

The Consistency of Web Conversations

UCLA CSD Technical Report TR-080018

Jeffrey Fischer Rupak Majumdar Francesco Sorrentino
University of California, Los Angeles
{fischer,rupak}@cs.ucla.edu sorrentino@dia.unisa.it

July 1, 2008

Abstract

We describe `BPELCheck`, a tool for statically analyzing interactions of composite web services implemented in BPEL. Our algorithm is *compositional*, and checks each process interacting with an abstraction of its peers, without constructing the product state space. Interactions between pairs of peer processes are modeled using *conversation automata* which encode the set of valid message exchange sequences between the two processes. A process is *consistent* if each possible conversation leaves its peer automata in a state labeled as consistent and the overall execution satisfies a user-specified predicate on the automata states.

We have implemented `BPELCheck` in the Enterprise Service Pack of the NetBeans development environment. Our tool handles the major syntactic constructs of BPEL, including sequential and parallel composition, exception handling, flows, and Boolean state variables. We have used `BPELCheck` to check conversational consistency for a set of BPEL processes, including an industrial example.

1 Introduction

Distributed message-passing web services are extremely difficult to get right. At a simple level, a service consists of two parts: activities and message exchanges. Activities co-ordinate the specific tasks necessary for the service, for example, through standard programming language constructs such as sequential and parallel composition and fault handling. In addition, activities update “state” such as underlying databases to reflect the effect of a service. A process interacts with its peers by sending and receiving messages that enable it to participate in a more complex service while maintaining autonomy. The programmer must ensure that the distributed interaction leaves each service in a consistent state at the end of their interactions. Our goal in this paper is to formalize a notion of *consistency* for interacting web services, and to develop a tool to check a service for consistency.

Ensuring consistency for web services is an exceptionally challenging task. First there is no central service mediating all interactions, and the state is distributed among the peers. Second, interactions can be long-running, and span several days, making traditional database solutions of atomic transactions not applicable. Consequently, web service programmers ensure consistent views through complex control flow constructs such as compensations [18]. Unfortunately, most often the programmer does not have tools beyond the debugger to ensure the correctness of these services.

As an example for consistency requirements, consider a store that accepts customer orders and (a) charges the customer's credit card and (b) forwards the order to a warehouse for delivery. This interaction can have several possible executions: the "normal" execution where the card is charged and the shipment made, and additional executions where either the bank refuses the credit card or the warehouse does not have the stock. In each case, we expect that (1) the store communicates with the bank and the warehouse according to a pre-set protocol, and there are no "surprise" messages that are not handled, and (2) either the card is charged and the shipment made, or both the card is not charged and the shipment not made.

In this work, we develop a local algorithm to check such consistency requirements in web service interactions. Our algorithm focuses on one process, and abstracts the interaction of this process with its peers through *conversation automata*. A conversation automaton is a finite-state machine that specifies an over-approximation of the valid message exchanges between the process and its peer [5]. Our notion of consistency has two parts. First, we check that each possible execution of the system generates consistent message orderings, that is, no conversation automaton gets stuck. Second, we annotate the states of the automaton with "committed" or "no-change" identifying whether the peer has updated its persistent state or whether no change has effectively been made (i.e., all changes, if any, have been rolled back). We require the programmer to provide a *consistency predicate* that relates the states of the peers at the end of every transaction. Our algorithm checks that a process running concurrently with the conversation automata for each of its peers is consistent.

For example, for the store example, we focus on the store, and abstract its interactions with the bank and the warehouse as conversation automata. Further, we formulate a consistency predicate that either both the warehouse state has been updated and the credit card has been charged (both "commit") or neither the warehouse nor the credit card state has changed (both "no-change").

By focusing on one process and abstracting its interactions with its peers as conversation automata, we avoid the state explosion inherent in an algorithm on the global state space. Unfortunately, the local checks give us weaker guarantees about global stuck-freedom. Since we abstract peers individually, we do not rule out the possibility of a deadlock arising out of multiple peers each waiting for the other. This is a price we have to pay, since in practice, it is unreasonable to expect that an organization will have access to the source code of every peer process it interacts with. It is more reasonable to expect a web-service contract in the form of a conversation automaton, and indeed there are standard ways

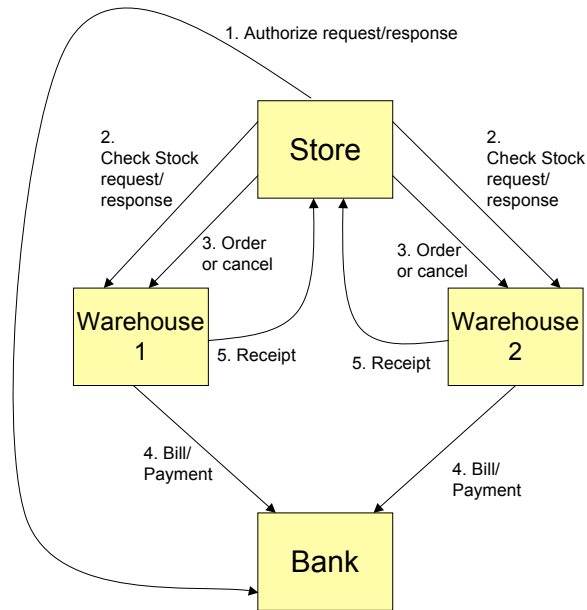


Figure 1: Peers interactions for e-commerce example.

(e.g., WSCL [1]) to specify this information.

The technical core of our algorithm is a reduction of the consistency verification problem to the satisfiability problem for bounded-domain logic with equality. The construction is similar to the reduction used in bounded model checking [3]. The technical challenge is in faithfully modeling the source level constructs in BPEL such as message-passing and synchronization, exception handling, and compensations, which do not appear in hardware or sequential software descriptions in which bounded model checking is commonly applied. We use an off-the-shelf decision procedure (Yices [11]) to check satisfiability of the resulting formula.

We have implemented our algorithm for checking consistency of BPEL processes in a tool called `BPELCheck`. While we describe our algorithm on a simple core calculus, `BPELCheck` is integrated with the NetBeans Enterprise Server Pack and handles BPEL processes directly. In initial experiments, we have run `BPELCheck` on a set of BPEL benchmarks, including the examples from the BPEL specification [12] as well as an industrial example from Sun Microsystems. In all cases, our tool was able to prove or disprove consistency within a second.

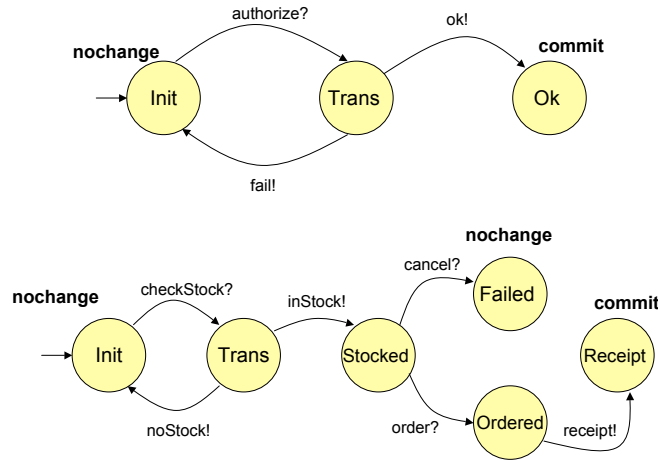


Figure 2: Automata for store-bank conversation (top) and store-warehouse conversation (bottom).

2 Example

To illustrate our approach to specifying and verifying business processes, we will use a simple e-commerce example, inspired by the one in [5]. As shown in Figure 1, our example involves the interactions of four business processes: a store process, which is attempting to fulfill an order, a bank process, which manages the funding for an order, and two warehouse processes, which manage inventory. To process an order, the store first authorizes with the bank the total value of the order. If this fails, the processing of the order stops. If the authorization is successful, the store checks whether the two warehouses have the necessary products in stock. If not, the order processing stops. The inventory checks actually reserve the products, so if the order fails at this point, any reserved inventory must be released. If both checks succeed, the store confirms the orders with the two warehouses. The warehouses then ship the goods and send a bill to the bank. The bank responds to the bill messages with payments to the warehouses. Finally, upon receiving payment, the warehouses each send a receipt to the store.

Conversation automata. We will now focus on this scenario from the store’s perspective. The store interacts with three *peer processes*: the bank and the two warehouses. We can represent the *conversation* (exchange of messages) with each peer using an automaton. The top automaton of Figure 2 represents the conversation with the bank. It has three states: **Init**, representing the initial state of the conversation, **Trans** representing the conversation after an **authorize** request has been sent to the bank, but before the bank has responded, and **Ok**, representing a successful authorization. Transitions between states occur when a message is sent or received between the process and the

peer represented by the automata. We write **authorize?** over the transition from **Init** to **Trans**, indicating that the peer receives an **authorize** message sent by the process. Likewise, **ok!** and **fail!** over transitions indicate when the peer sends **ok** and **fail** responses, respectively, to the process.

We label the **Init** state with **nochange** and the **Ok** state with **commit**. This labeling reflects the notion that a successful authorization changes the state of the bank, but a failed authorization does not. The **Trans** state is unlabeled, as it is an *inconsistent* state. The conversation between the store and the bank should not end in this state.

The bottom automaton of Figure 2 represents the conversation between the store and one of the warehouses (both have the same conversation protocol). This automaton is more complex, as it must encode a two-phase commit and then account for the receipt message. The store initiates the conversation by sending a **checkStock** message. If the product is available, the warehouse responds with a **inStock** message. If the product is unavailable, the warehouse responds with a **noStock** message, which terminates the conversation. In the successful case, the store must then respond by either committing the transaction with a **order** message or aborting it with a **cancel** message. The cancel is sent when the other warehouse was unable to fulfill its part of the order. Finally, a successful conversation will end with a **receipt** message from the warehouse. The **Init** and **Failed** states are both labeled as **nochange**, while the **Receipt** state is labeled as **commit**. All the other states are inconsistent.

Consistency predicates. In addition to guaranteeing that the conversations with the bank and the warehouses are always left in a consistent state, the designer of the store process would like to ensure certain relationships between these conversations. For the store, two invariants should be guaranteed:

1. If the bank authorization fails, neither warehouse transaction should occur.
2. If one of the warehouse transactions occurs, the other should occur and the bank authorization must be successful.

We can specify these requirements using a *consistency predicate*, a predicate over the **nochange/commit** labels of the conversation automata. **comm**(*A*) is **true** when the process terminates with peer *A*'s conversation in a state labeled **commit**. **nochange**(*A*) is **true** when the process terminates with peer *A*'s conversation in a state labeled **nochange**. A consistency predicate ψ , built from these atomic predicates over the peer automata, is *satisfied* when ψ evaluates to **true** for all executions of the process.

We can write our store process invariants as a consistency predicate:

$$\begin{aligned}
 &(\text{nochange}(\text{Bank}) \rightarrow \\
 &\quad (\text{nochange}(\text{Warehouse1}) \wedge \text{nochange}(\text{Warehouse2}))) \wedge \\
 &((\text{comm}(\text{Warehouse1}) \vee \text{comm}(\text{Warehouse2})) \rightarrow \\
 &\quad (\text{comm}(\text{Bank}) \wedge \text{comm}(\text{Warehouse1}) \wedge \\
 &\quad \text{comm}(\text{Warehouse2})))
 \end{aligned}$$

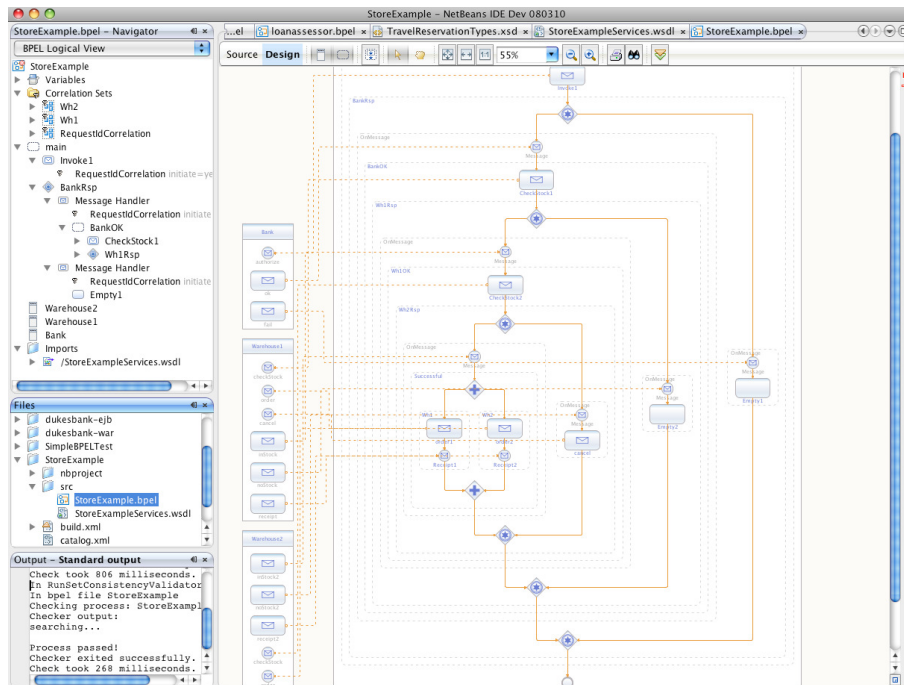


Figure 3: BPEL implementation of store process in NetBeans.

Store process implementation. Figure 3 shows an implementation of the store process in the NetBeans BPEL designer. Each peer process is represented by a *partner link*, appearing to the left of the flowchart. Thus, our approach of defining conversations between pairs of peers is directly reflected in the BPEL representation of the process. We write the implementation of the store process in a simple textual syntax similar to CCS (our implementation reads the XML representation produced by the BPEL designer in NetBeans). In our core language, `msg?p` and `msg!p` mean that the process receives message `msg` from peer `p` and sends message `msg` to peer `p`, respectively. `skip` is an atomic action which does nothing and `throw` raises an exception. These actions may be composed using four operators: “;” for sequential composition, “||” for parallel composition, “□” for non-deterministic choice, and “▷” for exception handling. The store process is written as:

```

authorize!Bank;
((ok?Bank; (checkStock!Warehouse1;
  ((inStock?Warehouse1; (checkStock!Warehouse2;
    (inStock2?Warehouse2;
      ((order!Warehouse1; receipt?Warehouse1) ||
        (order!Warehouse2; receipt?Warehouse2))) □

```

Atomic Action	$A ::=$	$a?i$	$a \in \text{Actions}, i \in \text{Peers}$
		$a!i$	$a \in \text{Actions}, i \in \text{Peers}$
Process	$P ::=$	skip throw	
		A	atomic actions
		$P; P$	sequence
		$P \parallel P$	parallel
		$P \square P$	choice
		$P \triangleright P$	exception handler

Figure 4: Syntax of process calculus.

```

(noStock?Warehouse2; cancel!Warehouse1))) □
(noStock?Warehouse1; skip))) □
(fail?Bank; skip)

```

The store process first sends an `authorize` message to the bank (message `authorize!Bank`). Then, it waits for either an `ok` or `fail` message from the bank. If the authorization is successful (i.e., `ok?Bank` is received), the store checks the stock of the two warehouses sequentially. If the first is successful but the second fails, the store must cancel the first warehouse’s stock reservation. If both are successful, the store submits the two order confirmation messages and waits for receipts in parallel.

When we run this process through `BPELCheck`, we find that it is indeed *conversationally consistent*: the process always terminates with all peer conversations in a consistent state and with the consistency predicate satisfied.

Bugs in our process can cause this verification to fail. For example, if the developer forgets to cancel the first warehouse’s stock reservation when the second reservation fails, the first warehouse’s conversation will be left in an inconsistent state, and `BPELCheck` will report an error. Processes which leave all conversations in consistent states but violate the consistency predicate will also fail. For example, a process which runs both warehouse conversations in parallel avoids leaving them in inconsistent states but violates the requirement that the two conversations must succeed or fail together. `BPELCheck` will report an error for this process as well.

3 Web Services

We use a simple process calculus to describe web services.

Processes. Figure 4 defines the syntax for processes. The operators in our syntax correspond to similar concrete operations in web service orchestration languages like BPEL. Our processes extend CCS [22] with an exception mechanism, but we do not allow recursion or hiding. Our processes can express many

concrete BPEL implementations; we extend the core language to additional features in the implementation.

Let **Actions** be a set of atomic messages, and **Peers** a set of *peers*. Processes are constructed from atomic messages, using a set of composition operations.

Atomic messages are indivisible send or receive operations to peers. For an action $a \in \mathbf{Actions}$ and a peer $i \in \mathbf{Peers}$, we write $a?i$ to denote a message a received from peer i , and write $a!i$ to denote a message a sent to peer i . We write \bowtie for $?$ or $!$, and $\bar{\bowtie}$ for the opposite of \bowtie , i.e., $\bar{\bowtie}=?$ if $\bowtie=!$ and $\bar{\bowtie}=!$ if $\bowtie=?$. We also write just a when the peer and whether it is a send or a receive is not relevant.

A process is either **skip** (which does nothing), **throw** (which throws an exception), an atomic action (send or receive), or a composition of atomic actions using one of the following composition operators. The sequence operator “;” runs the first process followed by the second. If the first fails, the second is not run. The parallel operator “||” runs two processes in parallel. The choice operator “□” non-deterministically selects one of two processes and runs it.¹ The exception handler “▷” runs the left process. If that process throws an exception (via **throw**), the right process is run. If the left process terminates without exception, the right process is ignored.

We define $|P|$, the size of process P , by induction: the size $|a \bowtie i|$ of an atomic message send or receive is 1, the size of **skip** and **throw** are both 1, and the size $|P_1 \otimes P_2|$ for any composition operation \otimes applied to P_1 and P_2 is $|P_1| + |P_2| + 1$.

Conversation Automata. A *conversation automaton* is a tuple $(Q, \Sigma, \delta, q^0, F^1, F^2)$ where Q is a finite set of states, $\Sigma = \mathbf{Actions} \times \{?, !\}$ is the alphabet, where **Actions** is the set of atomic messages and $?$ and $!$ denote message receive and message send respectively, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q^0 \in Q$ is an initial state, and $F^1 \subseteq Q$ and $F^2 \subseteq Q$ are disjoint subsets of Q called the *nochange* and *committed* states respectively. We call $F^1 \cup F^2$ the set of *consistent* states, and $Q \setminus (F^1 \cup F^2)$ the set of *inconsistent* states. We write $q \xrightarrow{a?} q'$ (respectively, $q \xrightarrow{a!} q'$) for $(q, (a, ?), q') \in \delta$ (respectively, $(q, (a, !), q') \in \delta$).

We assume that a conversation automaton is *complete*, that is, for every state $q \in Q$ and every action $a \bowtie \in \Sigma$, there is some state q' with $q \xrightarrow{a \bar{\bowtie}} q'$. This can be ensured in the standard way by adding “dead” states to the automaton.

Web Service. A web service consists of a process and k peers that exchange messages with the process. Formally, a web service $WS = (\mathbf{Actions}, P, \langle C_1, \dots, C_k \rangle)$ consists of a process P and an indexed set of conversation automata $\langle C_1, \dots, C_k \rangle$, one for each peer, where $\Sigma_i = \mathbf{Actions} \times \{?, !\}$ for each C_i , and each atomic action $a \bowtie i$ in P satisfies $i \in \{1, \dots, k\}$.

Semantics. Figure 5 gives structural congruence rules for processes. Henceforth, we identify structurally congruent processes.

¹For the purposes of this paper, we do not distinguish between internal and external choice. However, our results extend to the setting where this distinction is made.

$$\begin{aligned}
P &\equiv \text{skip}; P & P &\equiv P; \text{skip} \\
\text{throw}; P &\equiv \text{throw} & \text{throw} \parallel P &\equiv \text{throw} \\
(P_0; P_1); P_2 &\equiv P_0; (P_1; P_2) \\
P_0 \square P_1 &\equiv P_1 \square P_0 \\
P &\equiv P \parallel \text{skip} & P_0 \parallel P_1 &\equiv P_1 \parallel P_0 \\
P_0 \parallel (P_1 \parallel P_2) &\equiv (P_0 \parallel P_1) \parallel P_2
\end{aligned}$$

Figure 5: Structural congruence axioms

Figure 6 gives the small-step operational semantics of web services. The state of a web service consists of a process P and the states $s \in Q_1 \times \dots \times Q_k$ of the conversation automata of the peers. We write s_i to denote the i th component of the state s , and write $s[i \mapsto q]$ for the state whose i th component is q and all other components are the same as s .

The (**Msg**) rule deals with a message send or receive to the i th peer. This changes the state of the i th conversation automaton but leaves all other states unchanged. The (**Throw**) rule replaces an exception with its handler, and the (**Normal**) rule drops an exception handler for a process if the process terminates normally (via **skip**).

The operational semantics defines a transition relation \rightarrow on the states of a web service, which synchronizes the atomic actions (message sends and receives) of the process with the states of the peer automata. Let \rightarrow^* denote the reflexive transitive closure of \rightarrow .

A (*complete*) run of the web service $WS = (P, \langle C_1, \dots, C_k \rangle)$ is a sequence $\langle P_0, s^0 \rangle, \dots, \langle P_n, s^n \rangle$ such that (1) $P_0 \equiv P$ and for each $i \in \{1, \dots, k\}$, we have $s_i^0 = q_i^0$, i.e., each conversation automaton starts in its initial state; (2) for each $i \in \{0, \dots, n-1\}$, we have $\langle P_i, s_i \rangle \rightarrow \langle P_{i+1}, s_{i+1} \rangle$, and (3) there is no $\langle P, s \rangle$ such that $\langle P_n, s_n \rangle \rightarrow \langle P, s \rangle$.

The process at the last state of a complete run is always either **skip** or **throw**. In case it is **skip**, we say the run terminated normally. In case it is **throw**, we say the run terminated abnormally. The run is *consistent* if, for each $i \in \{1, \dots, k\}$, we have $s_i^n \in F_i^1 \cup F_i^2$, i.e., each conversation automaton is in a consistent state at the end of the run.

$$\frac{\delta_i(s_i, a\bar{\boxtimes}, q) \quad s' = s[i \mapsto q]}{\langle a \boxtimes i, s \rangle \rightarrow \langle \text{skip}, s' \rangle} \text{ (Msg)}$$

$$\frac{\langle P, s \rangle \rightarrow \langle P', s' \rangle}{\langle P; Q, s \rangle \rightarrow \langle P'; Q, s' \rangle} \text{ (Seq)}$$

$$\frac{\langle P, s \rangle \rightarrow \langle P', s' \rangle}{\langle P \square Q, s \rangle \rightarrow \langle P', s' \rangle} \text{ (Choice)}$$

$$\frac{\langle P, s \rangle \rightarrow \langle P', s' \rangle}{\langle P \parallel Q, s \rangle \rightarrow \langle P' \parallel Q, s' \rangle} \text{ (Par)}$$

$$\overline{\langle \text{throw} \triangleright Q, s \rangle \rightarrow \langle Q, s \rangle} \text{ (Throw)}$$

$$\overline{\langle \text{skip} \triangleright Q, s \rangle \rightarrow \langle \text{skip}, s \rangle} \text{ (Normal)}$$

$$\frac{\langle P, s \rangle \rightarrow \langle P', s' \rangle}{\langle P \triangleright Q, s \rangle \rightarrow \langle P' \triangleright Q, s' \rangle} \text{ (Except)}$$

$$\frac{P \equiv P' \quad \langle P', s \rangle \rightarrow \langle Q', s' \rangle \quad Q' \equiv Q}{\langle P, s \rangle \rightarrow \langle Q, s' \rangle} \text{ (Cong)}$$

Figure 6: Small-step operational semantics

4 Conversational Consistency

4.1 Consistency for Web Services

Let $WS = (P, \langle C_1, \dots, C_k \rangle)$ be a web service. Let $AP = \{\text{nochange}(i), \text{comm}(i) \mid i \in \{1, \dots, k\}\}$ be a set of atomic propositions. Intuitively, the proposition $\text{nochange}(i)$ is true if the conversation automaton C_i of peer i is in a **nochange** state, and the proposition $\text{comm}(i)$ is true if the conversation automaton C_i of peer i is in a **committed** state.

A conversational consistency predicate is a Boolean formula over AP . Let $s \in Q_1 \times \dots \times Q_k$ be a state and ψ a conversational consistency predicate. We write $s \models \psi$ if the Boolean formula ψ evaluates to true when each predicate $\text{nochange}(i)$ is replaced by true if $s_i \in F_i^1$ (i.e., C_i is in a **nochange** state) and by false otherwise, and each predicate $\text{comm}(i)$ is replaced by true if $s_i \in F_i^2$ (i.e., C_i is in a **committed** state) and by false otherwise,

A web service WS is *consistent* with respect to a conversational consistency predicate ψ if every complete run of WS is consistent and for every complete run with last state $\langle \text{skip}, s \rangle$ we have that $s \models \psi$. The conversation consistency verification problem takes as input a web service WS and a conversational consistency predicate ψ and returns “yes” if WS is consistent with respect to ψ and “no” otherwise.

Theorem 1 *The conversation consistency verification problem is co-NP complete.*

In case a web service is inconsistent, there is a violating execution that is polynomial in the size of the service. This shows membership in co-NP. The problem is co-NP hard by a reduction from Boolean validity, similar to the reduction for set consistency [13].

4.2 Consistency Verification

We now give an algorithm for the conversation consistency verification problem that reduces the problem to satisfiability checking. Given a web service $WS = (P, \langle C_1, \dots, C_k \rangle)$ and a consistency specification ψ , we construct a formula φ such that φ is satisfiable iff WS is not consistent w.r.t. ψ . We build φ by induction on the structure of the process, in a way similar to bounded model checking. The only technical difficulty is to ensure φ is polynomial in the size of the input; clearly, a naive encoding of the complete runs of the web service yields an exponential formula.

We start with some definitions. Given a process P , define the function depth by induction as follows:

$$\begin{aligned} \text{depth}(\text{skip}) &= 1 & \text{depth}(\text{throw}) &= 1 \\ \text{depth}(a?i) &= 1 & \text{depth}(a!i) &= 1 \\ \text{depth}(P_1 \otimes P_2) &= \text{depth}(P_1) + \text{depth}(P_2) & \otimes &\in \{;, \parallel, \triangleright\} \\ \text{depth}(P_1 \square P_2) &= \max(\text{depth}(P_1), \text{depth}(P_2)) \end{aligned}$$

For a process P , the value $\text{depth}(P)$ gives the maximum number of message exchanges on any run of the process P .

A process P can give rise to up to $\text{depth}(P)$ transitions. We call each evaluation step of P a *step* in its run. We introduce symbolic variables that represent the state and transitions of the web service for each step. A *state variable* is a k -tuple taking values in the domain $Q_1 \times \dots \times Q_k$. In the following, we use (superscripted) variables s^j to stand for states. Each s^j is a k -tuple taking values over $Q_1 \times \dots \times Q_k$, and the element s_i^j takes values over the set of states Q_i of the conversation automaton C_i and represents the state of C_i in the state s^j . A *transition predicate* is a formula over two state variables s and s' . A transition predicate gives a relation between the old state s and a new state s' that encodes one or more steps of a process execution.

For each conversation automaton C_i , and each action $a \bowtie \in \Sigma$, we define a transition predicate, written $\Delta_i(a \bowtie)(s, s')$, that encodes the transition relation of C_i for symbolic states s and s' . The symbolic transition relation $\Delta_i(a \bowtie)(s, s')$ is a formula with free variables s and s' that encodes the transition relation in the obvious way:

$$\bigvee_{(q, a \bowtie, q') \in \delta_i} s = q \wedge s' = q'$$

Constructing the Consistency Predicate. For simplicity of exposition, we first give the construction of the consistency predicate for processes without exception handling, i.e., without **throw** and $\cdot \triangleright \cdot$. We construct φ in two steps. First, given a process P , we construct by induction a sequence of $\text{depth}(P)$ transition predicates that represent the transitions in possible executions of P . Second, we tie together the transitions, constrain the executions to start in the initial state, and check whether at the end, the state is consistent according to the consistency specification.

We introduce some notation. Let P be a process, let $p = \text{depth}(P)$. Let $(s^1, t^1), \dots, (s^p, t^p)$ be pairs of state variables. We shall construct a formula $\text{transitions}[P](s^1, t^1, \dots, s^p, t^p)$ that encodes the sequence of transitions in an execution of P . This formula can have additional free variables. We construct $\text{transitions}[P](s^1, t^1, \dots, s^p, t^p)$ by induction on the structure of P as follows.

If $P \equiv \text{skip}$, then $\text{transitions}[P](s^1, t^1)$ is $t^1 = s^1$ (that is, the state does not change on executing a **skip**).

If $P \equiv a \bowtie i$, the formula $\text{transitions}[a \bowtie i](s^1, t^1)$ is

$$\bigwedge_{j \neq i} t_j^1 = s_j^1 \wedge \Delta_i(a \bowtie)(s_i^1, t_i^1)$$

Intuitively, this specifies that the i th part of the state changes according to the step of the automaton, while every other part of the state (i.e., the states of the other conversation automata) remain the same.

Let $P \equiv P_1; P_2$, and $p_1 = \text{depth}(P_1)$ and $p_2 = \text{depth}(P_2)$. We construct recursively the formulas $\text{transitions}[P_1](s^1, t^1, \dots, s^{p_1}, t^{p_1})$ and $\text{transitions}[P_2](u^1, v^1, \dots, u^{p_2}, v^{p_2})$, where the free variables in the two formulas are disjoint. Let $x^1, y^1, \dots, x^{p_1+p_2}, y^{p_1+p_2}$ be a fresh sequence of

state variables that we shall use to encode the transitions of P . Then $\text{transitions}[P](x^1, y^1, \dots, x^{p_1+p_2}, y^{p_1+p_2})$ is given by

$$\begin{aligned} & \text{transitions}[P_1](s^1, t^1, \dots, s^{p_1}, t^{p_1}) \wedge \\ & \text{transitions}[P_2](u^1, v^1, \dots, u^{p_2}, v^{p_2}) \wedge \\ & \bigwedge_{j=1}^{p_1} (x^j = s^j \wedge y^j = t^j) \wedge \bigwedge_{j=1}^{p_2} (x^{p_1+j} = u^j \wedge y^{p_1+j} = v^j) \end{aligned}$$

Intuitively, the transitions for a sequential composition of processes consist of the transitions of the first process followed by the transitions of the second process.

Let $P \equiv P_1 \square P_2$, and $p_1 = \text{depth}(P_1)$ and $p_2 = \text{depth}(P_2)$. Without loss of generality, assume $p_1 \geq p_2$. We construct recursively the formulas $\text{transitions}[P_1](s^1, t^1, \dots, s^{p_1}, t^{p_1})$ and $\text{transitions}[P_2](u^1, v^1, \dots, u^{p_2}, v^{p_2})$, where the free variables in the two formulas are disjoint. Let $x^1, y^1, \dots, x^{p_1}, y^{p_1}$ be a set of fresh state variables. Then, $\text{transitions}[P](x^1, y^1, \dots, x^{p_1}, y^{p_1})$ is given by

$$\begin{aligned} & \left(\begin{array}{c} \text{transitions}[P_1](s^1, t^1, \dots, s^{p_1}, t^{p_1}) \wedge \\ \bigwedge_{j=1}^{p_1} (x^j = s^j \wedge y^j = t^j) \end{array} \right) \\ & \quad \vee \\ & \left(\begin{array}{c} \text{transitions}[P_2](u^1, v^1, \dots, u^{p_2}, v^{p_2}) \wedge \\ \bigwedge_{j=1}^{p_2} (x^j = u^j \wedge y^j = v^j) \wedge \bigwedge_{j=p_2+1}^{p_1-p_2} y^j = x^j \end{array} \right) \end{aligned}$$

Intuitively, the transitions of a choice are either the transitions of the first subprocess or the transitions of the second (hence the disjunction), and we add enough “skip” transitions to ensure that each side of the disjunction has the same number of transitions.

If $P \equiv P_1 \parallel P_2$, and $p_1 = \text{depth}(P_1)$ and $p_2 = \text{depth}(P_2)$. We recursively construct formulas $\text{transitions}[P_1](s^1, t^1, \dots, s^{p_1}, t^{p_1})$ and $\text{transitions}[P_2](u^1, v^1, \dots, u^{p_2}, v^{p_2})$. We construct the formula $\text{transitions}[P](x^1, y^1, \dots, x^{p_1+p_2}, y^{p_1+p_2})$ as follows.

For $i \in \{1, \dots, p_1 + 1\}$ and $j \in \{1, \dots, p_2 + 1\}$, let $\alpha^{i,j}$ be fresh Boolean variables. For each $i \in \{1, \dots, p_1\}$ and $j \in \{1, \dots, p_2\}$, we introduce the following constraints on $\alpha^{i,j}$:

$$\alpha^{i,j} \leftrightarrow \left(\begin{array}{c} (x^{i+j-1} = s^i \wedge y^{i+j-1} = t^i \wedge \alpha^{i+1,j}) \\ \vee \\ (x^{i+j-1} = u^j \wedge y^{i+j-1} = v^j \wedge \alpha^{i,j+1}) \end{array} \right)$$

For $j \in \{1, \dots, p_2\}$, we introduce the constraint

$$\alpha^{p_1+1,j} \leftrightarrow (x^{p_1+j} = u^j \wedge y^{p_1+j} = v^j \wedge \alpha^{p_1+1,j+1})$$

For $i \in \{1, \dots, p_1\}$, we introduce the constraint

$$\alpha^{i,p_2+1} \leftrightarrow (x^{p_2+i} = s^i \wedge y^{p_2+i} = t^i \wedge \alpha^{i+1,p_2+1})$$

Finally, we set

$$\alpha^{p_1+1,p_2+1} = \text{true}$$

Then, $\text{transitions}[P](x^1, y^1, \dots, x^{p_1+p_2}, y^{p_1+p_2})$ is the conjunction of the constraints above for all i, j , together with the constraint $\alpha^{1,1}$.

Intuitively, the construction above takes two sequences of transitions, and encodes all possible interleavings of these sequences. A naive encoding of all interleavings leads to an exponential formula. Therefore, we use dynamic programming to memoize sub-executions. The variable $\alpha^{i,j}$ indicates we have executed the first $i - 1$ transitions from the first sequence and the first $j - 1$ transitions from the second sequence. Then the constraint on $\alpha^{i,j}$ states that we can either execute the i th transition of the first sequence and go to state $\alpha^{i+1,j}$, or execute the j th transition of the second and go to state $\alpha^{i,j+1}$. Conjoining all these constraints encodes all possible interleavings.

Finally, we construct φ as follows. Let P be a process and $\text{depth}(P) = p$. Let $\text{Init}(s)$ be the predicate

$$\bigwedge_{i=1}^k s_i = q_i^0$$

that states that each automaton is in its initial state, and $\text{Consistent}(s)$ be the predicate

$$\bigwedge_{i=1}^k s_i \in F_i^1 \cup F_i^2$$

stating that each automaton is in a consistent state (note that we can expand each set-inclusion into a finite disjunction over states).

Given the formula $\text{transitions}[P](s^1, t^1, \dots, s^p, t^p)$, we construct φ by “stitching together” the transitions, conjoining the initial and consistency conditions:

$$\begin{aligned} & \text{Init}(s^1) \wedge \\ & \text{transitions}[P](s^1, t^1, \dots, s^p, t^p) \wedge \bigwedge_{i=1}^{p-1} t^i = s^{i+1} \\ & \wedge (\neg \text{Consistent}(t^p) \vee \neg \psi(s^p)) \end{aligned}$$

Consistency Predicate with Exception Handling. We now extend the above construction with `throw` and exception handling. In this case, we keep an additional Boolean variable in the state that indicates whether an error has occurred. Initially, this variable is set to 0. The variable is set to 1 by a `throw` statement, and reset to 0 by the corresponding exception handler. Additionally, we modify the transitions function `transitions` to maintain the state once an error has occurred in order to simulate the propagation of exceptions.

Formally, a *state variable* is now a $(k + 1)$ -tuple, where the 0th element is a Boolean variable indicating if an error has occurred and the 1st to the k th elements are states of the k peer automata as before. We extend the constraints to additionally track the error status. First, $\text{transitions}[\text{throw}](s, t)$ below sets the error status:

$$t_0 = 1 \wedge \bigwedge_{j=1}^k t_j = s_j$$

Second, let P be the process $P_1 \triangleright P_2$ with $\text{depth}(P_1) = p_1$ and $\text{depth}(P_2) = p_2$. Let $x^1, y^1, \dots, x^{p_1+p_2}, y^{p_1+p_2}$ be new state variables. Then $\text{transitions}[P](x^1, y^1, \dots, x^{p_1+p_2}, y^{p_1+p_2})$ is given by

$$\begin{aligned} & \text{transitions}[P_1](s^1, t^1, \dots, s^{p_1}, t^{p_1}) \wedge \bigwedge_{j=1}^{p_1} (x^j = s^j \wedge y^j = t^j) \wedge \\ & \text{transitions}[P_2](u^1, v^1, \dots, u^{p_2}, v^{p_2}) \wedge \\ & (y_0^{p_1} = 1 \rightarrow \left(\begin{array}{l} x_0^{p_1+1} = 0 \wedge \bigwedge_{i=1}^k x_i^{p_1+1} = u_i^1 \wedge \\ \bigwedge_{j=2}^{p_2} (x^{p_1+j} = u^j \wedge y^{p_1+j} = v^j) \end{array} \right)) \\ & \quad \wedge \\ & (y_0^{p_1} = 0 \rightarrow \bigwedge_{j=1}^{p_2} (y^{p_1+j} = x^{p_1+j})) \end{aligned}$$

That is, the first p_1 steps of $P_1 \triangleright P_2$ coincides with the steps of P_1 (line 1), and the next p_2 steps are the steps of P_2 if the error bit is set to 1 at the end of executing P_1 (line 3), in which case the error bit is reset to 0, or identity steps (line 4) if the error bit is not set.

The constructions for sequential and parallel composition and choice are modified to stop executing once an error has been thrown. For example, the constraints for sequential composition are

$$\begin{aligned} & \text{transitions}[P_1](s^1, t^1, \dots, s^{p_1}, t^{p_1}) \wedge \\ & \text{transitions}[P_2](u^1, v^1, \dots, u^{p_2}, v^{p_2}) \wedge \\ & \bigwedge_{j=1}^{p_1} (x^j = s^j \wedge y^j = t^j) \wedge \\ & (y_0^{p_1} = 0 \rightarrow \bigwedge_{j=1}^{p_2} (x^{p_1+j} = u^j \wedge y^{p_1+j} = v^j)) \wedge \\ & (y_0^{p_1} = 1 \rightarrow \bigwedge_{j=1}^{p_2} (x^{p_1+j} = y^{p_1+j})) \end{aligned}$$

We omit the similar modifications for \square and \parallel .

Finally, the initial condition $\text{Init}(s)$ has the additional conjunct $s_0 = 0$ and the predicate $\text{Consistent}(s)$ has the additional conjunct $s_0 = 0$.

Soundness and Completeness. The following theorem states that our construction is sound and complete. It is easily proved by induction on the run of the web service.

Theorem 2 *For any web service $WS = (P, \langle C_1, \dots, C_k \rangle)$ and consistency specification ψ , we have φ is satisfiable iff WS does not satisfy ψ . Further, φ is polynomial in the size of WS .*

Further, while we have used equality in our description of φ , since each variable varies over a finite domain, we can compile φ into a purely propositional formula.

Corollary 1 *The consistency verification problem is in co-NP.*

Relationship to Assume-Guarantee. A *concrete web service* is a collection $W = \langle P_1, P_2, \dots, P_k \rangle$ of processes, whose semantics is given by the obvious modifications of the rules in Figure 6 (see e.g., [10]). *Stuck-freedom* [24] formalizes the notion that a concrete web service does not deadlock waiting for messages that are never sent or send messages that are never received.

An assume-guarantee principle would state that the local consistency checks using our algorithm would imply that the concrete web service is stuck-free. Assume-guarantee principles make strong non-blocking assumptions about processes. Since our processes may not be non-blocking, we do not have an assume-guarantee principle. Consider the three processes: process P_1 is $a?2; a!3$, process P_2 is $a?3; a!1$, and process P_3 is $a?1; a!2$. These processes are in deadlock. However, consider a conversation automaton $A = (\{q_1, q_2\}, \{a?, a!\}, \delta, q_1, \{q_1, q_2\}, \emptyset)$ with $\delta(q_1, a!, q_2)$ and $\delta(q_2, a?, q_1)$. It is clear that (P_i, A) is consistent for each $i \in \{1, 2, 3\}$. Thus, our local consistency checks may not imply stuck-freedom.

5 Implementation

5.1 A Consistency Checker for BPEL

We have implemented BPELCheck, a consistency verifier for BPEL web services in the Sun NetBeans Enterprise Service Pack. The implementation has three parts: a Java front-end converts the NetBeans BPEL internal representation into a process in our core language and also reads in the conversation automata for the peers, an OCaml library compiles the consistency verification problem into a satisfiability instance, and finally, the decision procedure Yices [11] is used to check for conformance, and in case conformance fails, to produce a counterexample trace.

Optimizations We implement several optimizations of the basic algorithm from Section 4.2. The main optimization is a partial-order reduction that only considers a particular interleaving for the parallel composition of processes. For a process P , let $\text{Peers}(P)$ be the set of peers which exchange messages with P . Formally, $\text{Peers}(P)$ is defined by induction:

$$\begin{aligned} \text{Peers}(\text{skip}) &= \emptyset & \text{Peers}(\text{throw}) &= \emptyset & \text{Peers}(m \bowtie i) &= \{i\} \\ \text{Peers}(P_1 \otimes P_2) &= \text{Peers}(P_1) \cup \text{Peers}(P_2) & \otimes &\in \{\square, ;, \parallel, \triangleright\} \end{aligned}$$

Define two processes P_1 and P_2 to be *independent* if (1) neither process has a `throw` subprocess that escapes the scope of the process, and (2) $\text{Peers}(P_1) \cap \text{Peers}(P_2) = \emptyset$. Independent processes talk to disjoint peers, so if they run in parallel, the state of the conversation automata is the same for any interleaving. Thus, for independent processes, instead of constructing the interleaved formula for parallel composition, we construct the sequential composition of the two processes. This reduces the number of case splits in the satisfiability solver. In our experience, most parallel composition of processes in BPEL web services satisfy the independence property.

5.2 Extensions to the Core Language

In addition to our core language, BPEL has a set of other source-level constructs such as compensations, synchronizing flows, and (serial and parallel) loops. We

now show how we can extend the consistency verification algorithm to handle these constructs. We sketch the intuition, but do not give the formal details.

Variables. BPEL allows processes to have local variables. Our core language can be augmented with state variables and *assignment* and *assume* constructs. An assignment stores a computed value in a variable, and an assume statement continues execution iff the condition in the assume is true. The operational semantics of processes is extended in the standard way [23] to carry a *variable store* mapping variables to values. Assignments update the store, and assume conditions continue execution if the condition evaluates to true under the current store. Similarly, we can augment the construction of the consistency predicate to track variable values in the state variables, and model their updates in the transitions. We assume a finite-width implementation of integer data, which allows all constraints on integer variables to be reduced to Boolean reasoning.

Links. BPEL allows processes executing in parallel to synchronize through *links* between concurrently executing flows. A link between process P_1 and process P_2 means that P_2 can start executing only after P_1 has finished. We model links using Boolean variables for each link that are initially *false*, then set to *true* when the source activity terminates. We guard each target process with the requirement that the link variable is *true*.

Loops. BPEL processes can also have sequential or parallel loops. While in general the presence of loops makes the theoretical complexity of the problem go up from co-NP complete to PSPACE-complete, we have found that in practice most loops can be unrolled a fixed finite number of times. For example, once the peers are fixed, all loops over sets of peers can be unrolled. We plan to add support for loops to our tool in the near future, using simple syntactic heuristics to unroll loops a minimal number of times.

Compensations. BPEL allows processes to declare *compensation handlers* that get installed when a process completes successfully and get run when a downstream process fails. Our core language can be extended with compensations by adding the syntax $P_1 \div P_2$ to denote a process P_1 compensated by P_2 . The semantics is modified by carrying a *compensation stack*: the semantics of $P_1 \div P_2$ runs P_1 , and if it terminates normally, pushes the process P_2 on the compensation stack. If a process P terminates abnormally, the processes on the compensation stack are executed in last-in first-out order [6, 7, 4].

We can extend our construction of the consistency predicate by tracking compensations. With each process, we now associate both a depth (the number of steps in a normal evaluation) and a *compensation depth*, the number of compensation steps if the process terminates abnormally. The sum of the depth and the compensation depth is polynomial in the size of the process, thus the consistency predicate remains polynomial in the size of the process. The construction of the consistency predicate becomes more complicated as there is a forward process that encodes the transitions of the forward (normal) execution as well as constructs the compensation stack, and a compensation execution that executes the compensation actions on abnormal termination. We have not yet implemented compensations in our tool.

Process	Result	Proc size	States	Spec size	Time (ms)
Store 1	pass	31	18	14	408
Store 2	fail	31	18	14	339
Store 3	fail	31	18	14	384
Travel	pass	38	12	22	375
Auction	pass	15	9	11	245
ValidateOrder	fail	51	17	1	448

Figure 7: Experimental Results. “Result” is the result returned by BPELCheck, “States” is the total number of states across all peer automata, and “Spec size” is the size of the consistency predicate.

6 Case Studies

To evaluate our tool, we ran it on several example BPEL processes. *Store* implements the store process example of Section 2. We also implemented the two error scenarios described at the end of Section 2.

Travel is a travel agency example from [19]. This process attempts to reserve transportation and a hotel for a trip. It first tries to reserve a flight, and if this fails, it tries to reserve a train. If one of the transport reservations succeeds, the process then attempts to reserve the hotel. If the hotel reservation fails, the transport reservation must be canceled. The consistency predicate checks that either all conversations were left in a **nochange** state or the hotel reservation succeeded along with one of the two transport reservations.

Auction is from the BPEL specification [12]. The process waits for two incoming messages — one from a buyer and one from a seller. It then makes an asynchronous request to a registration service and waits for a response. Upon receiving a response, it sends replies to the buyer and seller services. Each interaction in this process is a simple request-reply pair. Our specification checks that every request has a matching reply.

ValidateOrder is an industrial example provided by Sun Microsystems. It accepts an order, performs several validations on the order, updates external systems, and sends a reply. If an error occurs, a message is sent to a JMS queue. Using BPELCheck, we found an error in this process. In BPEL, each *request* activity should have a matching *reply* activity, enabling the process to implement a synchronous web service call. However, the *ValidateOrder* process does not send a reply in the event that an exception occurs.

Figure 7 lists the results of running these examples through BPELCheck. In each case, we obtained the expected result. We measure the size of a process as the number of atomic actions plus the number of composition operators, when translated to our core language. We compute the size of consistency predicates by summing the number of atomic predicates and boolean connectives. These experiments were run on a 2.16 Ghz Intel Core 2 Duo laptop with 2 GB of memory using MacOS 10.5. The running times were all less than a second,

validating that this approach works well in practice. In general, the running times were roughly linear with respect to input size.

7 Related Work

Verification of web service compositions (and the associated specification languages) has generally taken one of two perspectives: that of an individual process, where the specification constrains the valid sequences of steps in a process execution, and that of a conversation, where the specification constrains the inter-process messaging protocols. A goal of our work is to unify these two approaches.

Verifying Individual Processes.. Web service compositions can be checked by extending standard model checking approaches to flow composition languages. For example, in [21], programs written in StAC, a core language for service compositions, are translated to labeled transition systems for verification against CTL properties using the XTL model checker. To obtain a finite system, only tail recursion is supported and loop iterations are bounded.

Other approaches consider only the sets of actions called and not their relative orderings. Consistency interfaces [2] define, for each method/result pair of a service, the messages called by that method and their associated results. Sub-specifications can be combined using union and intersection operators. Our previous work on *set consistency* [13] defines predicates over the sets of successful calls made by a process. Non-execution, failed calls, and compensated calls are all treated as equivalent, resulting in compact specifications. Conversational consistency extends this work by defining predicates over conversation states rather than individual actions.

Global Process Interactions.. Specifications may also focus on the messages exchanged between processes. Such specifications can be global, tracking messages from the perspective of an omniscient viewer of the entire composition, or local, specifying only the requirements for a single process. *Conversation specifications* [5, 15, 17] are an important example of the former case: such specifications consist of an automata representing the sequence of message sends across all the processes in a composition. Each peer in a composition can be checked to see whether it conforms to its role in the global conversation. In addition, temporal logic specifications can be checked against the conversation.

Web Services Analysis Tool (WSAT) [16] checks BPEL processes against conversation specifications and was the initial inspiration for our work. WSAT compiles processes into automata, which are then model checked using SPIN. Our approach is local: we avoid building the product space using conversation automata. This is also a practical requirement as the source code for all peer processes may not be available. Finally, instead of an enumerative model checker, our analysis reduces the reachability analysis to SAT solving. In addition to conformance checks, WSAT can perform checks for *synchronizability* (the conversation holds for asynchronous as well as synchronous messaging) and *realizability* (a conversation which fully satisfies the specification can actually

be implemented. The BPEL standard uses WSDL to describe the interfaces of each service, which assumes a synchronous transport (HTTP). However, vendor-specific extensions permit the use of asynchronous transport layers, such as JMS. Thus, this is a relevant issue for our tool. We hope to define compositional algorithms for determining synchronizability and realizability which can be used by BPELCheck.

Message Sequence Charts, although less expressive than automata, can also be used to specify global process interactions. In [14], process interactions are specified using Message Sequence Charts and individual processes specified using BPEL. Both definitions are translated to automata (the BPEL translation obviously over-approximating the actual process's behavior) and checked against each other.

Global specifications make it easier to check for non-local properties such as deadlock and dependencies across multiple processes (e.g. process A sends a request to process B , which forwards it to C , which sends a response back to A). However, such specifications limit the ability to abstract the details of a given process, particularly the interactions a service has with peers to complete a request. For example, in the store scenario of Section 2, each warehouse may need to coordinate with other services in order to ensure the delivery of a requested product. However, this may not be of interest to the store and bank, and ideally should be left out when verifying their conversations. In addition, many services may be implemented locally and not have external dependencies. In this situation, a global specification is overkill and reduces the re-usability of a service specification.

Local Process Interactions.. Message exchanges can also be specified with respect to a single process. On the practical side, Web Services Conversation Language (WSCL) [1] describes web services as a collection of *conversations*, where a conversation is an automata over *interactions*. Interactions represent message exchanges with a peer and can be either *one-way*, *send-receive*, or *receive-send*. This language is very similar to our conversation automata, and we could conceivably use WSCL instead of our chosen automata format (with the addition of nochange/commit labels for the states).

On the more theoretical side, the interactions of a web service can be specified using *session types* [25, 20], which provide a core language for describing a sequence of interactions over a channel and a type system for checking compatibility between the two sides of a conversation. The interaction language includes the passing of values and channel names, which are not currently modeled by our conversation automata. Compositional proof principles have been studied for message passing programs modeled in the π -calculus and interactions coded in CCS [10, 24]. They introduce a conformance notion that ensures services are not stuck.

Web service contracts [8, 9] specify the allowable iterations of a web service using a subset of CSS (Calculus of Communicating Systems). These contracts have a flexible subtyping relation for determining when a service satisfying a given contract can be used where a different contract is specified.

8 Conclusion

We make two contributions in this paper. First, we formalize a notion of consistency for interacting web services that can be locally checked, given the code for a process and conversation automata specifying the interactions with its peers. Second, we provide a tool for BPEL process developers integrated within the NetBeans IDE that statically checks BPEL processes for consistency violations.

We are pursuing several future directions to make BPELCheck more robust and usable. First, we abstract the data in the services to only track local variables of base type. Services typically interact through XML data, and processes query and transform the XML through XPath and XSLT. We shall integrate reasoning about XML [15] in our tool. Further, we assume *synchronous* messaging, but certain runtimes can provide asynchronous (queued) messages, for which an additional synchronizability analysis [17] may be required.

References

- [1] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, and et al. Web services conversation language (WSCL) 1.0. Technical report, World Wide Web Consortium, March 2002. <http://www.w3.org/TR/wscl10/>.
- [2] D. Beyer, A. Chakrabarti, and T. Henzinger. Web service interfaces. In *WWW '05*, pages 148–159. ACM, 2005.
- [3] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS 99: Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1579, pages 193–207. Springer, 1999.
- [4] R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL '05*, pages 209–220. ACM, 2005.
- [5] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *WWW '03*, pages 403–410. ACM, 2003.
- [6] M. Butler and C. Ferreira. A process compensation language. In *IFM '00*, pages 61–76. Springer, 2000.
- [7] M. Butler and C. Ferreira. An operational semantics for StAC, a language for modelling long-running business transactions. In *Coordination '04*, volume 2949 of *LNCS*, pages 87–104. Springer, 2004.
- [8] S. Carpineti, G. Castagna, C. Laneve, and L. Padovani. A formal account of contracts for web services. *Web Services and Formal Methods*, pages 148–162, 2006.
- [9] G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. In *POPL '08*, pages 261–272. ACM, 2008.
- [10] S. Chaki, S.K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. In *POPL '02*, pages 45–57. ACM, 2002.
- [11] B. Dutertre and L. de Moura. The yices smt solver. Tool paper at <http://yices.csl.sri.com/tool-paper.pdf>, August 2006.

- [12] T. Andrews et al. Business process execution language for web services, May 2003. <http://dev2dev.bea.com/webservices/BPEL4WS.html>.
- [13] J. Fischer and R. Majumdar. Ensuring consistency in long running transactions. In *ASE '07*, pages 54–63. ACM, 2007.
- [14] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *ASE '03*. IEEE, 2003.
- [15] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *WWW '04*, pages 621–630. ACM, 2004.
- [16] X. Fu, T. Bultan, and J. Su. Wsat: A tool for formal analysis of web services. In *CAV '04*, pages 510–514, 2004.
- [17] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Trans. on Softw. Eng.*, 31(12):1042–1055, December 2005.
- [18] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD '87*, pages 249–259. ACM, 1987.
- [19] C. Hagen and G. Alonso. Exception handling in workflow management systems. *IEEE Trans. Softw. Eng.*, 26(10):943–958, 2000.
- [20] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, pages 122–138. Springer-Verlag, 1998.
- [21] J. Augusto, M. Leuschel, M. Butler, and C. Ferreira. Using the extensible model checker XTL to verify StAC business specifications. In *AVoCS '03*, 2003.
- [22] R. Milner. *A Calculus of Communicating Systems*. LNCS 92. Springer-Verlag, 1980.
- [23] J.C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
- [24] Sriram K. Rajamani and Jakob Rehof. Conformance checking for models of asynchronous message passing software. In *CAV*, LNCS 2404, pages 166–179. Springer, 2002.
- [25] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE '94*, pages 398–413, London, UK, 1994. Springer-Verlag.