable v if v is not defined in ti where $1 \leq i \leq n$. A *du-pair* of v is the transition pair (t_i, t_j) where v is defined in t_i and is computation-used or predicate-used in t_j. A *du-path* is a transition sequence that (t_i, t_j) is a du-pair and there is a def-clear-path from t_i to t_j.

Note that we are only interested in the variables explicitly declared in BPEL models (denoted as V_{bpel}), and the data dependency between BPEL activities. So, 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 is said to achieve **all-du-path coverage** of a set of WSAs $\{M_1, .., M_n\}$, if for each $v \in V_{bpel}$, every du-path with respect to each $v \in V_{bpel}$ can be executed at least once, and one of the final states $s_f \in S_{fMproc}$ can be reached from each du-path.

Now we can encode the test coverage criteria into CTL and LTL temporal logic. CTL (Computation Tree Logic) views time as branching, so from a given branch alternative states may be reached. In the following, we use the temporal operators E (there exists some path), F (finally), X (next), and U (until). [8] gives a detailed study of encoding various structural test coverage criteria into CTL. Based on this work, the negation of state, transition, and all-du-path coverage criteria are encoded into the CTL of 1), 2), and 3) as follows. Here M is a web service automaton.

1) $\{\neg EF(s_i \wedge EF\ s_f)\}$ where $s_i \in S_M, s_f \in S_{fMproc}$.

2) $\{\neg EF(t_i \wedge EF\ s_f)\}$ where $t_i \in T_M, s_f \in S_{fMproc}$.

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

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. We use temporal operators [] (always), $\langle\rangle$ (eventually), X (next), and U (until). Suppose M is a web service automaton. The negation of state, transition, and all-du-path coverage criteria are encoded into the LTL of 1), 2), and 3) as follows.

1) $\{\neg \langle\rangle (s_i \wedge s_f)\}$ where $s_i \in S_M, s_f \in S_{fMproc}$.

2) $\{\neg \langle\rangle (t_i \wedge s_f)\}$ where $t_i \in T_M, s_f \in S_{fMproc}$.

3) $\{\neg \langle\rangle (t_i \wedge X(\neg d(v) U t_j) \wedge \langle\rangle s_f)\}$ where $v \in V_M$, $t_i \in d(v), t_j \in u(v), s_f \in S_{fMproc}$.

3. Test Generation Framework

In this section, we will elaborate the proposed BPEL test case generation framework, shown in Fig 3.

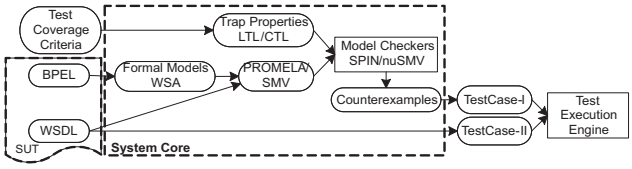

Figure 3. Test framework architecture

- 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 WSA and a WSA for link handling. A BPEL structural machine can start and stop its enclosed machines. The propositional input events of WSA can capture various BPEL features. The logical-AND operator can capture the synchronization of *end*. For instance, a BPEL flow activity will not end until all its enclosed activities finished. In a flow machine, this feature can be captured by adding the logical-AND to the incoming *done* messages from its children. The logical-OR operator can model the fault propagation. In an activity, a fault is propagated as long as one of its enclosed activities raises a fault. The logical-AND together with logical-NOT can model priority BPEL messages such as termination or stop messages. BPEL data flow is analyzed explicitly, so that interactions of BPEL activities can be modelled by message passing. Details of how WSA model various features can be found in [14]. It is essential to provide an intermediate model between BPEL and model checkers. Without such 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 relationships 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 relationships 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. However, the state space increases dramatically with such models. To reduce the state space, we implement a simple queue model holding only one message. Note that the values of *String* variables need to be enumerated in a user-define type section.

- From WSDL to PROMELA or SMV models

Since the message type of BPEL variables is declared in associated WSDLs, we do not model such message types in WSA. Rather, the required message types will be extracted from the corresponding WSDL and mapped to Promela and SMV models. In Promela, a message type in WSDL is declared as a complex type with keyword *typedef*. In SMV models, a message type in WSDL is declared as a complex type with keyword *MODULE*.

- Encoding Test Coverage Criteria into Trap Properties

In the 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 the 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 4 (a) shows a never claim for covering a state and eventually reaching a final state of the process machine. $\langle\rangle\ (p \wedge q)$ 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 LTL state coverage in definition 6, a set of the negation of such LTL formulae can provide the state coverage. Since SPIN can only verify a property in one run, it needs n runs for n pairs of (p, q).

The NuSMV model checker accepts LTL or CTL formulae. a formula starts with the keyword *SPEC* in SMV models. Fig 4 (b) shows a CTL formula declaration for covering a state and a final state of the process machine. According to the CTL state coverage definition in section 4, a set of such CTL formula can provide state coverage of the BPEL model. Since NuSMV can verify more than one property in a run, SMV only needs to run once for a set of CTL formula.

```
(a) #define p (loanapproval_flow_receive1:state == s2)
    #define q (loanapprovalstate == s3 || loanapprovalstate == s1)
    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 loanapprovalstate = s3 | loanapprovalstate=s1)
```

Figure 4. An example of state coverage

- From Counterexamples to Test Cases

The test cases retrieved from counterexamples are called BPEL based test cases. The BPEL based test cases focus on the sequencing of invocations of the provided services. The transition names (t_i) are modelled explicitly in Promela and SMV models, so that a transition name list can be retrieved from the generated counterexample. A test case can be derived from the transition name list, by extracting the corresponding transition input events, guards, actions, and output events from the associated WSA model. This kind of test case checks whether the web service interactions conform to the communication protocols modelled in BPEL. A test case consists of a set of execution paths of the BPEL. For instance; a path representing the customer's request is less than 10000, the assessor service is invoked, and the returned risk is low. This execution path involves two services; loanapproval and assessor. If the loanapproval is selected as the service-under-test, the customer and the assessor will be two testers. In this case, the test case will remind users to input the right range values of request and risk.

- From WSDL to Test Cases

The test cases generated from WSDL cover validation of single operations. The execution of test cases will invoke remote operations provided by the services. This kind of test case checks whether the implemented service operations conform to the published service modelled in WSDL.

- Execution of Test Cases

BPEL test cases will call the methods of WSDL test cases, so that the both dynamic interaction behaviour and static individual operation of remote web services can be tested. The two levels of test cases that are generated can run using the common JUnit test execution engine. A GUI is provide for users to input test data manually.

4. Symbolic Predicates

In a state machine with guards, the predicates of different paths need to be true alternatively, so that a model checker can be forced to explore alternative paths. For instance, $path_1$ and $path_2$ will be executed when $request.amount < 10000$ and $request.amount \geq 10000$ are true, respectively. Instead of inputing the actual parameter $request.amount$ with value less than 10000 (resp. greater than), we use a symbol $pred_i$ to represent each predicate. *Gray code* (e.g. [13]) on the predicates can be derived, where two successive values differ in only one digit, such that a model checker can explore all the paths. For instance, we use $pred_1$ and $pred_2$ to represent the above two predicates, the two-bit gray code matrix $(pred_1, pred2) : (0,0), (0,1), (1,1), (1,0)$. The values 1 and 0 denote boolean *true* and *false* respectively. For a state s_i, if t_{i0} and t_{i1} are the only two active transitions of s_i, where $pred_{i0} = t_{i0}.g, pred_{i1} = t_{i1}.g$, then the relationship $pred_{i0} \neq pred_{i1}$ can be derived. In this case, the combination $(0,0)$ and $(1,1)$ can be removed.

```
(a) bool pred1 /* request.amount<10000 */
    bool pred2 /* request.amount>=10000 */
    /* 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) ;
    }
```

```
(b) --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 5. An example of predicate handling

The corresponding Promela code is shown in Fig 5(a). First, each symbolic $pred_i$ is declared as a global boolean

variable. A gray code matrix is constructed by inputing the predicate symbols. Second, list all predicate relationships. These relationships will be read by an application to reduce the predicate combinations. Third, two processes will be inserted into the Promela model. A *runner* process assigns values of predicates based on the matrix from step 2), and a *chooser* process choose the current values of predicates. The *chooser* will be started from the top level *init* process. Similarly, the SMV code is shown in Fig 5(b). A *chooser* module declares the predicates, showing the predicate relationships as comments. An application will read the predicates and generate gray code matrix, reduce the matrix based on predicate relation, and finally the *ASSIGN* section will be inserted into the SMV model, so that *chooser* can select the current values of predicates. The *chooser* will be started from the top level *main* module.

5. Related Work

A number of papers in the literature propose formal semantics for BPEL, and apply model checking techniques to verify BPEL. However, there is less effort in using BPEL as the test model for deriving test cases. [2] proposes a framework to augment WSDL with a UML2.0 PSM (Process State Machine) to model the web service interactions. After transforming PSM to a Symbolic Transition System, existing *ioco-conformance* testing tools can be applied. [5] proposes to use Graph Transformation Rules along with WSDL to generate test cases. [12] use WSDL-S to be the service behaviour model, where it extends WSDL by adding a pre-condition and post-condition to each WSDL operation. The WSDL-S is mapped to EFSM so existing test techniques for EFSM can be applied.

Our work is different from existing work in two aspects. First, we propose an automata-based model which is suitable for model checking tools. The propositional input events of WSA can capture most BPEL features and reduce state space. During the mapping from BPEL to WSA, BPEL data flow is elicited. Also, the internal interactions of BPEL activities and external interactions of BPEL models can be captured by message passing. Second, we realize an automatic test case generation framework for BPEL. It is not new to apply model checking to achieve test coverage [8, 6, 11], but it is new to apply such a technique in the domain of web services. As far as we aware, we are the first to consider both control and data flow testing for BPEL.

6. Conclusions and Future Work

In this paper, we presented an automatic test generation framework for BPEL, based on model checking and the state, transition, and du-path test coverage criteria. The generated BPEL based and WSDL based test cases can check the conformance of web service behaviour and interface, respectively. The proposed web service automata has been implemented in XML, and the transformation engines are implemented in XSLT. An Eclipse plug-in was developed in Java to invoke various transformation engines, to enable a user to choose the service under test (i.e. BPEL) and the test coverage criteria, and to interact with model checkers. The test framework is part of the *DBEStudio* deliverable for the EU project [4]. An extension of this work is to define additional test coverage criteria which are suitabe for BPEL, i.e. criteria for integration testing, so that model checking can be used on scalable models.

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

References

[1] 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.

[2] A. Bertolino and A. Polini. The audition framework for testing web services interoperability. In *Pro. of EUROMICRO*, pages 134–142. IEEE Computer Society, 2005.

[3] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: A new symbolic model verifier. In *Proc. of CAV*, pages 495–499. Springer, 1999.

[4] D. B. Ecosystem. *http://www.digital-ecosystem.org*.

[5] R. Heckel and L. Mariani. Automatic conformance testing of web services. In *Proc. of FASE*, pages 34–48. Springer, 2005.

[6] M. P. E. Heimdahl, D. George, and R. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In *Proc. of HASE*, pages 178–186. IEEE Computer Society, 2004.

[7] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[8] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *Proc. of ICSE*, pages 232–242. IEEE Computer Society, 2003.

[9] R. Hull and J. Su. Tools for design of composite web services. In *Proc. of SIGMOD*, pages 958–961. ACM Press, 2004.

[10] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985.

[11] S. Rayadurgam and M. P. E. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. of ECBS*, page 0083. IEEE Computer Society, 2001.

[12] A. Sinha and A. Paradkar. Model-based functional conformance testing of web services operating on persistent data. In *Proc. of TAV-WEB*, pages 17–22. ACM Press, 2006.

[13] Wikipedia. *http://en.wikipedia.org/wiki/Gray_code*.

[14] Y. Zheng and P. Krause. Automata semantics and analysis of bpel. In *Proc. of DEST*. IEEE Computer Society, 2007.