Logic-based Verification for Web services Composition with TLA

Hongbing Wang
Dept. of Computer Science & Engineering
Southeast University, China
hbw@seu.edu.cn

Li Li
Faculty of Computer and Information Science
Southwest University, China
lily@swu.edu.cn

Chen Wang, Zuling Kang
Dept. of Computer Science & Engineering
Southeast University, China
{cwang, zlkang}@seu.edu.cn

Dongxi Liu, Jemma Wu, Athman Bouguettaya
CSIRO ICT Center, Australia
{dongxi.liu, jemma.wu, athman.bouguettaya}@csiro.au

Abstract

Web services composition is an emerging paradigm for enabling application integration within and across organizational boundaries. Effectively verifying service composition to comply with the requirements is challenging. Verifying the composed service against certain properties is discussed in this paper. TLA (Temporal Logic of Actions) is introduced to enable effective verification. In particular, algorithms to transforming Web services from OWL-S to TLA are proposed. We then show that automatically verifying Web service behaviours can be achieved by using TLC (TLA model checking tool). A case study is given to illustrate the transformation and verification.

1 Introduction

Web services composition is an emerging paradigm for enabling application integration within and across organizational boundaries. Although there is growing evidence that service composition attracts increasing attention in the research community [2], verifying service composition is yet to be fully exploited. Verification plays a rather important role in the success of practical applications, because it checks the composed service against certain properties in order to signify whether the used model complies with the requirements. Unless some kind of mathematical foundation is used in service modelling, verifying service composition effectively would be very challenging.

Among the widely used semantic Web service models, OWL-S is well known, as it has potential to achieve automatic Web service composition and interoperation. It is also supported by well defined documents. However, OWL-S does not provide a way to verify correctness of the model [13]. Often, the path to follow is to compile the service model into some formulae which can be feed to an automatic verification tool together with the properties to be checked, and then, let the tool do the checking [2]. Significant work on the verification of BPEL design has been reported [5, 16], but less has been done on OWL-S verification [2, 13].

In this paper, we propose to use Temporal Logic of Actions (TLA) to verify OWL-S service composition effectively. There are four reasons why we are planning to use TLA to achieve our goal: (1) Unlike earlier sequential systems, TLA, invented by Lamport [9], is a logic for reasoning in concurrent systems. It specifies program properties as a logical formula and then proves the formula in the logic; (2) TLA makes it practical to describe a system with a single formula [10], which was unable to be achieved with Pnueli's temporal logic [17] in practice; (3) TLA is especially well suited to describing asynchronous systems, such as Web applications; and (4) it is evident that TLC [11], the model checker of TLA, is able to complete checking of real-time specifications much faster [3, 8], and in some cases it is better than Uppaal [12]. Refer to [11] for a comparison of TLC, SPIN [6], and SMV [14]. With these in mind, we expect that TLA is able to enhance Web services composition verification practically.

Web services are Web components from the perspective of software engineering. Thus, it is possible to verify the

978-1-4244-5299-6/09/$26.00 (c)2009 IEEE

composed service (or the composed system) using formal methods and tools. In this paper, we do not distinguish between composed Web services and composed systems unless otherwise specified. Similarly, a TLA document, or a specification, or simply the $Spec$ are regarded as the same in the context of model checking.

The rest of the paper is organised as follows. Section 2 briefly introduces TLA and its formulae. Section 3 presents some background of the OWL-S service model in TLA. Particularly, the OWL-S process model is introduced with a service composition scenario. The way to transfer services from OWL-S to TLA formulae is examined thoroughly in Section 4. Section 5 is a proof-of-concepts prototype. Section 6 provides an overview of related work. Finally, Section 7 is the conclusion.

2 Temporal Logic of Actions

TLA [9] was invented by Lamport for specifying and reasoning in concurrent and reactive systems. TLA allows the specification of the correct behaviors of a system. In TLA, a behaviour represents an execution of the system as an infinite sequence of states where a state is an assignment of values to *state variables*. An action formula expresses the relationship between the values of variables in two successive states, also called a *step*. A TLA $Spec$ is often written in the form $Spec \triangleq Init \wedge \Box[\mathcal{A}]_v \wedge Temporal$, where $Init$ is a state predicate that specifies the initial values of the variables, $\mathcal{A}$ is an action formula representing the *next-state* relationship typically written as a disjunction of possible moves. The temporal operator $\Box$ means "always". The subscript v is defined as n-tuple of state variables of interest, and $Temporal$ represents temporal formulae that usually specify liveness conditions. In TLA, the *liveness* property means that something does eventually happen; and the *safety* property asserts that something may never happen.

A TLA $Spec$ is said to be valid iff (if and only if) it is satisfied by all possible behaviours. Syntactically, a TLA formula to be used in this paper has the following forms. Refer to [9] for general details about TLA.

D	$\Box[\mathcal{A}]_f$	$\Box F$	$\exists x : F$
$\neg F$	$F \wedge G$	$F \vee G$	$F \Rightarrow G$
$F \equiv G$	$WF_f(\mathcal{A})$	$SF_f(\mathcal{A})$	$\diamond F$

where D is a *predicate*, f is a *state function*, $\mathcal{A}$ is an *action* (to move to the next step), x is a *variable*, and F and G are *TLA formulae*. The Boolean operators have their usual meanings. Other operators are described as follows.

- D: Satisfied by a behaviour iff D is true for (the values assigned to variables by) the initial state.
- $\Box[\mathcal{A}]_f$: Satisfied by a behaviour iff every step satisfies $\mathcal{A}$ or leaves f unchanged.
- $\Box F$: (F is always true) Satisfied by a behaviour iff F is true for all suffixes of the behaviour.
- $\exists x : F$: Satisfied by a behaviour iff there are some values that can be assigned to x to produce a behaviour satisfying F.
- $WF_f(\mathcal{A})$: (Weak fairness of $\mathcal{A}$) Satisfied by a behaviour iff $\mathcal{A} \wedge (f' \neq f)$ is infinitely often not enabled, or infinitely many $\mathcal{A} \wedge (f' \neq f)$ steps occur, where f' is a primed variable.
- $SF_f(\mathcal{A})$: (Strong fairness of $\mathcal{A}$) Satisfied by a behaviour iff $\mathcal{A} \wedge (f' \neq f)$ is only finitely often not enabled, or infinitely many $\mathcal{A} \wedge (f' \neq f)$ steps occur.
- $\diamond F$: (F is eventually true) Defined to be $\neg\Box\neg F$.

3 OWL-S Model in TLA

Suppose a user sends a request to an online bookshop service (i.e. a composite service) for the cheapest book with a given book title. We assume that this composite service is able to find the cheapest price (in local currency) from available sites (e.g. Amazon and Barnes & Noble). This activity is depicted as an OWL-S *process*(as shown in Fig. 2), which is a specification of the ways a client may interact with a service. A process can be described by inputs, outputs, preconditions, and effects (IOPE for short) properties. A service can be modelled as a process in OWL-S.

Classes of processes in OWL-S include atomic, composite and simple processes. Atomic and composite processes are our foci in this paper. Atomic processes are directly executable with no sub-processes. They are executed in a single step. On the other hand, composite processes are subdivided into other processes, which are either non-composite or composite processes. Usually the subdivision of a composite process is specified by using control constructs such as `sequence` and `if-then-else`. As such, any composite process can be considered as a tree structure whose nonterminal nodes are labelled with control constructs, each of which has child nodes specified using components (e.g. the nodes in shaded boxes in Fig. 3). The terminal nodes (i.e. the leaves of the tree) are executable atomic processes.

In TLA, a *behaviour* is defined as a sequence of states. We specify an OWL-S modelled composed system by specifying a set of possible behaviors - those representing a correct execution of the system within OWL-S processes. Ideally, the transition of states initiates from *Start* and terminates at *End* (dummy *Start* and *End* are used here). They are not part of the process, but instead are introduced to make it easy to describe the process in TLA. Control constructs in the OWL-S process include: *Sequence, Split, Split-Join, Choice, Any-Order, If-Then-Else, Iterate,*

Repeat-While, and *Repeat-Until*. In this paper, we introduce the term 'point' to depict different points immediately before/after the corresponding constructs. They are named by a term involving the word point in line with different constructs (e.g. split point, split-join point). Each of the OWL-S constructs will be defined by *actions* in TLA. Below, we first define two variables P and Q, respectively.

Definition 1: A set P is a collection of variables. An element of P corresponds to a service that is being executed currently.

Definition 2: A set Q is a collection of variables. An element of Q corresponds to a service that has been executed.

Initially, let *Start*$\in P$, $Q = \emptyset$.

Definition 3: **Sequence**

$sequence(s_i, s_j) \triangleq (s_i \in P) \wedge (Q' = Q \cup \{s_i\}) \wedge (P' = P \cup \{s_j\} \setminus \{s_i\})$, where the symbol $\triangleq$ means equals by definition. According to the definition, s_i and s_j are executed sequentially.

Definition 4: **Split**

$split(s_i, s_{i1}, s_{i2}, ..., s_{ik}) \triangleq (s_i \in P) \wedge (Q' = Q \cup \{s_i\}) \wedge (P' = P \cup \{s_{i1}, s_{i2}, ..., s_{ik}\} \setminus \{s_i\})$, where s_i is the immediately preceding service before split point, and s_{i1}, s_{i2}, ..., s_{ik} are services to be executed in parallel immediately after the split point.

Definition 5: **Split-Join**

In reverse, $split\text{-}join(s_{i1}, s_{i2}, ..., s_{ik}, s_j) \triangleq (\{s_{i1}, s_{i2}, ..., s_{ik}\} \subseteq P) \wedge (Q' = Q \cup \{s_{i1}, s_{i2}, ..., s_{ik}\}) \wedge (P' = P \cup \{s_j\} \setminus \{s_{i1}, s_{i2}, ..., s_{ik}\})$, where $s_{i1}, s_{i2}, ..., s_{ik}$ are services to be executed in parallel immediately before the split-join point.

Definition 6: **Choice**

$choice(s_i, s_{i1}, s_{i2}, ..., s_{ik}) \triangleq (branch_1 \wedge \neg branch_2 \wedge ... \wedge \neg branch_n) \vee (\neg branch_1 \wedge branch_2 \wedge ... \wedge \neg branch_n) \vee ... \vee (\neg branch_1 \wedge \neg branch_2 \wedge ... \wedge branch_n)$, where s_i is a service immediately before the choice point, whereas $s_{i1}, s_{i2}, ..., s_{ik}$ are services immediately after the point, and $branch_k \triangleq (s_i \in P) \wedge (Q' = Q \cup \{s_i\}) \wedge (P' = P \cup \{s_{ik}\} \setminus \{s_i\}), k \in \mathbb{N}$. $Choice$ is a logic disjunction, and only one branch will be chosen exclusively.

Definition 7: **Repeat**

$repeat \triangleq (s_i \in P) \wedge (Q' = Q) \wedge (P' = P)$. This implies that service s_i is executed repeatedly.

Definition 8: **If-Then-Else**

$if\text{-}then\text{-}else(condition, s_i, s_j, s_k) \triangleq (s_i \in P) \wedge (Q' = Q \cup \{s_i\}) \wedge (if\ (condition)\ then\ (P' = P \cup \{s_j\} \setminus \{s_j\})\ else\ (P' = P \cup \{s_k\} \setminus \{s_i\}))$, where the service s_j is executed when the *condition* is *true*, s_k otherwise. There is one branch that will be selected at each execution. From this point of view, the mechanism of $if\text{-}then\text{-}else$ is similar to that of $choice$ where only one branch is chosen and executed exclusively. Thus the construct $if\text{-}then\text{-}else$ can be treated as a special case of $choice$ with two branches.

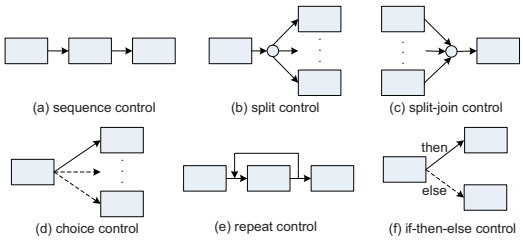

Figure 1. OWL-S process graphic notations

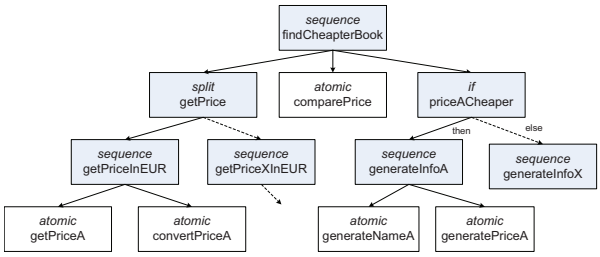

Figure 3. Online bookshop service represented as a tree structure

Before we move on, it is worth pointing out that a process in OWL-S is often viewed at different levels of granularity. It is sometimes referred to as a "black box" view. As such, *any-order* can be treated semantically equal to $split$.

Based on the above definitions, we present graphic notations of OWL-S control constructs (Fig. 1). With these graphic notations, the online bookshop service is redescribed in Fig. 2.

The online bookshop service example can also be described as a tree structure shown in Fig. 3. Please note that units in Euro (EUR) are used in this example.

4 Transforming OWL-S to TLA

In Section 2, a TLA specification is defined as $Spec \triangleq Init \wedge \Box[\mathcal{A}]_v \wedge Temporal$. In this section, we detail these formulae based on the given example. For simplicity, sample services such as Amazon (represented by symbol A) and Barnes & Noble (B) are used for illustration.

Generally, a specification in TLA consists of three parts. Firstly, constants used by the protocol are declared followed by variables. We make the *Spec* easier to understand by explicitly defining what values the variables can assume in a behaviour that satisfies the *Spec* by defining the `type invariant`, or simply put, `TypeInvariant` [10]. After that, we define *state functions*, *state predicates*, and finally assert a theorem `Spec`$\Rightarrow \Box$`TypeInvariant`. Detailed transformation is as follows.

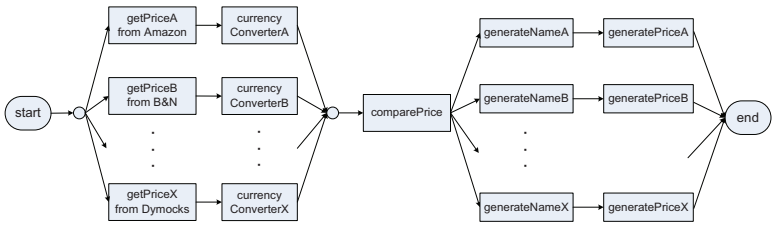

Figure 2. Online bookshop service

4.1 *Init* Predicate

As discussed, we will first define the constant. As the $Spec$ in TLA aims to describe services, we will have *Services* as parameters of the $Spec$. It is declared as:

`CONSTANT` *Services*

Variables are P and Q, with the meanings defined in Section 2 are declared as: `VARIABLE` P, Q

We then define `TypeInvariant`. The formula `TypeInvariant` will not appear as part of the *Spec*. Rather, its invariance will be asserted as a theorem. Below is the precise definition:

`TypeInvariant` $\triangleq$ P $\subseteq$(SUBSET Services) $\wedge Q \subseteq$(SUBSET Services)

It is easier to read as:

`TypeInvariant` $\triangleq$ $\wedge P \in$(SUBSET Services)
$\wedge Q \in$(SUBSET Services)

Putting them together, we have defined:

`CONSTANT` *Services*
`VARIABLE` $P,\ Q$
`TypeInvariant` $\triangleq$ $\wedge P \in$(SUBSET Services)
$\wedge Q \in$(SUBSET Services)

The *Init* predicate is straightforward. Let *Start* be the state being executed currently, then we have:

$$\begin{aligned} Init \triangleq\ & \wedge(P = \{\text{“}Start\text{”}\}) \\ & \wedge(Q = \{\}) \end{aligned}$$

4.2 Next-state Actions

According to the example, the $Spec$ consists of $getPrice$, $findCheaperPrice$, $comparePrice$, $convertPrice$, $generateInfo$, $toEnd$, and $loop$ actions. Therefore, the *next-state* action $\mathcal{A}$ is defined as the logical disjunction of these actions. More specifically, $\mathcal{A}$ is written as:

$$\begin{aligned} \mathcal{A} \triangleq\ & \vee getPrice \\ & \vee comparePrice \\ & \vee findCheaperPrice \\ & \vee convertPrice \\ & \vee generateInfo \\ & \vee toEnd \\ & \vee loop \end{aligned}$$

Let us now define these state predicates individually.

$$\begin{aligned} getPrice \triangleq\ & \wedge(\text{“}Start\text{”} \in P) \\ & \wedge(Q' = Q \cup \{\text{“}Start\text{”}\}) \\ & \wedge(P' = (P \setminus \{\text{“}Start\text{”}\}) \\ & \quad \cup\{\text{“}getPriceA\text{”}, \text{“}getPriceB\text{”}\}) \end{aligned}$$

The obtained price will be converted into a required currency (e.g. EUR). The following definition is based on the assumption that $priceA$ is cheaper than $priceB$ and is required to be converted to EUR. At the end, $convertPriceA$ action is defined as follows:

$$\begin{aligned} convertPriceA \triangleq\ & \wedge(\text{“}convertPriceA\text{”} \in P) \\ & \wedge(Q' = Q \cup \{\text{“}convertPriceA\text{”}\}) \\ & P' = (P \setminus \{\text{“}convertPriceA\text{”}\} \\ & \quad \cup\{\text{“}comparePrice\text{”}\}) \end{aligned}$$

$comparePrice$ is defined as comparing two prices. It is defined as:

$$\begin{aligned} comparePrice \triangleq\ & \wedge(\text{“}getPriceA\text{”} \in P) \\ & \wedge(\text{“}getPriceB \in P) \\ & \wedge(Q' = Q \cup \{\text{“}getPriceA\text{”}, \text{“}getPriceB\text{”}\}) \\ & \wedge(P' = (P \setminus \{\text{“}getPriceA\text{”}, \text{“}getPriceB\text{”}\}) \\ & \quad \cup\{\text{“}comparePrice\text{”}\}) \end{aligned}$$

As discussed, the *if-then-else* construct will be treated as a special case of *Choice*. In this example, we define two branches accordingly.

$$\begin{aligned} BranchA \triangleq\ & \wedge(\text{“}comparePrice\text{”} \in P) \\ & \wedge(Q' = Q \cup \{\text{“}comparePrice\text{”}\}) \\ & \wedge(P' = (P \setminus \{\text{“}comparePrice\text{”}\}) \\ & \quad \cup\{\text{“}generateInfoA\text{”}\}) \end{aligned}$$

$$\begin{aligned} BranchB \triangleq\ & \wedge(\text{“}comparePrice\text{”} \in P) \\ & \wedge(Q' = Q \cup \{\text{“}comparePrice\text{”}\}) \\ & \wedge(P' = (P \setminus \{\text{“}comparePrice\text{”}\}) \\ & \quad \cup\{\text{“}generateInfoB\text{”}\}) \end{aligned}$$

With the defined branches, we define action $findCheaperPrice$ as:

$$\begin{aligned} findCheaperPrice \triangleq\ & \vee(BranchA \wedge \neg BranchB) \\ & \vee(\neg BranchA \wedge BranchB) \end{aligned}$$

Either Amazon or B&N book information will be generated depending on the given conditions. The following definition is based on the fact that the cheaper price is found at Amazon. Predicate $generateInfoB$ can be defined similarly.

$generateInfoA \triangleq$ $\wedge(\text{“}generateInfoA\text{”} \in P)$
$\wedge(Q' = Q \cup \{\text{“}generateNameA\text{”}\})$
$\wedge(P' = (P \setminus \{\text{“}generateNameA\})$
$\cup\{\text{“}generatePriceA\text{”}\})$

No matter which branch it takes, the process will end up at *End*. Also because the branch within *Choice* is executed exclusively, we need to define some other predicates in the $Spec$.

$joinA \triangleq$ $\wedge(\text{“}generatePriceA\text{”} \in P)$
$\wedge(Q' = Q \cup \{\text{“}generatePriceA\text{”}\})$
$\wedge(P' = (P \setminus \{\text{“}generatePriceA\})$
$\cup\{\text{“}End\text{”}\})$

Similarly, $joinB$ predicate is defined as:

$joinB \triangleq$ $\wedge(\text{“}generatePriceB\text{”} \in P)$
$\wedge(Q' = Q \cup \{\text{“}generatePriceB\text{”}\})$
$\wedge(P' = (P \setminus \{\text{“}generatePriceB\})$
$\cup\{\text{“}End\text{”}\})$

At the end, predicate $toEnd$ is defined as:

$toEnd \triangleq$ $\vee(joinA \wedge \neg joinB)$
$\vee(\neg joinA \wedge joinB)$

As we expect that the $Spec$ fulfills the whole search space strategy, we need to define predicate $loop$ to guarantee this. It is:

$loop \triangleq$ $\wedge(\text{“}End\text{”} \in P)$
$\wedge(Q' = \{\})$
$(P' = \{\text{“}Start\text{”}\})$

With these predicates, the $Spec$ is defined as:

$Spec \triangleq$ $\wedge Init$
$\wedge\Box[\mathcal{A}]_{P,Q}$ with $WF_{P,Q}(\mathcal{A})$ to guarantee that
$\wedge WF_{P,Q}(\mathcal{A})$

P and Q can infinitely change in line with $\mathcal{A}$.

The last thing is the asserted theorem: $THEOREM\ Spec \Rightarrow \Box TypeInvariant$.

5 Prototype

We develop a prototype to facilitate the transformation of composite services from OWL-S to TLA. Core modules, the shaded parts shown in Fig.4, include *Parser*, *T2P* (i.e. the transformation from a tree structure to an OWL-S process model) and *P2L* (i.e. the transformation from an OWL-S process model to TLA logic formulae). Finally, after the TLA document is generated, it is sent to TLC to be verified. In the following, we first introduce several OWL-S APIs used in the prototype, followed by the discussion of each individual module.

5.1 OWL-S APIs

Some OWL-S APIs are used to implement the above modules. The two most popular efforts are from the University of Maryland Institute for Advanced Computer Studies (*http://www.mindswap.org/2004/owl-s/api/*) and Carnegie Mellon University (*http://www.semwebcentral.org/projects/owl-s-api/*). The former is chosen based on its performance in terms of extensibility and flexibility. Table 1 lists APIs used in the prototype. Refer to the link *http://www.mindswap.org/2004/owl-s/api/doc/javadoc/* for more detail.

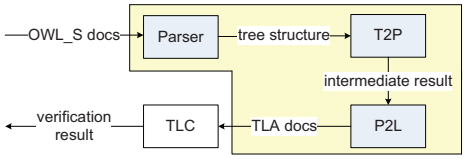

Figure 4. Core modules in the prototype

Table 1. OWL-S APIs used in prototype

```
Interface OWLOntology
Interface OWLKnowledgeBase
OWLOntology OWLKnowledgeBase.read(URI)
Service OWLKnowledgeBase.getService()
Service OWLKnowledgeBase.readService(URI)
Process Service.getProcess()
ControlConstruct CompositeProcess.getComposedOf()
ProcessList ControlConstruct.getAllProcesses()
```

5.2 Parser Module

The *Parser* module has an OWL-S document as an input and the tree structure of this OWL-S document as an output. Fig. 3 is a sample output. The algorithm is given below (see Table 2).

5.3 T2P Module

The *T2P* module has a tree structure as an input and an intermediate process model as an output. The main idea is to find the immediately preceding and succeeding nodes of a given node. Table 3 is the algorithm with two constructs considered.

5.4 P2L Module

The *P2L* module has a process structure as an input and a TLA document as an output. In this algorithm (Table 4), the input is treated as a directed acyclic graph (DAG) G. The process in essence is to carry out a valid topological sorting of the input, then to generate a TLA formula for each node in the sorted list. We defined two data structures: list V and list L, with L the topologically sorted list. An example of control construct translation can be found in Section 4.

Table 2. *Parser* **module**

```
(01) parser (Process proc) {
(02)   define tree nodes and a stack;
(03)   push these nodes into the stack;
(04)   while (the stack is non-empty) do {
(05)     the top element from the stack is analysed;
(06)     if (the current node is an AtomicProcess)
(07)       go back to (04);
(08)     for each of CompositeProcess do {
(09)       call OWL-S API to return a list of all the Process
           objects used either directly or indirectly in
           this construct;
(10)       develop ancestor - descendant relationships of
           nodes;
(11)       push them into the stack accordingly;
(12)    }// end for
(13)  }//end while
(14) }
```

Table 3. *T2P* **module (with two constructs)**

```
(01) T2P (Process proc) {
(02)   search non-terminal nodes using Breadth-First-Search
       with first priority given to Sequence construct;
(03)   case 1 (sequence):
(04)     let current_seq = the current Sequence node;
(05)     if (immediate preceding node of this Sequence exists)
(06)     let pre_seq = this node;
(07)     let first_child = the utmost left child of current_seq;
(08)     let each child of current_seq point to its immediate
           right sibling until no sibling;
(09)     let pre_seq = first_child;
(10)   break;
(11)   case 2 (split):
(12)     let current_split = the current Split node;
(13)     let pre_split = the immediate preceding node of this
           split node, where
           pre_split = Start in the extreme case;
(14)     let post_split = the immediate succeeding node of this
           split node, where
           post_split = End in the extreme case;
(15)     let pre_split point to all children of current_split;
(16)     let all children of current_split point to post_split;
(17)   break;
(18) }
```

Table 4. *P2L* **module**

```
(01) P2L (Process proc) {
(02)   let V the list of nodes which have no incoming edges
(03)   let the sorted list L = ∅;
(04)   while (V is non-empty) do {;
(05)     remove a node v from V;
(06)     insert v into L;
(07)     for (each node n with an edge e from v to n) do {
(08)       remove edge e from graph G;
(09)       if n has no other incoming edges then
(10)         insert n into V;
(11)     } end for
(12)   } end while
(13)   while (L is non-empty) do {
(14)     remove a node v from L;
(15)     translate v into a TLA formula;
(16)   } end while
(17) }
```

5.5 Experimental Results

TLC is the model checker of TLA. It can find errors in specifications based on TLA/TLA+ [10]. The most effective way to find errors in a specification is to verify that it satisfies the properties it should. TLC can check that the specification satisfies a large class of TLA formulae.

By default, TLC normally checks for (1) "*Silliness*" errors. In other words, it checks a silly expression whose meaning is not determined by the semantics of TLA/TLA+; (2) *Deadlock*. Deadlock freedom is a particular property that a specification is required to satisfy. With TLA, we often run TLC to check the *liveness* property in addition to the *safety* property. The core modules (Fig. 4) and the TLC model checker are integrated together in our prototype. The online bookshop example and its results are illustrated in Figures 5, 6, 7 (due to space limit, only the "Output of Model Checking Results" are presented in Figures 6, 7). On the top of the left hand side of these figures is the OWL-S description of the example, the bottom is the generated TLA formulae corresponding to the above OWL-S document; on the right hand side of these figures is the result of running TLC model checker against the generated TLA formulae.

5.5.1 Checking Safety

With the generated TLA document from the chain of *Parser*, *T2P* and *P2L*, we run TLC to first check the *safety* property with the output shown in Fig. 5(RHS).

Obviously, deadlock is absent from the $Spec$. Now let us

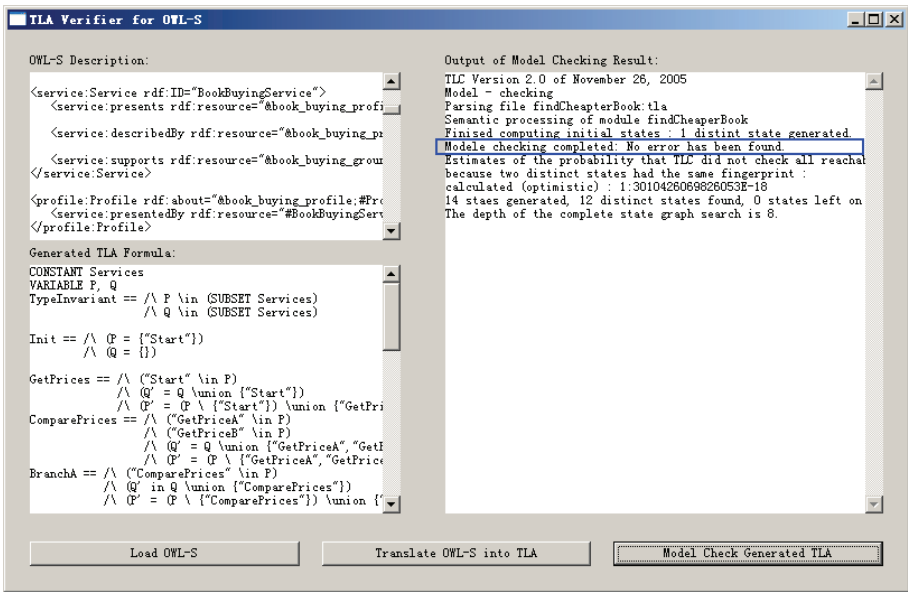

Figure 5. *Safety* **property checking - absence of deadlock**

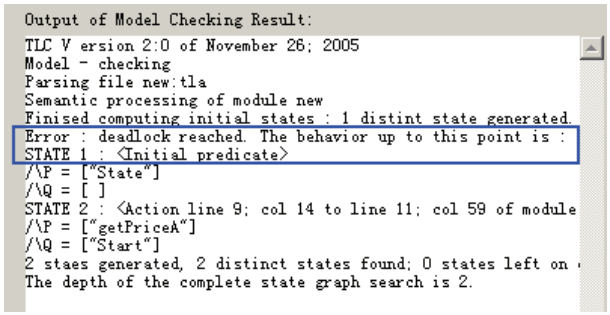

Figure 6. *Safety* **property checking - deadlock**

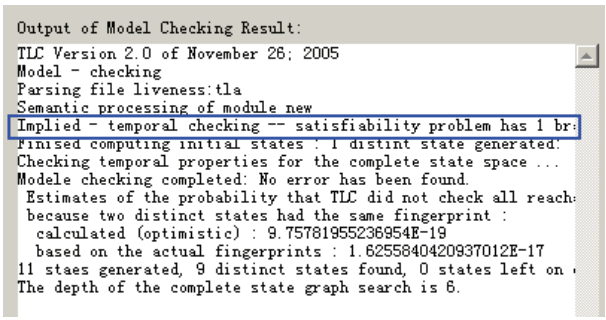

Figure 7. *Liveness* **property checking**

modify the *Spec* slightly. What we want to do is to block the service *getPriceB*. This will influence the composed service as it is needed by the service *comparePrice*. We run TLC again with the modified *Spec*. Fig. 6 is the result. The output from TLC reveals that the model checking is not completed due to errors. It also gives us detailed information on where to locate the error and clues on how to correct it.

5.5.2 Checking Liveness

It is always expected that the composite service will terminate eventually. After checking the default properties, we are interested in checking the liveness property of a given TLA document. To do this, we add two more predicates *done* and *achieve*. Predicate *done* means that *End* node is being executed, and predicate *achieve* indicates that *done* will be true eventually. Their definitions are as follows:

$$done \triangleq \quad (\text{“}End\text{”} \in P)$$

$$achieve \triangleq \quad \Diamond done$$

We run TLC to check the *liveness* property of the *Spec* (it is the file *liveness.tla* in this example) with the output shown on in Fig. 7. The sentence ‘Implied-temporal checking–satisfiability problem has 1 branches’ from the output implies that the *Spec* satisfies the defined liveness property. That is, the *done* does eventually happen.

6 Related Work

The majority of service composition verification is concentrated on BPEL expression [2]. The general path is to compile the BPEL design into some formulae which can be

used as the input by some automatic verification tools [16]. The tools then do the checking with the representation of the properties (e.g., termination and absence of deadlock) to be checked.

Compared to the BPEL modelling approaches, less attention has been given to OWL-S [2, 13]. Following are the most closely related work. Huang et al. [7] propose to translate the service from the OWL-S model to a C-link language which is the input of the model checker BLAST. Narayanan et al. [15] propose a more widely used method, in which services in DAML-S (currently OWL-S) are first translated into first-order logic and then to Petri nets to be verified. An improvement of Narayanan's seminal work in three directions can be found in [1].

Petri nets are one of the formal verification approaches often used in service composition and composition correctness verification [16, 4]. Petri nets can be used to verify some properties (e.g. deadlock freedom), but with no intention to check any temporal property of a given specification. Many extensions to Petri nets such as hierarchical colored Petri nets and time constraints Petri nets are also explored in order to well support verification of Web services. Since generating a well abstracted model is crucial to the verification of Web services, other approaches such as ISPL (Interpreted Systems Programming Language) [13] were also discussed to enable a suitable abstraction of Web services.

In this paper we are interesting in using TLA to describe Web service composition. We explore the translation from Web services of OWL-S to specifications of LTA, because it is a method of proof arising directly from the logic [9], and it can be used to describe the dynamic behaviours of services in open context.

7 Conclusion

TLA (Temporal Logic of Actions), rooted in first order logic, is introduced to provide the theoretical mechanism for verifying composed services against certain properties. Practically, algorithms to translate composed Web services from OWL-S to TLA specification are discussed. A proof-of-concepts prototype is implemented to demonstrate the effectiveness of TLA in service composition verification.

Although the transformation mechanism discussed in this paper is generic, for simplicity, we have limited our discussion to the given example. Future work aims at improving the developed transformation tools to deal with more temporal properties.

Acknowledgements

This work is partially supported by NSFC of China under No.60673175.

References

[1] M. P. Anupriya Ankolekar and K. Sycara. Spinning the owl-s process model–towards the verification of owl-s process models. In *ISWC'04 Workshop on Semantic Web Services*, Nov. 2004.

[2] G. Baryannis and M. C. et al. Overview of the state of the art in composition and coordination of services. Deliverable po-jra-2.2.1, European Community's Seven Framework Programme S-Cube consortium, 2008.

[3] B. Batson and L. Lamport. High-level specifications: Lessons from industry. In *In Proceedings of the 1st International Symposium on Formal Methods for Components and Objects*, pages 242–261, 2003.

[4] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing web service choreographies. *Electr. Notes Theor. Comput. Sci.*, 105:73–94, 2004.

[5] H. Foster and S. U. et al. Ltsa-ws: a tool for model-based verification of web service compositions and choreography. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *ICSE'2006)*, pages 771–774. ACM, May 2006.

[6] G. J. Holzmann. *The Spin Model Checker - Primer and Reference Manual*. Addison-Wesley, 2003.

[7] H. Huang and W.-T. T. et al. Automated model checking and testing for composite web services. In *ISORC'05*, pages 300–307. IEEE Computer Society, May 2005.

[8] J. E. Johnson, D. E. Langworthy, L. Lamport, and F. H. Vogt. Formal specification of a web services protocol. *Electr. Notes Theor. Comput. Sci.*, 105:147–158, 2004.

[9] L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[10] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[11] L. Lamport. Real-time model checking is really simple. In D. Borrione and W. J. Paul, editors, *13th IFIP WG 10.5 Advanced Research Working Conference, CHARME'05*, volume 3725 of *LNCS*, pages 162–175. Springer, Oct. 2005.

[12] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International J. of Soft. Tools for Tech. Transfer*, 1(1-2):134–152, 1997.

[13] A. Lomuscio and M. Solanki. Towards an agent based approach for verification of owl-s process models. In L. A. et al., editor, *ESWC'09*, volume 5554 of *LNCS*, pages 578–592. Springer, 2009.

[14] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[15] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *WWW'02*, pages 77–88, 2002.

[16] C. Ouyang and E. V. et al. Formal semantics and analysis of control flow in ws-bpel. *Sci. Comput. Program.*, 67(2-3):162–198, 2007.

[17] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.