# Message Race Detection for Web Services by an SMT-Based Analysis

Mohamed Elwakil[1], Zijiang Yang[1], Liqiang Wang[2], and Qichang Chen[2]

[1] Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008, USA
[2] Department of Computer Science, University of Wyoming, Laramie, WY 82071, USA
{mohamed.elwakil,zijiang.yang}@wmich.edu,
{wang,qchen2}@cs.uwyo.edu

**Abstract.** The success of the cloud computing initiative is heavily dependent on realizing trustworthy Web Services. The trustworthiness of a Web Service is judged by four factors: security, privacy, reliability and business integrity. Web Services use message-passing for communication which opens the door for messages races. Messages race with each other when their order of arrival at a destination is not guaranteed and is affected non-deterministically by factors such as network latencies and scheduling variations. Message races are dangerous to Web Services because they can be unforeseen consequences of bugs, causing messages to arrive in an unexpected ordering. In this paper we present a novel approach for improving the reliability of Web Services by detecting message races using SMT-based analysis. We model a BPEL process as a Web Service Modeling Graph (WSMG). A WSMG model is then encoded into a set of SMT constraints. The satisfiability of these constraints means that message races will occur during the actual execution of the Web Service. Hence, we reduce the message race detection problem to constraint solving problem based on satisfiability modulo theories (SMT).

**Keywords:** web services, satisfiability modulo theories, symbolic analysis.

## 1 Introduction

Reliability is one of the four pillars necessary for producing trustworthy Web Services [1]. Writing reliable Web Services is difficult due to the unique challenges of this domain. In particular, Web Services are prone to concurrency errors due to 1) concurrent processing of user/service requests; and 2) complex interaction behavior resulting from diverse communication mechanisms such as synchronous and asynchronous operations. In order to develop reliable Web Services, effective testing, analysis and verification techniques must be available to address these challenges. In this paper we attack the problem of detecting message races in Web Services. Race conditions are listed among the top 25 dangerous programming errors [2]; hence, detecting them is critical for Web Services development.

Fig. 1 illustrates a simple message race. WS1, WS2, and WS3 are three Web Services. WS1 sends messages M1 and M2 to WS2 and WS3, respectively. WS3 reacts to the received message, by sending message M3 to WS2. Since M3 is sent in response to M1, WS3 would expect receiving M2 before receiving M3 as in scenario A. However, M3 may arrive at WS2 before M2 (scenario B) due to unexpected network latency between WS1 and WS2, or due to unforeseen impediment at WS1 that delays sending M2. Messages M2 and M3 are said to be racing with each other. Intuitively, two messages race with each other if either could be received first due to the unpredictability of schedulers and message delays. Message races should be detected since they may be manifestations of bugs and can cause unpredictable results.

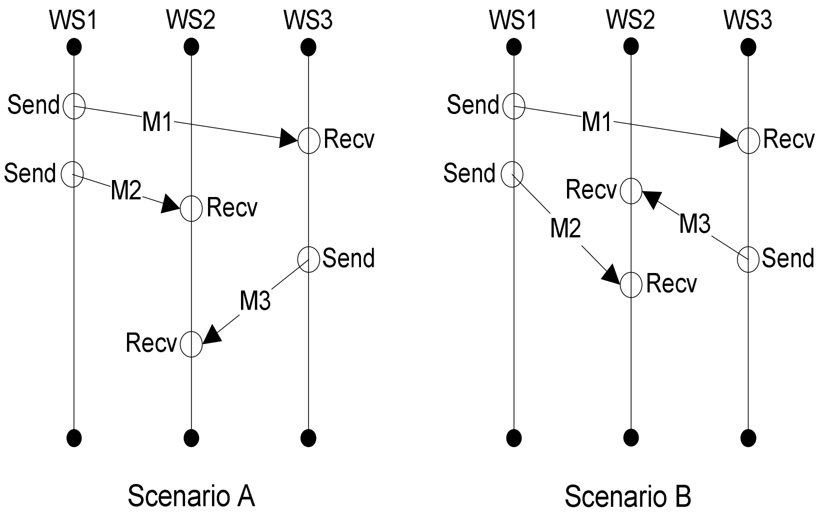

**Fig. 1.** Messages can arrive at different orderings

Unfortunately, traditional testing approaches that repeatedly execute or simulate a Web Service are not effective in detecting message races. First, such testing can be used to prove the existence of errors, but not the absence of them. Not detecting message races in multiple executions or simulations does not necessarily imply that they can't happen. To completely verify the behavior of a Web Service, all possible scenarios must be examined. Explicitly examining all possible scenarios is a taunting task, if not impossible, as the number of possible scenarios is astronomical. Also, controlled testing can't take into account unpredictable interactions that appear in the field. Second, Web Services testers have to interpret vast amount of output to determine whether there exists message races. This task alone takes non-trivial amount of time, and in many cases the output of an execution or simulation is considered correct by mistake even if there are message races. In the case where a message race is detected, the particular execution sequence that manifested the message race cannot be easily reproduced.

In this paper we present a novel approach that addresses these problems that plague traditional testing approaches. Our approach can be used to prove the absence of message races within a bound specified by the user. Unlike most other static analysis approaches that report large amount of false negatives, only real message races are reported by our approach. In order to explore the astronomical amount of possible scenarios we model Web Services using suitable classes of constraints and reducing various analysis problems to constraint solving. Fig. 2 depicts the steps of our approach. First, a BPEL [3] process is translated to a WSMG model. Second, the WSMG is encoded as an SMT [3] formula. Third, an SMT solver is used to decide the satisfiability of the formula. We chose using SMT solvers as their performance has benefited from recent significant advances in Boolean satisfiability (SAT) solvers (e.g. [5], [6], [7]) and SMT solvers (e.g. [8], [9]).

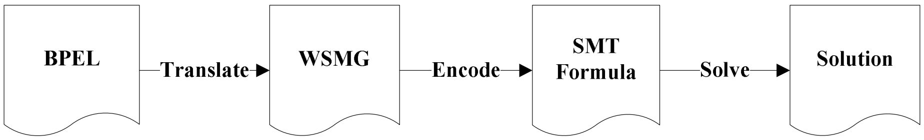

**Fig. 2.** Steps for finding messages races in a Web Service

The semantics of non-determinism such as network latency is represented implicitly by the SMT formula. The solution reported by the SMT solver offers detailed information that explains how the message race happened. Thus the bug is reproducible in the sense that the user can always simulate Web Service execution based on our bug report to obtain the same message race.

The rest of this paper is organized as follows. Section 2 briefly reviews related work. Section 3 presents our modeling language for Web Services. Section 4 details our approach to reduce message race detection problem to constraint solving problem. Section 5 describes two case studies and we conclude in Section 6 with contributions and limits of this paper.

## 2 Related Work

Netzer and Miller [10] first characterize message races and design an on-the-fly algorithm for detecting them. Afterwards, Netzer et al. [11] improve their previous approaches by using a two pass hybrid on-the-fly/post-mortem scheme, and remove artifact races that are side effects of non-determinism from the bug report. In [12], Park *et al.* present an on-the-fly detection tool, which detects message races in MPI programs by checking communication concurrency in distributed processes.

In [13], message race is identified as one type of undesirable interactions between Web Services. In that work, Web Services are modeled as feature interactions and then analyzed to discover potential message races. Zhang *et al.* proposed a Petri net based approach to detect race conditions in Web Services [14]. They subsequently presented another model checking based technique using SPIN, where the business process execution language for Web Services (BPEL4WS) is translated to Promela (SPIN model definition language) [15]. The most significant difference between previous work and ours is that most previous work uses existing languages and models that are intended for other domains such as hardware and network protocol designs. On the other hand, we use our Web Service Modeling Graph (WSMG) which is targeted for Web Services. Also, our SMT-based analysis eliminates false positives and produces a trace that facilities pinpointing the source of the message race.

Similar ideas that apply symbolic analysis to detect message races have been reported in [17] [18]. However, the technique has been applied in a different domain on MCAPI (Multicore Association Communication API), which leads to totally different modeling and encoding algorithms.

## 3 Web Service Modeling Graph

In this section we define the Web Service Modeling Graph (WSMG) that is inspired by hierarchical reactive modules [16]. WSMG is a compact representation that exhibits concurrency and control flow in Web Services.

A WSMG model represents a Web Service as a set of threads that communicate via messages over a set of channels. A thread consists of a set of sequential transitions $\mathcal{T}$. The set of transitions is defined as $\mathcal{T} \subseteq \text{P} \times \text{Q} \times \text{Guard} \times \text{Action}$, where $P$ is the state before the transition, $Q$ is the state after the transition, $Guard$ is a conditional expressions and $Action \in Asgn \cup Snd \cup Rcv \cup \{-\}$. $Asgn$ is a set of assignment statements. $Snd \in Ch \times E$ sends the result of expression $E$ over a channel in $Ch$. $Rcv \in Ch \times Var$ receives a value from a channel and saves the value to a variable in $Var$. No-op is denoted by $-$. In WSMG there are two types of channels $Ch = Ch_S \cup Ch_A$. $Ch_S$ is a set of synchronous channels, over which both send and receive are blocking. $Ch_A$ is a set of asynchronous channels, over which both send and receive are non-blocking if the buffer in a channel is not full during send action, and not empty during receive action.

We say a transition $\tau$ in thread $t$ has a $token$, denoted as $tk_t = \tau$, if it is a candidate for execution in a thread $t$. At any time one transition per thread can have the token. We say a transition $\tau$ is $fired$ if it is selected for execution. When $\tau$ is fired, the token moves to the next transition in that thread. $succ(\tau)$ denotes the next transition of transition $\tau$. In the following we explain the execution semantics of a WSMG model:

- Let $\tau = (g, v := expr)$ be a transition in thread $t$. $\tau$ can be fired if $t$ is scheduled and $tk_t = \tau \wedge g = true$ . After the firing, $tk_t = succ(\tau)$, and the assignment is executed.
- Let $\tau = (g, snd(ch, E))$ be a transition in thread $t$ that sends the value of $E$ to synchronous channel ch, and $\tau' = (g', rcv(ch, v))$ is a transition in thread $t'$ that receives to the variable $v$ from channel $ch$. Transition $\tau$ can be fired if $tk_t = \tau$, $tk_{t'} = \tau'$, $t'$ is scheduled, and both $g$ and $g'$ is true. In this case, $\tau$ and $\tau'$ are fired simultaneously. After the firings, the value of $v$ is updated by the result of $E$, and the tokens in $t$ and t' are transferred to $succ(\tau)$ and $succ(\tau')$ , respectively.
- Let $\tau = (g, asnd(ch, E))$ be a transition in thread $t$ that sends the value of $E$ to asynchronous channel $ch$. Transition $\tau$ can be fired if $t$ is scheduled, $tk_t = \tau$ , $g$=true and the buffer in $ch$ is not full. After the firing, $tk_t = succ(\tau)$, and the value of $E$ is delivered to $ch$'s buffer.
- Let $\tau = (g, arcv(ch, v))$ be a transition in thread $t$ that receives a value from asynchronous channel $ch$. Transition $\tau$ can be fired if $t$ is scheduled, $tk_t = \tau$ ,

$g$=true, and the buffer in $ch$ is not empty. After the firing $tk_t = succ(\tau)$, and the value of $v$ is updated by the removed value from $ch$.

- Let $\tau = (g, \text{fork}\ (t'))$ be a transition in thread $t$ that forks thread $t'$, and $\tau'$ be the first transition in thread $t'$. Both $\tau$ and $\tau'$ will be fired if $t$ is scheduled, $g$=true and $tk_t = \tau$. After the firings $tk_t = succ(\tau)$ and $tk_{t'} = \tau'$.
- Let $\tau = (g, \text{join}\ (t'))$ be a transition in thread $t$ that joins thread $t$ with thread $t'$, and $\tau'$ be the last transition in thread $t'$. Both $\tau$ and $\tau'$ will be fired if $tk_t = \tau \wedge tk_{t'} = \tau'$, $g$=true and $t'$ is scheduled. After the firings, $tk_t = succ(\tau)$ and $tk_{t'} = \perp$.

## 4 Symbolic Encoding

In this section we present an encoding approach that converts a given WSMG model $G$ to an SMT formula that consists of initial constraint $\iota_0(G)$, thread scheduling constraint $\chi_B(G)$, transition constraint $\tau_B(G)$ and message race constraint $\rho_B(G)$. Whether there is message race up to the predefined bound $B$ can be checked by the validity of formula 1 which is equivalent to checking the satisfiability for formula 2.

$$\iota_0(G) \wedge \chi_B(G) \wedge \tau_B(G) \wedge \rho_B(G) \tag{1}$$

$$\iota_0(G) \wedge \chi_B(G) \wedge \tau_B(G) \rightarrow \neg\, \rho_B(G) \tag{2}$$

We use the SMT solver Yices [8] to solve formula 2. If the formula is satisfiable, the solution gives a trace that leads to a message race from the initial state in $G$; otherwise, it is proved that $G$ has no message race within $B$ steps. In the following we first discuss the symbolic variables needed for the encoding, and then discuss the constraints.

### 4.1 Symbolic Variables

In our symbolic analysis we check race conditions up to a pre-defined bound $B$. For each step $i < B$, we add a fresh copy for each variable introduced in this section. That is, $var[i]$ denotes the copy of $var$ at the $i$-th step. The symbolic variables are:

- *Token variable*: In order to encode the threads interleaving semantics symbolically we identify the set of threads in a given WSMG model and introduce one token variable $tk_t$ for each thread $t$. A transition $\tau$ has a token iff $tk_t = \tau$. Before a thread $t$ is created or after it is terminated, we set $tk_t$ to be $\top$ or $\perp$, respectively.
- *Model variables*: Given a WSMG model $G$, we introduce a symbolic variable for each model variable in $G$.
- *Scheduling variable*: To model non-determinism in the scheduler, we add a symbolic variable $s$ whose domain is the set of thread identifiers. The value of $s[i]$ indicates which thread is scheduled to execute at step $i$. This is an important feature to our symbolic analysis in our approach. As in most cases the value of $s[i]$ is unspecified, the SMT solver is forced to consider the case where any thread can be scheduled to execute at step $i$.

- *Asynchronous channel buffers*: In our encoding we only consider channels with finite size buffers. Let the size of the buffer in $ch$ be $F$, we introduce $F$ symbolic variables $buf_1^{ch}$ ... $buf_F^{ch}$, each of which represents a cell in the buffer of $ch$. A buffer is treated as a queue with $buf_1^{ch}$ and $buf_F^{ch}$ as its tail and head, respectively. We use a sentinel value $stnl$ to denote a cell without valid information. The buffer in $ch$ is full iff $buf_1^{ch} \neq stnl$ and is empty iff $buf_F^{ch} = stnl$.

## 4.2 Initial Condition Constraint

The initial condition constraint $\iota_0(G)$ specifies the starting locations for each thread as well as the initial values of model variables, including the values set by the input vector.

## 4.3 Scheduling Constraint

Our approach analyzes all possible valid interleavings, and excludes invalid ones. Therefore, we add thread scheduling constraint $\chi_B(G)$ to prevent invalid interleavings from being considered. In a WSMG model, a thread $t$ must not be scheduled at step $i$ in four cases: 1) before its creation, or after its termination (formula 3), 2) when an asynchronous send transition is pending and the relevant buffer is full, or when an asynchronous receive transition is pending and the relevant buffer is empty (formula 4), 3) when a synchronous send transition is pending, and there is no corresponding pending receive transition at another thread (formula 5), or 4) when a synchronous receive transition is pending, and there is no corresponding pending send transition at another thread (formula 6). $\tau_{as}$ is an asynchronous send transition, $\tau_{ar}$ is an asynchronous receive transition, $\tau_{ss}$ is a synchronous send transition, and $Rv$ is all potential receive transitions of $\tau_{ss}$, $\tau_{sr}$ is a synchronous receive transition, and $Sd$ is all potential send transitions of $\tau_{sr}$.

$$(tk_t[i] = \top \vee tk_t[i] = \bot) \rightarrow s[i] \neq t \tag{3}$$

$$\begin{gathered}(tk_t[i] = \tau_{as} \wedge buf_1^{ch} \neq stnl) \\ \vee (tk_t[i] = \tau_{ar} \wedge buf_F^{ch} = stnl) \rightarrow s[i] \neq t\end{gathered} \tag{4}$$

$$(tk_t[i] = \tau_{ss} \wedge \bigwedge_{(t',p') \in Rv} (tk_{t'}[i] \neq p')) \rightarrow s[i] \neq t \tag{5}$$

$$(tk_t[i] = \tau_{sr} \wedge \bigwedge_{(t',p') \in Sd} (tk_{t'}[i] \neq p')) \rightarrow s[i] \neq t \tag{6}$$

The thread scheduling constraint is encoded as in formula 7, where $\chi_B[i]$ is the conjunction of the constraints listed in formulas 3, 4, 5, and 6.

$$\chi_B(G) = \bigwedge_{i=1}^{B} \chi_B[i] \tag{7}$$

## 4.4 Transition Constraint

The execution semantics of a thread is specified by the encoding of its transitions in a WSMG model. In the following we discuss the translation from transitions to SMT formulas based on the types of transitions:

1. An assignment transition in the format of $\tau = (g, v := expr)$ where $g$ is a guard, and $v := E$ assigns the results of $E$ to variable $v$ is encoded as in formula 8.

$$s[i] = t \wedge tk_t[i] = \tau \wedge g[i] \rightarrow tk_t[i] = succ(\tau) \wedge v[i+1] = E[i] \wedge \delta(\{s, tk_t, v\}) \tag{8}$$

Formula 8 states that at step $i$, $\tau$ is fired under the following conditions: Thread $t$ is selected ($s[i] = t$), $\tau$ has token ($tk_t[i] = \tau$) and guard is true ($g[i]$). Note that $g[i]$ (or $E[i]$) means that all variables in the guard $g$ (or expression $E$) are replaced by their corresponding versions at step $i$. The following updates occur at step $i+1$ when $\tau$ is fired at step $i$: the transition that succeeds $\tau$ in $t$ will have the token ($tk_t[i+1] = succ(\tau)$), the value of $v$ at step $i+1$ is the result of $E$ at step $i$ ($v[i+1] = E[i]$) and the values of all variables except $s, tk_t, v$ remain unchanged from step $i$ to $i+1$. Note that $\delta_t(S)$ means that all the variables except those listed in set $S$ keep their values step $i$ to $i+1$.

2. A synchronous send/receive transition pair in the format of $\tau = (g, snd(ch, E))$ and $\tau' = (g', recv(ch, v))$, will be encoded as:

$$s[i] = t \wedge tk_t[i] = \tau \wedge g \wedge g' \wedge tk_{t'}[i] = \tau' \rightarrow \big(tk_t[i+1] = succ(\tau) \wedge tk_{t'}[i+1] = succ(\tau') \wedge v[i+1] = E[i] \wedge \delta(\{s, tk_t, tk_{t'}, v\})\big) \tag{9}$$

3. An asynchronous send transition in the format of $\tau = (g, asnd(ch, E))$, is encoded as:

$$s[i] = t \wedge tk_t[i] = \tau \wedge g \wedge buf_1^{ch}[i] = stnl \rightarrow \Big(tk_t[i+1] = succ(\tau) \wedge \delta(\{s, tk_t, buf_F^{ch}, \dots, buf_1^{ch}\}) \wedge (buf_F^{ch}[i] = stnl ? buf_1^{ch}[i+1] = E[i] : \dots buf_2^{ch}[i] = stnl ? buf_2^{ch}[i+1] = E[i] : buf_1^{ch}[i+1] = E[i])\Big) \tag{10}$$

4. An asynchronous receive transition in the format of $\tau = (g, arcv(ch, v))$, is encoded as:

$$s[i] = t \wedge tk_t[i] = \tau \wedge g \wedge buf_F^{ch}[i] \neq stnl \rightarrow \big(tk_t[i+1] = succ(\tau) \wedge v[i+1] = buf_F^{ch}[i] \wedge \textstyle\bigwedge_{f=2}^{F}\big(buf_f^{ch}[i+1] = buf_{f-1}^{ch}[i]\big) \wedge buf_1^{ch}[i+1] = stnl \wedge \delta(\{s, tk_t, buf_F^{ch}, \dots, buf_1^{ch}, v\})\big) \tag{11}$$

5. A fork transitions in the format of $\tau = (true, \text{fork}\ (t'))$, will be encoded according to formula 12, such that $\tau'$ is the first transition in thread $t'$.

$$s[i] = t \wedge tk_t[i] = \tau \rightarrow tk_t[i+1] = succ(\tau) \wedge tk_{t'}[i+1] = \tau' \wedge \delta(\{s, tk_t, tk_{t'}\}) \tag{12}$$

6. A join transitions in the format of $\tau = (true, \text{join}\ (t'))$, will be encoded according to formula 13, such that $\tau'$ is the last transition in $t'$.

$$s[i] = t' \wedge tk_t[i] = \tau \wedge tk_{t'}[i] = \tau' \rightarrow tk_t[i+1] = succ(\tau) \wedge tk_{t'}[i+1] = \perp \wedge \delta(\{s, tk_t, tk_{t'}\}) \tag{13}$$

Let $\gamma_\tau[i]$ denote the constraint for transition $\tau$ at step $i$ and $T$ be the set of all transitions in a WSMG model, the transition constraint can be specified as

$$\gamma_B(G) = \bigwedge_{\tau \in T} (\bigwedge_{i=1}^{B} \gamma_\tau[i]) \tag{13}$$

## 4.5 Message Race Constraint

A message race occurs on a synchronous channel $ch$ when two conditions exist: a receive operation on $ch$ is pending, and two or more send operations simultaneously attempt to deliver messages on $ch$. In such case, the received message is non-deterministic. Let $SR_{ch}$ be the set of transitions with synchronous receive from $ch$ and $SS_{ch}$ be the set of transitions with synchronous send to $ch$. The constraint for synchronous message race at step $i$ on channel $ch$ can be specified as:

$$\alpha_{ch}[i] \equiv \underset{\tau \in SR_{ch}}{\exists} ((s[i] = t \wedge tk_t = \tau) \wedge \underset{\tau_1 \in SS_{ch}}{\exists} \underset{\tau_2 \in SS_{ch}}{\exists} (tk_{t1}[i] = \tau_1 \wedge tk_{t2}[i] = \tau_2\ )) \tag{14}$$

Message race happens on an asynchronous channel $ch$ if $ch$ is not full and there are multiple transitions trying to send messages over $ch$ at the same time. In such case, the message saved in the buffer of $ch$ is non-deterministic. Let $AS_{ch}$ be the set of transitions with asynchronous send to $ch$. The constraint for asynchronous message race at step $i$ on channel $ch$ can be specified as in formula 15, where $\tau_1 = (g_1, asnd(ch, E_1))$ and $\tau_2 = (g_2, asnd(ch, E_2))$ are transitions in thread $t_1$ and $t_2$, respectively.

$$\beta_{ch}[i] \equiv \left( ch \neq full \wedge \underset{\tau_1 \in AS_{ch}}{\exists} \underset{\tau_2 \in AS_{ch}}{\exists} (tk_{t1}[i] = \tau_1 \wedge tk_{t2}[i] = \tau_2) \right) \tag{15}$$

Let $ACH$ and $SCH$ be the set of asynchronous and synchronous channels in a WSMG model $G$. The message race property, up to bound $B$, can be specified by:

$$\rho_B(G) = \bigvee_{i=1}^{B} \left( \bigvee_{ch \in ACH} \alpha_{ch}[i] \vee \bigvee_{ch \in SCH} \beta_{ch}[i] \right) \tag{16}$$

## 5 Experiments

To assess the feasibility of our approach, we applied it on the stock-trading and the loan-approval case studies from the BPEL-WS 1.1 standard [2].

As shown in Fig. 3, the stock-trading case study consists of three sub-services: a quote service (SQS), a trading service (STS), and a bank service (Bank). The quote service has two threads that continuously send updated stock prices to the bank and the trading services. The trading service compares a received price to a minimum threshold and a maximum threshold. If the price is less than the minimum threshold, the trading service will send to the bank a buy request message. If the price is greater than the maximum threshold, the trading service will send to the bank a sell request message. Otherwise the trading service does nothing. The bank service updates its database when it receives new stocks prices from the quote service, and performs either selling or buying operations according to the requests received from the trading service.

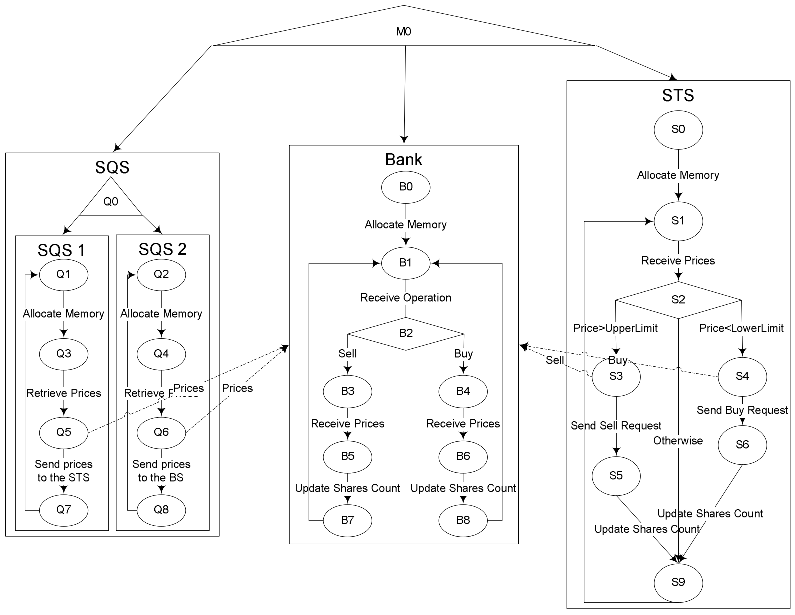

**Fig. 3.** The stock-trading Web Service

We followed the steps depicted in Fig. 3 and used Yices as the SMT solver. The solution produced by Yices indicates that a message race will occur when two quote services send prices-update messages to the bank service.

Table 1 shows the output of Yices which is an interpreted partial valuation to the symbolic variables in the SMT formula. In particular, we show the values of the token variables and the thread selection variable. The values of token variables indicate which transition is ready to be executed in a thread, and the value of thread selection variable shows which thread is scheduled at a given step. With the values of these two kinds of variables, the trace that leads to a message race can be replayed, thus solving the non-repeatability problem in the debugging of Web Services. According to table 1, at the 10th step, the variable values satisfy the message race constraint: the bank thread is scheduled for execution $(S, T4)$ and its pending transition is a receive operation $(tk_4, B1)$. At the same time, there exist two send operations $(tk_3, Q6)$ and $(tk_5, S3)$ and all the three operations are on the same channel.

**Table 1.** Partial valuation to the FOL formula translated from the stock-trading WSMG model

| Step | Partial Valuation |
|---|---|
| 0 | $(tk_0, M0), (tk_1, \top), (tk_2, \top), (tk_3, \top), (tk_4, \top), (tk_5, \top), (S, T0)$ |
| 1 | $(tk_0, \bot), (tk_1, Q0), (tk_2, \top), (tk_3, \top), (tk_4, B0), (tk_5, S0), (S, T1)$ |
| 2 | $(tk_0, \bot), (tk_1, \bot), (tk_2, Q1), (tk_3, Q2), (tk_4, B0), (tk_5, S0), (S, T3)$ |
| 3 | $(tk_0, \bot), (tk_1, \bot), (tk_2, Q1), (tk_3, Q4), (tk_4, B0), (tk_5, S0), (S, T3)$ |
| 4 | $(tk_0, \bot), (tk_1, \bot), (tk_2, Q1), (tk_3, Q6), (tk_4, B0), (tk_5, S0), (S, T2)$ |
| 5 | $(tk_0, \bot), (tk_1, \bot), (tk_2, Q3), (tk_3, Q6), (tk_4, B0), (tk_5, S0), (S, T5)$ |
| 6 | $(tk_0, \bot), (tk_1, \bot), (tk_2, Q3), (tk_3, Q6), (tk_4, B0), (tk_5, S1), (S, T2)$ |
| 7 | $(tk_0, \bot), (tk_1, \bot), (tk_2, Q5), (tk_3, Q6), (tk_4, B0), (tk_5, S1), (S, T2)$ |
| 8 | $(tk_0, \bot), (tk_1, \bot), (tk_2, Q7), (tk_3, Q6), (tk_4, B0), (tk_5, S2), (S, T5)$ |
| 9 | $(tk_0, \bot), (tk_1, \bot), (tk_2, Q7), (tk_3, Q6), (tk_4, B0), (tk_5, S3), (S, T4)$ |
| 10 | $(tk_0, \bot), (tk_1, \bot), (tk_2, Q7), (tk_3, Q6), (tk_4, B1), (tk_5, S3), (S, T4)$ |

The second case study is based on the loan-approval Web Service which is shown in Fig. 4. It consists of four sub-services: a customer service (Customer), an approval service (Approval), an approver service (Approver), and an assessor service (Assessor). The approval service receives loan requests from the customer service. If the requested loan amount is less than a predetermined threshold, the loan request is sent to the approver service for automatic approval. Otherwise, the loan request is sent to the assessor service. When the assessor service receives a loan request, it assesses the risk associated with the customer, and then sends the risk assessment to the approval process. If the risk is high, the approval process denies the request; otherwise, the request is forwarded to the approver process. When the approver process receives a request, it automatically stamps the request as approved, and sends it back to the approval process. When the approval process receives an approved request from the approver process, it forwards the request to the customer.

When Yices is fed the SMT formulas corresponding to the loan-approval Web Service, it was able to detect a potential message race that happens when two quote services send prices-update messages to the bank service. Table 2 shows the output of Yices. At the 8th step, the variable values satisfy the message race constraint: $L1$ is scheduled for execution and it is a receive operation. At the same time both the pending transitions of $C4$, and $C5$ are send operations.

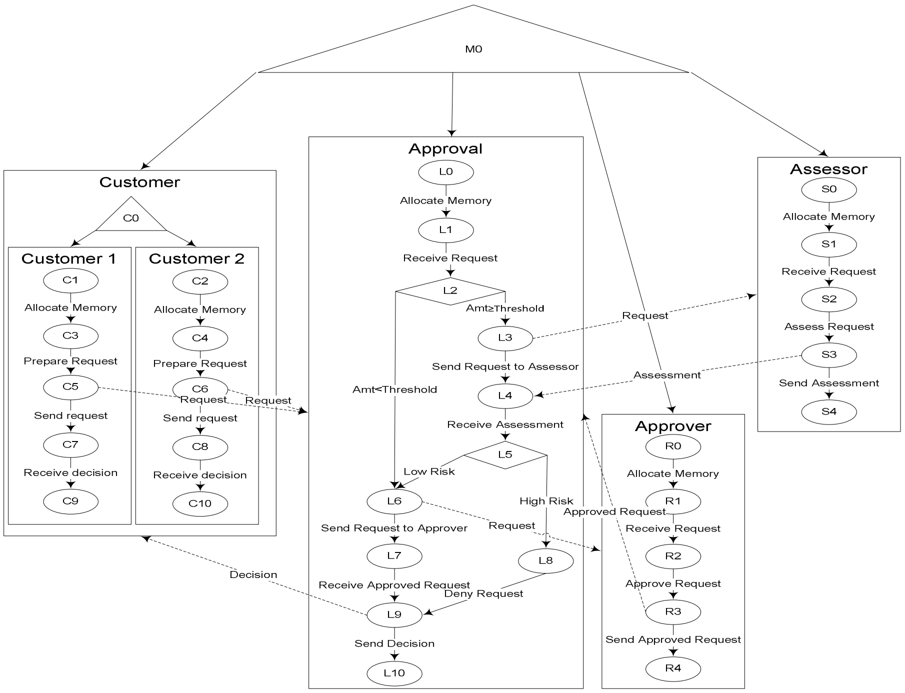

**Fig. 4.** The loan-approval Web Service

**Table 2.** Partial valuation to the FOL formula translated from the loan-approval WSMG model

| Step | Partial Valuation |
|---|---|
| 0 | $(\mathrm{tk}_0, \mathrm{M0}), (\mathrm{tk}_1, \top), (\mathrm{tk}_2, \top), (\mathrm{tk}_3, \top), (\mathrm{tk}_4, \top), (\mathrm{tk}_5, \top), (\mathrm{tk}_6, \top), (\mathrm{S}, \mathrm{T0})$ |
| 1 | $(\mathrm{tk}_0, \bot), (\mathrm{tk}_1, \mathrm{C0}), (\mathrm{tk}_2, \mathrm{R0}), (\mathrm{tk}_3, \mathrm{L0}), (\mathrm{tk}_4, \mathrm{S0}), (\mathrm{tk}_5, \top), (\mathrm{tk}_6, \top), (\mathrm{S}, \mathrm{T1})$ |
| 2 | $(\mathrm{tk}_0, \bot), (\mathrm{tk}_1, \bot), (\mathrm{tk}_2, \mathrm{R0}), (\mathrm{tk}_3, \mathrm{L0}), (\mathrm{tk}_4, \mathrm{S0}), (\mathrm{tk}_5, \mathrm{C1}), (\mathrm{tk}_6, \mathrm{C2}), (\mathrm{S}, \mathrm{T3})$ |
| 3 | $(\mathrm{tk}_0, \bot), (\mathrm{tk}_1, \bot), (\mathrm{tk}_2, \mathrm{R0}), (\mathrm{tk}_3, \mathrm{L1}), (\mathrm{tk}_4, \mathrm{S0}), (\mathrm{tk}_5, \mathrm{C1}), (\mathrm{tk}_6, \mathrm{C2}), (\mathrm{S}, \mathrm{T5})$ |
| 4 | $(\mathrm{tk}_0, \bot), (\mathrm{tk}_1, \bot), (\mathrm{tk}_2, \mathrm{R0}), (\mathrm{tk}_3, \mathrm{L1}), (\mathrm{tk}_4, \mathrm{S0}), (\mathrm{tk}_5, \mathrm{C3}), (\mathrm{tk}_6, \mathrm{C2}), (\mathrm{S}, \mathrm{T5})$ |
| 5 | $(\mathrm{tk}_0, \bot), (\mathrm{tk}_1, \bot), (\mathrm{tk}_2, \mathrm{R0}), (\mathrm{tk}_3, \mathrm{L1}), (\mathrm{tk}_4, \mathrm{S0}), (\mathrm{tk}_5, \mathrm{C5}), (\mathrm{tk}_6, \mathrm{C2}), (\mathrm{S}, \mathrm{T4})$ |
| 6 | $(\mathrm{tk}_0, \bot), (\mathrm{tk}_1, \bot), (\mathrm{tk}_2, \mathrm{R0}), (\mathrm{tk}_3, \mathrm{L1}), (\mathrm{tk}_4, \mathrm{S1}), (\mathrm{tk}_5, \mathrm{C5}), (\mathrm{tk}_6, \mathrm{C2}), (\mathrm{S}, \mathrm{T6})$ |
| 7 | $(\mathrm{tk}_0, \bot), (\mathrm{tk}_1, \bot), (\mathrm{tk}_2, \mathrm{R0}), (\mathrm{tk}_3, \mathrm{L1}), (\mathrm{tk}_4, \mathrm{S1}), (\mathrm{tk}_5, \mathrm{C5}), (\mathrm{tk}_6, \mathrm{C4}), (\mathrm{S}, \mathrm{T6})$ |
| 8 | $(\mathrm{tk}_0, \bot), (\mathrm{tk}_1, \bot), (\mathrm{tk}_2, \mathrm{R0}), (\mathrm{tk}_3, \mathrm{L1}), (\mathrm{tk}_4, \mathrm{S1}), (\mathrm{tk}_5, \mathrm{C5}), (\mathrm{tk}_6, \mathrm{C4}), (\mathrm{S}, \mathrm{T6})$ |

The experiments were performed on a computer with Intel Core 2 Duo 2.6GHz processor and 4GB memory. Table 3 reports statistics that are related to solving the SMT formulas in the two case studies, including the number of decisions, number of conflicts, number of Boolean variables and memory usage during the SMT solving procedure. The last two rows list the memory and time usage of the two case studies.

**Table 3.** Yices statistics

| Yices Statistics | Loan-approval | Stock-trading |
|---|---|---|
| #Decisions | 11833 | 7954 |
| #Conflicts | 6411 | 869 |
| Boolean variables | 8845 | 5176 |
| Memory used (MB) | 20.1 | 13 |
| CPU Time (sec.) | 2.8 | 0.45 |

## 6 Conclusion and Discussion

To improve the reliability and consequently the trustworthiness of Web Services, potential messages races should be detected. We have addressed the problem of detecting message races in BPEL Web Services. The main contribution of this paper is a novel approach that reduces message race detection to constraint solving and uses modern SMT solvers to check the satisfiability of the SMT formula translated from the WSMG models. Given a predefined bound $B$, our approach is both sound and complete within the bound. Compared with traditional testing approaches that repeatedly execute or simulate a Web Service, the advantages of our approach include 1) ability to prove the absence of message races within a predefined bound, 2) implicit exploration of astronomical amount of possible scenarios, 3) no need to control the non-deterministic factors in Web Services in testing environment, and 4) detailed bug reports.

However, even though all the message races reported by our approach are real, there are benign message races that are allowed by certain Web Services. How to differentiate benign and malicious message races is an important area that is out of the scope of this paper. For the future work, we plan to perform more significant case studies to future investigate the effectiveness of the approach.

**Acknowledgements.** The work was supported in part by ONR Grant N000140910740 and NSF Grant CCF-0811287.

## References

1. Schneider, F.B.: Trust in Cyberspace. National Academies Press, Washington (1999)
2. Christey, S. (eds), Top 25 most dangerous programming errors, CWE/SANS report (2009), http://cwe.mitre.org/top25
3. Klein, J., Leymann, F., Roller, D., Curbera, F., Goland, Y., Weerawarana, S.: Business process execution language for web services, version 1.1 (2003)
4. Satisfiability Modulo Theories, http://en.wikipedia.org/wiki/Satisfiability_Modulo_Theories
5. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)
6. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: 38th Design Automation Conference (DAC), pp. 530–535. ACM Press, New York (2001)

7. Een, N., Sorensson, N.: An extensible sat-solver. In: Satisfiability Workshop, pp. 333–336 (2003)
8. Dutertre, B., de Moura, L.: Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
9. Moura, L.D., Bjrner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
10. Netzer, R.H.B., Miller, B.P.: Optimal tracing and replay for debugging message-passing parallel programs. In: Super computing 1992: Proceedings of the 1992 ACM/IEEE conference on Supercomputing, pp. 502–511. IEEE Computer Society Press, Los Alamitos (1992)
11. Netzer, R.H.B., Brennan, T.W., Damodaran-Kamal, S.K.: Debugging race conditions in message-passing programs. In: SPDT 1996: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, pp. 31–40. ACM Press, New York (1996)
12. Park, M.Y., Shim, S.J., Jun, Y.K., Park, H.R.: Mpirace-check: Detection of message races in MPI programs. In: Cérin, C., Li, K.-C. (eds.) GPC 2007. LNCS, vol. 4459, pp. 322–333. Springer, Heidelberg (2007)
13. Weiss, M., Esfandiari, B.: On feature interactions among web services. In: IEEE International Conference on Web Services. IEEE Computer Society, Los Alamitos (2004)
14. Zhang, J., Su, S., Yang, F.: Detecting race conditions in web services. In: AICT ICIW 2006: Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services, p. 184. IEEE Computer Society, Washington (2006)
15. Zhang, J., Yang, F., Su, S.: Detecting feature interactions in web services with model checking techniques. The Journal of China Universities of Posts and Telecommunications 14(3), 108–112 (2007)
16. Alur, R., NcDougall, M., Yang, Z.: Exploiting Behavioral Hierarchy for Efficient Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 338. Springer, Heidelberg (2002)
17. Elwakil, M., Yang, Z., Liqiang, W.: CRI: Symbolic Debugger for MCAPI Applications. In: Chin, W.-N. (ed.) ATVA 2010. LNCS, vol. 6252, pp. 353–358. Springer, Heidelberg (2010)
18. Elwakil, M., Yang, Z.: Debugging Support Tool for MCAPI Applications. In: Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD - VIII). ACM, Trento (2010)