The Consistency of Web Conversations

Jeffrey Fischer, Rupak Majumdar, Francesco Sorrentino

***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.

I. 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 [9]. 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 [3]. 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 (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

This research is sponsored in part by the NSF grants CCF-0546170 and CCF-0702743.

J. Fischer and R. Majumdar are with the Computer Science Department, University of California Los Angeles, USA e-mail: ({fischer,rupak}@cs.ucla.edu). F. Sorrentino is with the Computer Science Department, University of Salerno, Italy e-mail: sorrentino@dia.unisa.it

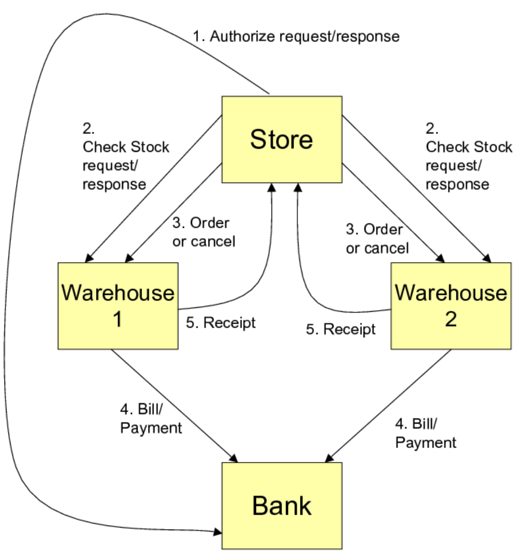

Fig. 1. Peers interactions for e-commerce example.

similar to the reduction used in bounded model checking [2]. We describe and formalize our algorithm in an associated technical report [6]. This report also includes a more detailed comparison to related work.

We have implemented our algorithm for checking consistency of BPEL processes in a tool called BPELCheck. 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 [5] as well as an industrial example from Sun Microsystems. In all cases, our tool was able to prove or disprove consistency within a second.

II. Example

To illustrate our approach to specifying and verifying business processes, we will use a simple e-commerce example, inspired by the one in [3]. 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.

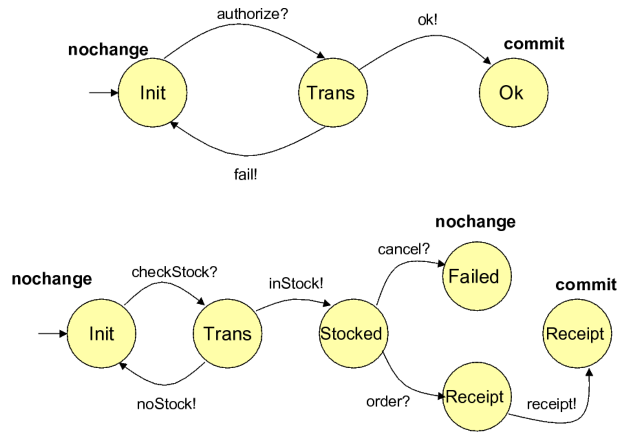

Fig. 2. Automata for store-bank conversation (top) and store-warehouse conversation (bottom).

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. $\mathsf{comm}(A)$ is `true` when the process terminates with peer A's conversation in a state labeled `commit`. $\mathsf{nochange}(A)$ 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:

$$\begin{array}{l}(\mathsf{nochange}(Bank) \ \rightarrow \\ \quad (\mathsf{nochange}(Warehouse1) \wedge \mathsf{nochange}(Warehouse2))) \ \wedge \\ ((\mathsf{comm}(Warehouse1) \vee \mathsf{comm}(Warehouse2)) \ \rightarrow \\ \quad (\mathsf{comm}(Bank) \wedge \mathsf{comm}(Warehouse1) \wedge \\ \quad \mathsf{comm}(Warehouse2)))\end{array}$$

Store process implementation. 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))) □
      (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.

III. Consistency Verification

Let $WS = (P, \langle C_1, \ldots, C_k \rangle)$ be a web service and s be a vector of all the conversation states (s_1 is the state of C_1, ...). A web service WS is *consistent* with respect to a conversational consistency predicate ψ if every complete run of WS is consistent (leaves each conversation automaton in either a `nochange` or `committed` state) and for every terminating run with a last state of s, 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.

We now briefly sketch our consistency verification algorithm. A full description of the algorithm may be found in [6]. Given a web service WS 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. This is done in three steps.

First, given a process P, we construct by induction a sequence of transition predicates that represent the conversation automaton transitions in possible executions of P. For example, if the vector of conversations states at the start and end of a step are represented by s and t respectively and automaton i transitions from state q to state q' (due to a message send or receive), we assert that $s_i = q$, $t_i = q'$, and, for all other automata $j \neq i$, we have $t_j = s_j$. The effects of process composition operators are represented by combining the formula for each subprocess. For example, nondeterministic choice can be represented by conjoining the subformulas of the two subprocesses. A naive encoding of parallel composition could lead to an exponential formula due to the possible interleavings of two processes. We avoid this by using dynamic programming to memoize sub-executions. To encode exception handling, we add a boolean variable to our state vector. This variable is initially `false`, set to `true` when an exception is thrown, and reset when the exception is caught.

We tie together these transitions by asserting that, for each step i in the possible executions of a process, the end states of step i are equal to the start states of step $i+1$. Finally, we constrain the executions to start in the initial state, and check whether at the end, the state is consistent according to the consistency specification (via boolean implication).

IV. Case Studies

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 [4] is used to check for conformance, and in case conformance fails, to produce a counterexample trace.

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

$Travel$ is a travel agency example from [10]. 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 [5]. 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 3 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.

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

Fig. 3. 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.

V. 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 [7] in our tool. Further, we assume *synchronous* messaging, but certain runtimes can provide asynchronous (queued) messages, for which an additional synchronizability analysis [8] 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] 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.

[3] 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.

[4] B. Dutertre and L. de Moura. The yices smt solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[5] T. Andrews et al. Business process execution language for web services, May 2003. http://dev2dev.bea.com/webservices/BPEL4WS.html.

[6] J. Fischer, R. Majumdar, and F. Sorrentino. Conversational Consistency. Technical Report TR-080018, UCLA Computer Science Department, July 2008.

[7] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *WWW '04*, pages 621–630. ACM, 2004.

[8] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Trans. on Softw. Eng.*, 31(12):1042–1055, December 2005.

[9] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD '87*, pages 249–259. ACM, 1987.

[10] C. Hagen and G. Alonso. Exception handling in workflow management systems. *IEEE Trans. Softw. Eng.*, 26(10):943–958, 2000.