In Proceedings of the 8th IEEE International Symposium on Object-oriented Real-time Distributed Computing, 2005, pages 300-307.

Automated Model Checking and Testing for Composite Web Services

Hai Huang Wei-Tek Tsai Raymond Paul* Yinong Chen
Dept. Computer Sci. & Eng., Arizona State University, Tempe, AZ, 85287-8809
{hai, wtsai, yinong}@asu.edu

**OSD NII, Department of Defense*
raymond.paul@osd.mil

Abstract

Web Services form a new distributed computing paradigm. Collaborative verification and validation are important when Web Services from different vendors are integrated together to carry out a coherent task. This paper presents a new approach to verify Web Services by model checking the process model of OWL-S (Web Ontology Language for Web Services) and to validate them by the test cases automatically generated in the model checking process. We extend the BLAST, a model checker that handles control flow model naturally, to handle the concurrency in OWL-S. We also propose enhancement in OWL-S and PDDL (Planning Domain Definition Language) to facilitate the automated test case generation. Experiments on realistic examples are provided to illustrate the process.

1. Introduction

Web Services (WS) receive significant research recently from both academia and industry due to its broad applications and flexible architecture supporting re-composition and reconfiguration [23, 20]. As the complexity of composition increases, verification and validation (V&V) of the composite WS become a sophisticated task that deserves and has received many studies. Two major V&V approaches are automated testing [22] and model checking [5, 6, 8]. The main benefit of model checking is to provide an exhaustive proof-style certificate that the model, if not the software itself, satisfies some properties, such as safety temporal properties. The violation of the properties, e.g., deadlock, may be harmful; hence complete elimination is desirable.

This paper presents an approach in a Collaborative Verification and Validation (CV&V) framework [24]. If any fault is found during the verification phase, the WS will be re-composed. The testing part is to validate whether the composite WS exhibits the desired properties as guaranteed on the model by verification phases and does not exhibits the undesired properties. Hence both positive and negative tests need to be employed.

Several existing approaches discussed model checking composite WS. These studies adopt different models, utilize different model checking approaches, or check different types of properties [10, 11, 17, 25]. The majority of them adopt BPEL4WS (Business Process Execution Language for Web Services) [1] or a BPEL-like model to model WS, and utilize SPIN [15] or CCS-based (Calculus of Communicating Systems) model checker [17], e.g., CWB (CCS Workbench) [9], to do the model checking. A common theme of existing approaches is that they treat atomic WS as black-box. The process-oriented models successfully capture the temporal properties among atomic WS. However, if the internal structure of each atomic WS is blank in the model specification, it is inherently hard to describe and check more delicate properties involving the effect and output of each atomic WS. For example, consider the property that a customer must have at least one product in the cart before invoking atomic WS *checkout*. Clearly we cannot describe or check the property, unless we know that the effect of other atomic WS, such as *add-to-cart*, change the amount of products. Though we can still check the temporal property that *add-to-cart* must occur at least once before *checkout*, this property cannot substitute the former one due to the subtle difference between them. Formally, we name such properties that at location l a variable holds certain values as *data-bound properties*. The problem becomes more complicated when the effect is

conditional with respect to the effects or the outputs of previous atomic WS. Note that pure in-lining technique may not be feasible here since a composite WS could consist of atomic WS that are proprietary from different parties.

Our model checking technique for composite WS is based on the process model of OWL-S (Web Ontology Language for Web Services) [3] and the model checker BLAST [13]. Each atomic WS is no longer a black-box because its behavior is bounded by the OWL-S specification. The goal of OWL-S is to enable automatic WS discovery, composition, invocation, and monitoring. The process model of OWL-S is a control flow model. The BLAST was designed for handling control flow automata and applies directly to C source code. The checkable properties of BLAST include the *predicate-bound properties* that at location *l* a predicate *p* holds a certain truth value. The data-bound properties can be translated to predicate-bound properties. Both the control flow model and data-bound properties are intuitive to software engineers. BLAST also generates positive test cases automatically [14].

Table 1. Expressiveness of OWL-S, BPEL4WS, and WSCDL

	Internal logic
OWL-S	Conditional and unconditional **I**nput, **O**utput, **P**arameter, and **E**ffect (IOPE)
BPEL4WS	N/A
WSCDL	State change and alignment among WS
	Variable and data
OWL-S	Lack the connection from data to predicates
BPEL4WS	Support integral expression (all values are integer)
WSCDL	States (only support state transition)
	Control logic
OWL-S	Sequence / if-then-else / choice one / concurrent execution (split + join) / unordered / loop (iteration, repeat-while, repeat-until)
BEPL4WS	Sequence / switch / while / pick m out of n / fault handler / event handler / concurrent execution
WSCDL	State transition
	Exception handling
OWL-S	N/A
BPEL4WS	Fault handler
WSCDL	N/A

Table 1 compares the expressiveness of OWL-S, BPEL4WS, and WSCDL (Web Services Choreography Description Language). The expressiveness breaks further into four sub criteria: internal logic of each atomic WS, variable and data, control logic, and exception handling.

The contribution of this paper include (1) applying BLAST to WS and evaluating atomic WS while previous approaches treat atomic WS as a black box; (2) extension of BLAST to handle concurrency in precise OWL-S semantics; (3) extension of OWL-S and PDDL (Planning Domain Definition Language) [12] to better facilitate both positive and negative test case generation.

This paper is organized as follows. Section 2 presents the process through an example. Section 3 elaborates the conversion from OWL-S process model to the input of BLAST. Section 4 presents the extension of BLAST to handle the concurrency semantics in OWL-S. Section 5 discusses how to automatically embed to-be-checked properties. Section 6 studies test case generation and presents some experimental data. Section 7 concludes this paper.

2. The overall process

This section uses an example to illustrate the process of automated model checking and testing on OWL-S process model. The process consists of the following steps: (1) convert OWL-S model to a C-like specification language; (2) embed properties to be checked into the specification language; (3) feed the specification language into BLAST; (4) model checking and positive test case generation; and (5) negative test case generation. The process is shown in Figure 1.

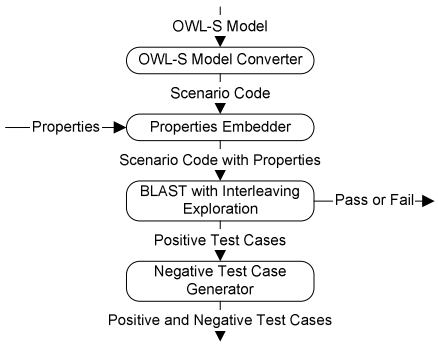

Figure 1. Automated model checking process

2.1. Online shopping

The example is a process model shown in Figure 2. We avoid the verbose XML (Extensible Markup Language) specification here for the sake of clarity.

Instead, we denote the types of the structural constructs on the links among atomic WS.

As shown in Figure 2, a customer must first login. If login succeeds, the customer has a *choice* to add a product to the cart, remove a product from the cart, or place an order. After placing an order, the product will be shipped to the customer and the customer's credit card will be charged. The *split* and *join* means that shipping and charging credit card are concurrently executed. The *repeat-until* construct means that the user can keep adding or removing products before placing the order.

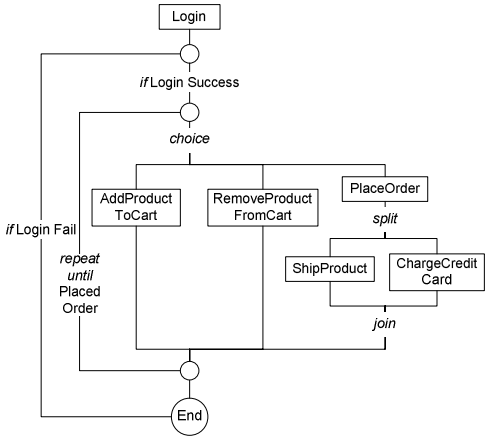

Figure 2. Online shopping WS

Although the control flow diagram resembles the control flow automata for BLAST, translating it to the C-like language is not straightforward. The challenges are: (1) extract proper Input, Output, Precondition, and Effect (IOPE) information from the XML specification, without which, certain properties are not checkable. For example, we need to know that the effect of *AddProductToCart* to check the data-bound property, i.e., when placing an order, there must be at least one product in the cart; (2) certain structural constructs, such as *choice* and *split* + *join*, are not recognizable to BLAST; (3) BLAST cannot handle the OWL-S concurrency semantics, such as the computation between *split* and *join*.

The IOPE information is encoded as logical formulas in OWL-S [3]. Candidate languages for logical formulas are PDDL [12] and KIF (Knowledge Interchange Format) [2]. The problem with KIF and previous versions of PDDL is the lack of capability of specifying how variables acquire values. This problem is solved by PDDL2.1 with the *assign* operator. To properly support model checking, we utilized PDDL2.1. Figure 3 lists the sample specification of atomic WS *Login*, *AddProductToCart*, and *RemoveProductFromCart* in PDDL2.1.

```
Login
:effect (assign (amount ?item) 0)

AddProductToCart
:effect (increase (amount ?item) 1)

RemoveProductFromCart
:effect (decrease (amount ?item) 1)
```

```
Login
item = 0;

AddProductToCart
item = item + 1;

RemoveProductFromCart
item = item - 1;
```

Figure 3. Sample PDDL 2.1 specification and corresponding C-like code

The names starting with "*?*" are variables. The specification simply says that *Login* will set the cart to empty; *AddProductToCart* will increase *?item* by 1; while *RemoveProductFromCart* will decrease *?item* by 1. Most structural constructs, except *choice* and *split* + *join*, can be translated into C-like code directly. For *choice*, we implement it by a sequence of *if-then-else if-…-else* statements.

There is no corresponding C construct for *split* + *join*. We thus generate all effective interleaving of concurrent execution. In other words, BLAST will check two copies of C-like code with both interleaving of *ShipProduct* and *ChangeCreditCard*. Actually the interleaving algorithm is integrated in the BLAST to avoid duplicated checking on the same non-concurrent part.

Next we need to embed the to-be-checked properties into the C-like code. Essentially, BLAST computes a reachability set on statements during model checking process. By introducing an ***ERROR*** statement, all checkable properties can be translated to the reachability of certain ***ERROR*** statements. Suppose we want to check two properties: (1) a data-bound property, when placing order, there must be at least one product in the cart; and (2) a temporal property, we must charge the credit card before shipping the product. The properties are translated into ***ERROR*** statement as shown in Figure 4.

```
if ( item < 1 )
    ERROR;
PlaceOrder;
shipped = false;
ShipProduct;
shipped = true;
if ( shipped )
    ERROR;
ChargeCreditCard;
```

```
if ( item < 1 )
    ERROR;
PlaceOrder;
shipped = false;
if ( shipped )
    ERROR;
ChargeCreditCard;
ShipProduct;
shipped = true;
```

Figure 4. Embedding to-be-checked properties

The inserted code for to-be-checked properties is highlighted. Note that we introduce a new variable *shipped* to indicate whether *ShipProduct* is executed. Clearly, property (1) is violated if and only if the first ***ERROR*** is reached, and analogously to property (2) and the second ***ERROR***. Embedding the properties and interleaving exploration are two independent steps.

In reality, embedding happens before interleaving as interleaving is integrated with the BLAST model checker. We present the different order to reveal the ***ERROR*** immediately.

Now we can feed the code into BLAST. For each violation, BLAST will return a counterexample. In this case, we will have a counterexample:

Login, *RemoveProductFromCart*, *PlaceOrder*,

which violates property (1), and counterexample

Login, *AddProductToCart*, *PlaceOrder*, *ShipProduct*, *ChargeCreditCard*,

which violates property (2).

If no property is violated, BLAST will generate positive test cases. Negative test cases can be selectively generated by using the technique presented in [24]. Both positive and negative test cases are applied to verify the WS.

3. Convert OWL-S to C-Like code

This section presents the translation from OWL-S process model to the C-like specification language for BLAST. The translation is divided into two sub-problems: converting structural constructs into OWL-S and converting logical formulas into PDDL2.1.

3.1. Convert structural constructs

According to OWL-S 1.1 beta version [3], the process model supports the following structural constructs: *sequence*, *split*, *split + join*, *unordered*, *choice*, *if-then-else*, *iterate*, *repeat-while*, and *repeat-until*. Most constructs have natural correspondence in our C-like language and thus the translation is straightforward. Hence we focus on *split*, *split + join*, *choice*, and *unordered*. Some constructs has a condition associated with it, such as *if-then-else*, *repeat-while*, and *repeat-until*. The conditions are in logical formulas, and their translations will be discussed in Section 3.2.

Choice: The *choice* is significantly scaled down to choose one from multiple choices. We implement this by introducing a dummy variable, *choice_m*, and a sequence of *if-then-else if-else if-…-else* that relies on *choice*. The value of variable *choice_m* is not specified, to force BLAST to explore all branches. Formally, choosing one from $branch_i$, $i \in [n]$, is translated into the code listed in Figure 5. The postfix *_m* is to differentiate the different *choice*s.

Split and split + join: These constructs specify the concurrent executions of threads. We translate concurrent executions of $thread_i$, $i \in [n]$, into the code listed in Figure 5. There are two types of threads, *join_thread* and *thread*. A *join_thread* indicates that after the thread is finished, the control will return to the synchronization point specified by the *join* in OWL-S, while *thread* has no such restriction, which correspondents to the loose restriction on synchronization of *split* in OWL-S. Note that OWL-S supports partial synchronization, i.e., only "join" some threads. Hence it is possible that in the scope of the *parallel*, not all threads are *join_thread*. The interleaving exploration algorithm presented in Section 4 will process the code and generate all meaningful interleaving.

```
if ( choice_m == 1 ) {
    branch_1
}
else if ( choice_m == 2 ) {
    branch_2
}
...
else {
    branch_n
}
```

```
parallel {
    join_thread {
        thread_1
    }
    ...
    thread {
        thread_n
    }
}
```

Figure 5. C-like code for *choice*, *split*, and *join*

Unordered: *Unordered* is implemented in the same way with *split + join* to explore all meaningful execution orders, as it requires the completion of all components.

3.2. Convert logical formulas

We assume logical formulas are expressed in PDDL2.1 [12]. The *assign* operator in PDDL2.1 links variables to values. We support the *Action* (and related) syntax listed in Appendix 2 in [12], which includes arithmetic operators, numeric functions, numeric comparison operators, predicates, and conditional effects. We do not support the *forall* and *exists* in PDDL2.1 yet, and they can be added later. The conversion of formulas is straightforward though the formulas in PDDL2.1 are in pre-order while in C it is in in-order.

4. BLAST for concurrency in OWL-S

This section presents an enhancement on BLAST to check concurrency in OWL-S. BLAST bears capability to model checking concurrent threads [14]. However, the synchronization semantics in *split + join* is not supported by C grammar, we enhance BLAST to directly exhaust every interleaving of the concurrent threads that matters. We extend the partial order reduction approach traditionally developed for state transition system [8] to control flow automata. We incorporate the summarizing technique [19] to reduce

the complexity of the interleaving. The enhancement includes 1) single forward pass (without backward tracking) algorithm for interleaving exploration; and 2) amenable for the **NEXT** operator in LTL (Linear Temporal Logic).

An atomic WS is a WS whose process is virtually transparent, which is the fundamental unit for model checking concurrent execution of composite WS. Due to the transparency of atomic WS, only the order of atomic WS matters during model checking. Let w_1 and w_2 be two atomic WS, W be the set of all atomic WS. Let S be the set of *global states* defined by the combination of all parameters except the local parameters of atomic WS in W. Define *independency* relation $I \subseteq W \times W$ be a binary relation, $(w_1, w_2) \in I$ if for any state s, there exists a state s', such that both the ordered execution (w_1, w_2) and (w_2, w_1) transit s to s', without violating any properties that consists of **NEXT** operator. The two atomic WS w_1 and w_2 are independent, if $(w_1, w_2) \in I$. Clearly, the execution order of independent atomic WS does not affect the model checking, thus we can execute them in any order.

Define $RW(\bullet)$: $W \rightarrow 2^P$ to be the *read-write* set of an atomic WS, where P is the set of all global parameters. For a parameter $p \in P$, if p is referenced by the IOPEs of a WS w, then $p \in RW(w)$. Let $N \subseteq P \times P$ be a binary relation, such that $(p, p') \in N$ if there exists a property to be model checked, the property contains **NEXT** operator, p occurs to one side of the **NEXT** operator, and q occurs to the other side of the **NEXT** operator. The following Lemma presents a simple criterion to identify independent atomic WS.

Lemma 1 Let w_1 and w_2 be two atomic WS. If $RW(w_1) \cap RW(w_2) = \varnothing$, and for any $p \in RW(w_1)$ and $p' \in RW(w_2)$, $(p, p') \notin N$, then w_1 and w_2 are independent.

The proof is straightforward and hence omitted. Note that Lemma 1 takes care of the **NEXT** operator. During model checking, we need only to exhaust all the orders among dependent atomic WS. We remark that counting all read-write parameters is sufficient but sometimes overapproximating. We remark that a fine-grained read-write, write-read, and write-write analysis may further reduce the complexity.

We incorporate concurrent summarizing technique [19] into our analysis. The concurrent summarizing technique can be viewed as a precise inter-procedural data-flow analysis for concurrent threads. It represents the execution of statements inside a transaction (atomic WS) by a single state transition to reduce the model checking effort when the transaction is visited again in the model checking process.

The transaction boundary partitions each thread into two parts. Instead of applying summarizing technique to the transactions, which is inside the atomic WS and cannot help us much in our case, we apply summarizing technique to the partition outside the transitions, which expands the independency relation to $B \times B$, where B is the set of basic blocks of atomic WS. Two basic blocks b and b' are independent if for any $w \in b$ and $w' \in b'$, w and w' are independent. When exploring all the possible interleaving, we can advance on a thread one independent basic block instead one atomic WS in each step. We also summarize dependent atomic WS to the independent basic block following it, to be a dependent basic block. The above summarizing enables the single-forward-pass algorithm because it guarantees no meaningful interleaving will be missed in the forward exploration (detailed in the proof of Theorem 1).

We devise an interleaving exploration algorithm to facilitate BLAST to exhaust all possible concurrent execution sequences. The algorithm is relatively independent of the LazyAbstraction algorithm [13]. The decoupling process facilitates the correctness analysis and the implementation of the algorithm. We first present an example, and then the algorithm.

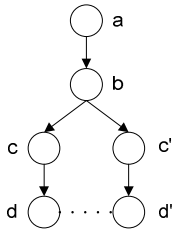

Figure 6. Sample control flow automata

Consider the control flow automata in Figure 6. It contains two concurrent threads (c, d) and (c', d') are concurrent. The dotted line between d and d' means that d and d' are dependent. During the single thread part (a, b), we just apply the LazyAbstraction algorithm of BLAST to explore atomic WS one-by-one and check whether any properties are violated. In order to handle concurrency, we introduce a frontier set F that marks the choices of the atomic WS to be explored in the next step. When arriving at the split point b, there are two choices in $F = \{c, c'\}$. Since c and c' are independent, we pick up an arbitrary one, say, c in this example. The interesting part is when it reaches the point where $F = \{d, d'\}$. Since d and d' are dependent, we need to explore all possible orders. We achieve this by spawning two successors, one with d picked up, the other with d' picked up. Finally we

explored the two possible interleaving (*d*, *d'*) and (*d'*, *d*). We remark that this will not change the structure of LazyAbstraction, since it already maintains a reachability tree and we just spawn more successors.

Algorithm 1 (Interleaving Exploration)

Initial Phase

1. Identify all dependent atomic WS
2. Summarize all sequential independent atomic WS into basic blocks
3. $F \leftarrow \varnothing$

Main Loop

1. *If* no dependent basic blocks in *F*, choose one basic block *b*; extend the current path with *b*; $F \leftarrow F - \{b\} + \{\text{successor of } b\}$
2. *Else* //exists dependent basic blocks in *F*
3. *Foreach* dependent basic block *b*
4. Spawn a successor *b* and attach to the current reachability tree
5. On *b*, attach $F - \{b\} + \{\text{successor of } b\}$
6. *End Foreach*
7. *End If*

In the line 1 and 5 in the main loop, only after all the basic blocks in all *join_thread*s are explored, could we continue on that following the *join* point. This is to guarantee the correctness on the synchronization of OWL-S. The spawned basic block *b* may consist of more than one atomic WS, which will guide the LazyAbstraction to explore the atomic WS in the basic block first. Embedding Algorithm 1 into LazyAbstraction is straight forward. The Initial Phase goes to that of LazyAbstraction. The Main Loop of Algorithm 1 is embedded into the main loop of LazyAbstraction. LazyAbstraction will explore the atomic WS in its current basic block spawned by Algorithm 1 first.

Theorem 1 Algorithm 1 enumerates all orders of dependent basic blocks with no duplication.

Proof (sketch) Prove by induction on number of threads. If there are two threads, each with only one dependent basic block, it is straight forward to show that both orders are produced. The interesting part is when one thread is (*a*), the other is (*a'*, *b'*), where *a* is dependent on both *a'* and *b'*. We need to generate the order (*b'*, *a*). This is achieved by first choose *a*', and then in the next iteration, at step 1, *F* will be updated to be {*a*, *b'*}, and *b'* has the chance to be chosen first and generate the order (*b'*, *a*). Note that we will not encounter the case that one thread is (*a*), the other is (*a'*, *c'*, *b'*), where *a* is dependent on both *a'* and *b'*, but not *c'*, since we summarized *a'* and *c'* together.

Suppose Theorem 1 holds for *k* threads, prove for *k+1*. The loop at step 3 will pick up basic block from each of the *k+1* threads. After that either we fall back to the *k* thread cases or we apply a second induction on the number of basic blocks. €

Corollary 1 Algorithm LazyAbstraction with Algorithm 1 embedded is correct. If LazyAbstraction terminates on each thread, then LazyAbstraction with Algorithm 1 embedded terminates.

Corollary 1 guarantees the correctness and the termination of our enhancement to BLAST.

5. Embed properties

This section presents how to embed to-be-checked properties into the C-like code. First we formally define the checkable properties.

Let *s* be a statement and *S* be the set of all statements. Define $Reach \subseteq S$ to be the reachability set. A *reachability property* is defined as to determine whether $s \in Reach$, for any given $s \in S$.

Let *f* be any formulas on predicates $p \in P$ with Boolean operators **and**, **or**, and **not**, and temporal operators **NEXT**, **FUTURE**, **ALWAYS**, **UNTIL**, and **RELEASE** with universal quantifier as shared by various types of temporal logic in [8]. A *safety temporal property* is defined to determine whether *f* holds universally. Note that we can encode the temporal relation on statement *s* by introducing a predicate *executed*(*s*), which will be *false* before the statement is executed, and *true* afterwards, and express the formulas in terms of *executed*(*s*).

Let *g* be any formulas on predicates $p \in P$ with Boolean operators **and**, **or**, and **not**. Let $l \in V$ be a location in the control flow automata, where *V* is the set of nodes of the control flow automata. A *predicate-bound property* is defined as to determine whether at location *l*, *g* (or **not** *g*) holds.

The essential algorithm of BLAST is reachability computation that computes the set of statements which is reachable in the program. Hence reachability properties are naturally handled by BLAST.

Now we convert a predicate-bound property to reachability computation. Consider the property: at location *l*, *g* holds. Define an ***ERROR*** statement and embed the following segment of code at location *l*.

if (**not** *g*) ***ERROR***

Then the property is violated if and only if the corresponding ***ERROR*** statement is reached. Predicate-bound property, at location *l*, **not** *p* holds, can be handled analogously.

Now consider safety temporal properties. According to [8], all temporal operators can be expressed by **NEXT** and **UNTIL**, thus we only address these two operators.

f* NEXT *g: the semantics is that *f* holds at the current location and then *g* holds at the very next location. We construct the function **Check_NEXT**.

```
bool Check_NEXT (f, g)
{
    static bool armed = false
    if ( armed and not g ) ERROR
    if ( f ) armed = true
    return armed
}
```

We embed it following each statement where either *f* or *g* is at the left hand side of an assignment. Clearly that **Check_NEXT** encounters ***ERROR*** if and only if *f* **NEXT** *g* is violated. Note that **Check_NEXT** is a Boolean function, so we can nest in another formula. For example, *f* **NEXT** (*g* **NEXT** *h*) can be checked by

Check_NEXT_F_G(*f*, **Check_NEXT_G_H**(*g*, *h*))

The additional postfixes are to differentiate the static variable *armed* in **Check_NEXT**.

f* UNTIL *g: the semantics is that *f* holds until *g* holds, and *g* holds at some location. Similarly, we translate it to the function **Check_UNTIL**.

```
bool Check_UNTIL (f, g, start, end)
{
    static bool armed = false
    if ( armed and ((not f and not g) or end) )
        ERROR
    if ( start ) armed = true
    return armed
}
```

We embed it at each location where either *f* or *g* is at the left hand side of an assignment, and the end of the entire code. The last occurrence of **Check_UNTIL** will be invoked with *end* = *true*. The parameter *start* indicates when we start to check the property.

6. Test case generation

BLAST can generate positive test cases to exhibit the properties [7] if the properties hold. By applying the negative test case generation in [24], we can generate corresponding negative test cases. The negative test cases will strengthen the validation of the composite WS in the sense that it not only exhibits the desired properties, but also does not exhibit any properties not desired.

The typological algorithm [24] requires that we know the complete positive condition to generate negative test cases. The test vector generated by one counterexample may contain only partial information and hence is not complete. BLAST may generate multiple copies of a node in the control flow for different counterexamples. Each copy will be associated with a *region*, which is the collection of predicates that rule out the corresponding counterexample. We aggregate all the regions returned by BLAST to generate the complete positive conditions. These conditions are used for selectively generating positive test cases and negative test cases by the technique proposed in [24].

Another problem is in the PDDL, there is a lack of connection between the predicate and the data in the PDDL. We propose a scheme similar to the *effect* operator in PDDL to define the connection. The syntax is as follows.

effect: (*when* ([*not*] <variable> <relop> <value>) <predicate>) [*type* <type>]

The *effect*, *when*, *not*, and *type* are keywords. Note that the *not* is optional. And <variable> is a variable name, <value> is a numerical or string value, <relop> is one of =, >, >=, <, and <=, <predicate> is the pair of predicate name and truth value, and the <type> defines the connection type. The type specification is optional. Currently, our extension supports the following types: *boundary value*, *partition*, *random*, and *plain-sample*. Each corresponds to different type of test cases. Thus when generating test cases, one can select different testing methods, such as boundary testing, partition testing, etc.

We conduct experiments on the corrected "online shopping" example presented in Section 2.1. The model checking phase verifies that the WS violates no properties, and generates the complete positive test vectors. Then we selectively generate negative test cases, as listed in Table 2. We also categorize positive test cases according to criticality.

Table 2 Test cases selection

	Proximity		Degree		Total	Pass %
Posi-tive	1	7 cases	2	3 cases	7 cases	100%
			1	4 cases		100%
Nega-tive	1	8 cases	3	1 cases	9 cases	100%
			1	7 cases		100%
	2	1 cases				100%

Two factors affect the criticality of a test case, the proximity to the opposite class, i.e., positive cases to negative cases and vice versa, and the degree of proximity, i.e., the number of proximate opposite cases. The selection enables a time-based testing plan. If we have less time, we test only on the most critical test cases. Test cases are ordered according to their criticality from top to bottom in Table 2.

7. Conclusion

This paper proposed an integrated process to automatically translate OWL-S specification of composite WS into a C-like specification language that can be processed by BLAST model checker, perform model checking of composite WS, generate positive and negative test cases during model checking, and test the WS using the test cases. The existing techniques applied in this integrated process do not meet the requirements of composite WS verification and testing. We extended BLAST to handle concurrent execution of threads in OWL-S. We extended OWL-S and PDDL to connect predicates and data (values and variables) to better facilitate test case generation. One example is illustrated using the this integrated process and the experiment results reveal the process was effective.

8. References

[1] "Business Process Execution Language for Web Services Version 1.1", available at: http://www-106.ibm.com/developerworks/library/ws-bpel/, 2003.

[2] "KIF, Knowledge Interchange Format: Draft proposed American national Standard (dpans)", Technical Report 22/98-004, ANS, 1998.

[3] "OWL-S: Semantic Markup for Web Services", available at: http://www.daml.org/services/owl-s/1.1B/owl-s/owl-s.html.

[4] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[5] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft", Technical Report: MSR-TR-2004-8, Microsoft Research, 2004.

[6] T. Ball and S. K. Rajamani. "Automatically Validating Temporal Safety Properties of Interfaces", In Proceedings of the 8th International SPIN Workshop on Model checking of Software, 2001, pp. 103-122.

[7] D. Beyer, A. J. Chlipala, and R. Majumadr, "Generating Tests from Counterexamples", In Proceedings of the 26th International Conference on Software Engineering, 2004, pp. 326-335.

[8] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 2002.

[9] R. Cleaveland, and S. Sims. "The NCSU concurrency workbench", In Proceedings of the 8th Conference on Computer Aided Verification, volume 1102 of Lecture Notes in Computer Science, 1996, pp. 394-397.

[10] X. Fu, T. Bultan, and J. Su, "Analysis of Interacting BPEL Web Services", In Proceeding of the 13th International World Wide Web Conference, 2004, pp. 621-630.

[11] X. Fu, T. Bultan, and J. Su, "Model Checking Interactions of Composite Web Services", Technical Report 2004-05, Computer Science Department, University of California at Santa Barbara, 2004.

[12] M. Ghallab. "PDDL – The Planning Domain Definition Language V. 2". Technical Report, report CVC TR-98-003 / DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[13] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction", In Proceedings of the 29th Annual Symposium on Principles of Programming Languages, 2002, pp. 58-70.

[14] T. A. Henzinger, R. Jhala, and R. Majumdar, Race Checking by Context Inference, In Proceedings of ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, 2005, pp. 1-13.

[15] G. Holzmann, *The Spin Model Checker*, Addison-Wesley, 2003.

[16] M. Koshkina, and F. van Breugel, "Modelling and Verifying Web Service Orchestration by Means of the Concurrency Workbench", to appear in ACM SIGSOFT Software Engineering Notes.

[17] R. Milner, *A Calculus of Communicating Systems*, LNCS-92, Springer-Verlag, 1980.

[18] M. Paolucci, N. Srinivasan, K. Sycara, and T. Nishimura, "Towards a Semantic Choreography of Web Services: from WSDL to DAML-S", available at: http://pericles.cimds.ri.cmu.edu:8080/wsdl2owls/wsdl2damls.pdf

[19] S. Qadeer, S. K. Rajamani and J. Rehof , "Summarizing Procedures in Concurrent Programs", In Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 2004, pp. 245-255.

[20] W. Tsai, Z. Cao, Y. Chen, R. Paul, "Web Services-based Collaborative and Cooperative Computing", to appear in Workshop on Cooperative Computing, Internetworking, and Assurance Chengdu, China, April 5, 2005.

[21] W. Tsai, Y. Chen, Z. Cao, X. Bai, H. Huang, and Paul, "Testing Web Services Using Progressive Group Testing", Advanced Conference on Content Computing, Zhenjiang, November, LNCS 3309, 2004, pp. 314-322.

[22] W. Tsai, R. Paul, Z. Cao, L. Yu, A. Saimi, and B. Xiao, "Verification of Web Services Using an Enhanced UDDI Server", Proc. of IEEE WORDS, 2003, pp. 131-138.

[23] W. Tsai, W. Song, R. Paul, Z. Cao, H Hunag, "Services-Oriented Dynamic Reconfiguration Framework for Dependable Distributed Computing", COMPSAC, September 2004, pp. 554-559

[24] W. Tsai, X. Wei, Y. Chen, B. Xiao, R. Paul, and H. Huang, "Developing and Assuring Trustworthy Web Services", to appear in Proceedings of the 7th International Symposium on Autonomous Decentralized Systems, Chengdu, April 2005.

[25] C. Walton, "Model Checking Multi-Agent Web Services", In Proceeding of AAAI Spring Symposium on Semantic Web Services, Stanford, 2004.