Modular Verification of Web Services Using Efficient Symbolic Encoding and Summarization [*]

Fang Yu
University of California, Santa Barbara
yuf@cs.ucsb.edu

Chao Wang
NEC Laboratories America
chaowang@nec-labs.com

Aarti Gupta
NEC Laboratories America
agupta@nec-labs.com

Tevfik Bultan
University of California, Santa Barbara
bultan@cs.ucsb.edu

ABSTRACT

We propose a novel method for modular verification of web service compositions. We first use symbolic fixpoint computations to derive conditions on the incoming messages and relations among the incoming and outgoing messages of individual BPEL web services. These pre- and post-conditions are accumulated and serve as a repository of summarizations of individual web services. We then compose the summaries of the invoked BPEL services to model external invocations, resulting in a scalable verification approach for web service compositions. Our technical contributions include (1) an efficient symbolic encoding for modeling the concurrency semantics of systems having both multi-threading and message passing, and (2) a scalable method for summarizing concurrent processes that interact with each other using synchronous message passing, along with a modular framework that utilizes these summaries for scalable verification.

Categories and Subject Descriptors: D.2.4 [Software/ Program Verification]: Model checking

General Terms: Verification

Keywords: Modular Verification, Summarization, BPEL

1. INTRODUCTION

The increasing interest in web-based business process management has heightened the need for development of automatic verification techniques and tools for analyzing complex concurrent behaviors among web services. Web services play an important role in web-based business process management, serving as basic building blocks of inter-organizational interaction and cooperation. Business Process Execution Language (BPEL), used to orchestrate web services, is one of the standard languages designed to enable universal interoperability. A BPEL process can dynamically invoke external services asynchronously or synchronously, as well as *fork* and *join* concurrent threads internally. A composite web service implemented in BPEL can thus be viewed as a distributed system with both multi-threading and message passing. These concurrent language constructs give BPEL processes the ability to execute complex business tasks. However, they also result in complex concurrent behaviors that make the web service composition prone to errors and difficult to analyze.

To tackle the difficult problem of analyzing a composite web service, a widely adopted approach is modeling individual processes as variants of communicating finite state machines [15, 16, 12, 23] or variants of Communicating Sequential Processes (CSP [18]) [14, 20]. In these approaches, the composed system is modeled by a product automaton, where the state space is exponential in the number of component web services. While model checking has been used to analyze the concurrent behaviors of such composite systems, it suffers from the well-known state space explosion problem. Since languages like BPEL are designed for composing large distributed systems, state explosion limits the application of such verification techniques.

We propose a scalable method for analyzing web service composition which is based on a symbolic encoding of the interleaving execution semantics of a BPEL process and a summarization algorithm for concurrently running processes. The process summaries are utilized in a modular verification framework. We also introduce an intermediate graph representation, called *Concurrent Process Graph (CPG)*, to model the interacting processes. CPG is a natural extension of the standard control flow graph, with special nodes added to handle concurrency. It provides a uniform representation of a set of BPEL processes and facilitates a simple definition of the formal semantics.

Summarizing concurrently running BPEL processes is not a trivial task, since it involves handling both internal multi-threading and external message passing. Specifically, there are two concurrency related features in BPEL. The first one is the FLOW construct, which creates multiple threads that are executed concurrently within a process. We model the concurrent execution of activities associated with different threads using the interleaving semantics. We propose a disjunctive symbolic representation of the transition relation of the multi-threaded component to avoid the unnecessary addition of stuttering transitions often seen in a more conventional conjunctive representation. This makes the symbolic reachability analysis of a single component more efficient in practice.

The second concurrent feature comes from the external service invocation, i.e. when one process invokes another by message passing. For synchronous invocation, the invoker waits for the invoked process to finish before it continues. For asynchronous invocation, the invoker executes in a non-blocking fashion and proceeds forward, waiting for the reply at a future point. We propose summa-

[*]This work is supported in part by NSF grants CCF-0614002 and CCF-0716095.

rization of the invoked service as a weakest pre-condition, strongest post-condition pair. Since the invoker and invokee processes are running in parallel and may share multiple messages, a naive approach similar to sequential procedure call summarization does not work, e.g., read-write conflicts over common variables may invalidate the summaries. We address this problem by adding a special set of auxiliary variables that store the *snapshots* of the messages at the time of send/receive events; we derive and compose the summaries using these auxiliary variables. An essential observation is that, in service-oriented architecture, the interaction between services are limited to messages (without any shared variables) and hence concise summarization of processes is often achievable.

We have implemented these techniques in a prototype tool called Concurrent System Verifier (CSV), a modular static verifier for analyzing concurrent processes. Instead of building a monolithic model through in-lining or automata product composition, we analyze processes individually and perform modular verification by composing summaries of processes. The tool automatically parses BPEL and WSDL, constructs the corresponding CPGs and transition relations, computes process summaries, composes summaries of the invoked services, and checks assertions and safety properties.

2. RELATED WORK

Below we discuss some of the related work on the verification of BPEL processes and modular verification techniques.

BPEL Verification.

Niels *et al.* [21] developed a complete Petri-Net semantics for BPEL. They model interacting behaviors of BPEL processes as Workflow Nets (WFN), a Petri Net where tokens are bounded. They further characterize proper interactions of environment as an operating guideline, and ensure the controllability based on synchronization and production over Petri Nets.

Foster *et al.* [14] use Labeled Transition System Analyzer (LTSA) based on a process algebra to check compatibility of web service compositions in BPEL. Huynh [20] also presents a mapping from BPEL processes to BPE-processes, which can subsequently be analyzed by the Concurrency Workbench tool [5] to check if it satisfies CTL* properties.

Fu *et al.* [15, 16] perform LTL model checking by translating BPEL to Promela and feeding it to SPIN [19]. BPEL processes are represented as guarded automata, where XML data and XPath expressions are interpreted precisely by bounding the XML Schema types. Nakajima [23] represents BPEL processes as Extended Finite-state Automata and also uses SPIN for LTL model checking.

Fault handling and compensation handling in BPEL can be analyzed by translating them to timed automata, which are then analyzed using UPPAAL [25]. To deal with the timing aspects of BPEL like time-outs, an abstract notion of global system time is introduced [12]. Haddad *et al.* [17] further extend this work from discrete time to real time by using timed automata to model the XLANG activities.

In most of the works discussed above, concurrent processes are considered as separate entities (e.g., automata) and concurrency is modeled by a parallel product of these entities. This approach has scalability problems due to the well-known *state space explosion* problem, i.e., the number of states of a system in the worst case is exponential in the number of concurrent components.

Modular Verification.

Modular verification techniques [1, 6, 4, 27, 13, 28] try to assuage the state space explosion problem by decomposing the verification task.

In order to achieve modular verification of web services, Beyer *et al.* [2] proposed web service interfaces that specify proper sequences of service invocations and the corresponding return values. These interfaces can then be used to check whether two services are composable, and whether two services are substitutable. However, this work does not address how to automatically extract the service interfaces from the service implementations.

Duan *et al.* [8, 9] present a weakest pre-condition and strongest post-condition semantics for a subset of BPEL activities. However, their derivation of pre- and post-conditions over concurrent threads relies on the explicit enumeration of interleaved thread executions and, hence, is not efficient for handling the interleaving semantics of `flow` activities (pairs of fork and join), an essential feature of BPEL processes. We address this problem by using a symbolic fixpoint computation with disjunctive symbolic encoding.

The ability to summarize concurrent processes is fundamental to a modular verification framework. Esparza and Podelski generalized the inter-procedural analysis for summarizing sequential procedures to parallel programs [10] using process algebra, in which states are specified as terms and transitions are specified via rewriting rules. They showed that the extension of the inter-procedural setting to parallel programs does not increase the complexity of computing transition closures.

When concurrent executions are allowed to interfere with each other, the pre- and post-conditions as summaries are no longer adequate. To address this problem, Qadeer *et al.* [24] proposed a transaction-based summarization. Each transaction is an atomic region such that for any execution with interference, there is an execution without interference which gives the same result. Based on this, one can analyze each atomic region sequentially.

The main difference between our work and the earlier results on summarization in the presence of concurrency [10, 24] is that, we summarize message-passing processes rather than shared-memory threads. In BPEL web services, remote processes communicate solely by sending and receiving messages without access to any shared variables. A process may consist of multiple local threads that share common variables among themselves; however, threads from the same process do not communicate with each other via messages. Therefore, processes form a natural functional boundary; summaries (pre- and post-conditions of processes) can be expressed solely in terms of the incoming and outgoing messages.

Finally, compositional reasoning for concurrent programs has been explored in other contexts [6, 4, 27]. Cobleigh *et al.* [6] apply assume-guarantee reasoning in the LTSA tool to achieve compositional verification. The assumption of the environment is iteratively refined by counterexamples. In Magic [4] and Comfort [27], Chaki *et al.* employ predicate and action-guided abstractions within a counter-example guided abstraction refinement scheme. Parallel components are abstracted by conservatively aggregating states together, and are refined later to eliminate spurious counterexamples. Compared to those iterative refinement approaches, we perform efficient verification on concrete states by leveraging the distinction between shared memory and message-passing structures.

3. MODELING BPEL PROCESSES

We propose concurrent process graphs (CPGs) for modeling BPEL processes. CPGs are an extension of the standard control flow graphs used for modeling sequential programs, with additional features added to model concurrency constructs. The CPG representation can model both shared-variable multi-threading and processes communicating via messages. Hence, CPGs can serve as an intermediate representation of individual BPEL processes as well as composite services with multiple interacting BPEL processes.

3.1 Concurrent Process Graphs

DEFINITION 1. *A concurrent process graph (CPG) is a tuple* $\langle N, \mathit{Var}, \mathit{Chl}, E\rangle$*, such that*

- N *is a finite set of nodes,*
- *Var is a finite set of variables,*
- *Chl is a finite set of communication channels, and*
- $E \subseteq N \times N \times \mathit{Guard} \times (\mathit{Assign} \cup \mathit{Send} \cup \mathit{Receive} \cup \{-\})$ *is a finite set of edges, where*
  - *Guard is a set of conditional expressions over Var,*
  - *Assign is a set of assignment statements over Var,*
  - $\mathit{Send} \subseteq \mathit{Chl} \times \{!\} \times \mathit{Exp}$ *is a set of send activities, where Exp is a set of expressions over Var,*
  - $\mathit{Receive} \subseteq \mathit{Chl} \times \{?\} \times \mathit{Var}$ *is a set of receive activities.*

We characterize the set of nodes and edges in a CPG below.

Node Set.

The set of nodes in a CPG is defined as the union of three disjoint sets of nodes: $N = N_n \cup N_f \cup N_j$. A node $n \in N_n$ is a `normal` node, which models the sequential execution of a thread. One of the incoming edges must be executed before control is transferred to this node, and from this node only one of the outgoing edges can be executed. A node $n \in N_f$ is a `fork` node, which represents the starting point for the parallel execution of multiple threads. One of the incoming edges must be executed before control is transferred to this node, and from this node all the outgoing edges are executed simultaneously in one step. Finally, a node $n \in N_j$ is a `join` node, which represents the end point for the parallel execution of multiple threads. All incoming edges must be executed (simultaneously in one step) before control is transferred to this node, and from this node only one of the outgoing edges can be executed.

The two distinct sources of concurrency in a composite web service can be modeled by `fork` and `join` nodes. The first one comes from the FLOW activity inside a BPEL process, where all child activities run concurrently. The second one comes from the concurrent execution of interacting BPEL processes. Although there is no explicit BPEL language construct for the second case, this type of concurrency can be modeled by adding a pair of `fork` and `join` nodes: the fork node has outgoing edges to the entry points of the individual BPEL processes, and the join node has incoming edges from the exit points of the individual BPEL processes.

Edge Set.

An edge $e \in E$ is associated with a tuple $\langle n_1, n_2, g, \alpha\rangle$, where $n_1 \in N$ is the source node, $n_2 \in N$ is the target node, $g \in \mathit{Guard}$ is the guard, and $\alpha \in \mathit{Assign} \cup \mathit{Send} \cup \mathit{Receive} \cup \{-\}$ is the action, which can be an `assignment` ($\alpha \in \mathit{Assign}$), a `send` ($\alpha \in \mathit{Send}$), a `receive` ($\alpha \in \mathit{Receive}$), or a `no-op` ($\alpha \in \{-\}$).

Let exp be an expression over variables in *Var*. An action $\alpha \in \mathit{Assign}$ is an `assignment` of the form $v := exp$, which sets the next state value of v to the current value of exp.

An action $\alpha \in \mathit{Send}$ is a `send` of the form $ch!exp$, which sends the current value of exp as a message through the channel ch. An action $\alpha \in \mathit{Receive}$ is a `receive` of the form $ch?v$, which sets the next-state value of v to the value of the message received from the channel ch. A `send` and a `receive` are paired if they communicate through the same channel. We assume that both `send` and `receive` are blocking and the execution of the corresponding send and receive actions are synchronous (like CSP [18]). Whenever asynchronous communication is needed, i.e., when `send` is non-blocking and `receive` is blocking, we can model it by explicitly adding a buffer thread to the channel. For example, a channel with arbitrary delay can be modeled by renaming the channel of `send` ($ch!x$) and `receive` ($ch?y$) into $ch_1!x$ and $ch_2?y$ and then adding a separate buffer thread. The buffer thread consists of two nodes n_1, n_2, and two edges $e_1 = (n_1, n_2, \texttt{true}, ch_1?z)$ and $e_2 = (n_2, n_2, \texttt{true}, ch_2!z)$. Between the execution of e_1 and the execution of e_2, the buffer thread may introduce arbitrary delay (when the scheduler decides to execute threads other than this buffer thread). Note that in this example the size of the buffer is one. One can extend the construction to a bounded size buffer by adding more auxiliary variables to record the received messages.

The set of edges contains two types of edges $E = E_n \cup E_l$ where $E_l \cap E_n = \emptyset$. An edge $e \in E_n$ is a `normal` edge defined as above. An edge $e \in E_l$ is a `link` edge, which imposes a "happens-before" relation between its source and target nodes; that is, the outgoing edges of the target node cannot be executed before the execution of the incoming edges of the source node completes. Typically, the source and target nodes of a `link` edge belong to different threads of the same BPEL process, and the action of a `link` edge is `no-op`. As discussed below in section 3.2, `link` edges are defined particularly for modeling BPEL processes. In section 3.3, we show that one can remove `link` edges from a given CPG by adding auxiliary shared variables.

3.2 Translating BPEL Processes to CPGs

A composite web service consists of a set of interacting BPEL processes, each of which may have more than one thread. The root of a BPEL process is identified by the PROCESS keyword, which contains the actual workflow defined by the top activity. A BPEL *activity* can be a basic activity or a structured activity: Basic activities are RECEIVE, REPLY, INVOKE, ASSIGN, THROW, TERMINATE, WAIT, and EMPTY. Structured activities are SEQUENCE, SWITCH, WHILE, PICK, FLOW, SCOPE, COMPENSATE.

Figure 2 shows a sample BPEL specification [11] (with simplified BPEL code). The entire system consists of four interacting processes: *approval*, *approver*, *assessor*, and *customer*. The approval process is invoked by the customer, and invokes the assessor and approver processes within a FLOW activity. In this FLOW activity five threads are activated simultaneously. All processes are executed concurrently and interact with the approval process.

The activities RECEIVE, REPLY, INVOKE are related to the sending and receiving of messages. Specifically, a RECEIVE activity in BPEL directly maps to a CPG receive action ($ch?x$) and a reply activity (as well as an asynchronous INVOKE) in BPEL directly maps to the CPG send action ($ch!exp$). A synchronous INVOKE activity maps to a CPG send ($ch!exp$) which is then immediately followed by a CPG receive ($ch?x$); that is, after sending a message having the current value of exp to invoke a remote service, it immediately waits for the returning message in x. The ASSIGN activity maps directly to a CPG assignment action ($x := exp$). The TERMINATE, WAIT, EMPTY activities in BPEL can also be easily modeled by CPG nodes and edges.

The SEQUENCE activity in BPEL represents the sequential execution of its child activities and is modeled by nodes of normal type. The SWITCH, WHILE, SCOPE activities also map directly to corresponding structures in a standard (sequential) control flow graph. The FLOW activity in BPEL represents concurrent execu-

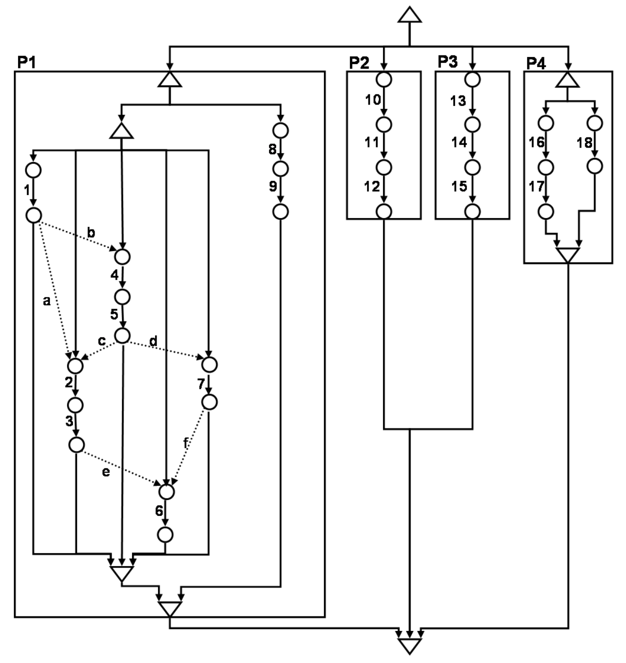


Figure 1: The CPG of the *loanapproval* example

l	(g,α)	l	(g,α)
a	(request.amount<4, $-$)	7	(true, approvalInfo.accept:='yes')
b	(request.amount≥4, $-$)	8	(true, catch?error)
c	(riskAssessment.risk='low', $-$)	9	(true, ch14!error)
d	(riskAssessment.risk≠'low', $-$)	10	(true, ch13?request)
e	(true, $-$)	11	(true, riskAssessment.risk:='low')
f	(true, $-$)	12	(true, ch31!riskAssessment)
1	(true, ch41?request)	13	(true, ch12?request)
2	(true, ch13!request)	14	(true, approvalInfo.accept:='yes')
3	(true, ch31?approvalInfo)	15	(true, ch24!approvalInfo)
4	(true, ch12!request)	16	(true, ch41!request)
5	(true, ch21?riskAssessment)	17	(true, ch41?approvalInfo)
6	(true, ch14!approvalInfo)	18	(true, catch?error)

Table 1: The guards and actions of the edges in Figure 1

tion of its child activities; it is mapped to a pair of `fork` and `join` nodes in the CPG. The PICK activity is similar to SWITCH, except that control may transfer to its child activities in a nondeterministic fashion.

Now we use the example in Figure 2 to show the modeling of BPEL concurrency constructs in CPG. The corresponding CPG is given in Figure 1. P1, P2, P3 and P4 model *approval*, *assessor*, *approver* and *customer* process, respectively. When drawing this graph, the following notation is used: ◯ denotes a `normal` node, △ denotes a `fork` node, and ▽ denotes a `join` node. There are up to ten concurrent executions in the CPG shown in Figure 1. Note that inside the *approval* and *customer* processes, the FAULT-HANDLER also contributes an execution thread that runs concurrently with the main flow of the process. The guards and actions of labeled edges are shown in Table 1. The edges labeled from 1 to 18 are `normal` edges, while the edges labeled from a to f are `link` edges. The modeled BPEL statements are also labeled in Figure 2.

A partial execution order on threads within a process can be specified by LINK attributes of the FLOW activity in BPEL. For example, LINK a is between the RECEIVE activity and the INVOKE activity for invoking remote process P3. The guard of LINK a is $(request.amount \geq 4)$, meaning that "the *assessor* process must be invoked after the RECEIVE activity completes, and when the request amount is greater than or equal to 4". As mentioned earlier, LINK can be modeled directly as a special CPG edge.

3.3 Constraints on the CPG Representation

Before we delve into how to perform symbolic verification on CPGs using the interleaving semantics in the next section, we impose several constraints on the CPG to ease the symbolic encoding. Note that this is done without loss of generality: If a given CPG does not satisfy these constraints, we rewrite it into a functionally equivalent form that satisfies the constraints.

Let $\langle N, \text{Var}, \text{Chl}, E\rangle$ be a CPG representing a BPEL process. For an edge $e = \langle n_1, n_2, g, \alpha\rangle$, we use $e.src$, $e.tgt$, $e.cond$, and $e.act$ to denote the source node, the target node, the guard, and the action of e, respectively. The set $n.E_{in} = \{e \mid e \in E, e.tgt = n\}$ denotes the set of incoming edges of n, and the set $n.E_{out} = \{e \mid e \in E, e.src = n\}$ denotes the set of outgoing edges of n.

We assume that the following constraints hold for all $n \in N$:

- $\forall n \in N_f$, $\forall e \in n.E_{out}$, we have $e.cond = \texttt{true}$ and $e.act = -$; otherwise, we insert a new node n_3 between $e.src$ and $e.tgt$ (Figure 3-left).
- $\forall n \in N_j$, $\forall e \in n.E_{out}$, we have $e.cond = \texttt{true}$ and $e.act = -$; otherwise, we insert a new node n_3 between $e.src$ and $e.tgt$ (Figure 3-middle).
- $\forall n \in N$, among the two sets $n.E_{in}$ and $n.E_{out}$, at most one set contains `link` edges, i.e., $n.E_{in} \cap E_l = \emptyset$ or $n.E_{out} \cap E_l = \emptyset$; otherwise, we split the node into n_{1a} and n_{1b} such that $n_{1a}.E_{in} = n_1.E_{in}$ and $n_{1b}.E_{out} = n_1.E_{out}$ (Figure 3-right).

We also remove the `link` edges in a given CPG by introducing new binary variables. For each `link` edge $e_a = (n_1, n_2, g, -) \in E_l$, we allocate a new binary state variable lk_a. Recall that $E = E_l \cup E_n$, where E_n is the set of `normal` edges.

- First, we add $lk_a := 1$ as an action to all $e \in E_n \cap n_1.E_{in}$.
- Second, we add $g \wedge lk_a = 1$ as a guard and $lk_a := 0$ as an action to all $e \in E_n \cap n_2.E_{in}$.

Note that lk_a is set to 1 upon the completion of the source node n_1, and lk_a is set to 0 upon the completion of the target node n_2. Furthermore, the execution of node n_2 is guarded additionally by $g \wedge lk_a = 1$. After these modifications, we remove the link edge from the CPG. After removing all `link` edges, we expand the set of variables to *Var* $\cup \{lk_a \mid e_a \in E_l\}$.

4. SYMBOLIC ANALYSIS OF CPGS

In this section we discuss verification of CPGs using symbolic reachability analysis [22]. In order to encode the interleaving semantics symbolically we identify the set of threads in a given CPG and introduce one program counter variable for each thread.

Specifically, we assume that all edges are executed by a set of concurrent threads *Thread*. For the purpose of tracking the thread execution, each thread $t_i \in$ *Thread* is associated with a distinct program counter (PC) variable pc_i. Let $P = \{pc_i \mid t_i \in \textit{Thread}\}$ denote the set of PC variables. We assume that each node $n \in N$ has a unique index, denoted as $n.id$. The domains of the PC variables satisfy the following constraint: $\forall pc \in P, \mathbf{dom}(pc) \subseteq \{n.id \mid n \in N\} \cup \{\bot\}$. If $pc_i = n.id$, then some $e \in n.E_{out}$ will be executed by thread t_i in the next step. If $pc_i = \bot$ then thread t_i is inactive. We assign each $n \in N$ a PC variable, denoted as $n.pc$,

```
        ⟨process Approval⟩
          ⟨flow ⟩
            ⟨links ⟩
a.            ⟨link name=receive-to-approval/⟩
b.            ⟨link name=receive-to-assess/⟩
c.            ⟨link name=setMessage-to-reply/⟩
d.            ⟨link name=assess-to-setMessage/⟩
e.            ⟨link name=assess-to-approval/⟩
f.            ⟨link name=approval-to-reply/⟩
            ⟨/links ⟩
1.          ⟨receive ⟩ . . .
            ⟨source linkName=receive-to-assess
              transitionCondition=. . ./⟩
            ⟨source linkName=receive-to-approval
              transitionCondition=. . ./⟩
            ⟨/ receive ⟩
            ⟨invoke ⟩
2.            inputVariable=. . .
3.            outputVariable=. . .
              ⟨target linkName=receive-to-approval /⟩
              ⟨target linkName=assess-to-approval /⟩
              ⟨source linkName=approval-to-reply/⟩
            ⟨/invoke ⟩
            ⟨invoke ⟩
4.            inputVariable=. . .
5.            outputVariable=. . .
              ⟨target linkName=receive-to-assess /⟩
              ⟨source linkName=assess-to-setMessage
                transitionCondition=. . ./⟩
              ⟨source linkName=assess-to-approval
                transitionCondition=. . ./⟩
            ⟨/invoke ⟩
6.          ⟨reply ⟩ . . .
              ⟨target linkName=setMessage-to-reply/⟩
              ⟨target linkName=approval-to-reply/⟩
            ⟨/ reply ⟩
7.          ⟨assign ⟩ . . .
              ⟨target linkName=assess-to-setMessage/⟩
              ⟨source linkName=setMessage-to-reply/⟩
            ⟨/ assign ⟩
          ⟨/flow ⟩
          ⟨faultHandler ⟩
8.          ⟨catch ⟩ . . . ⟨/catch ⟩
9.          ⟨reply ⟩ . . . ⟨/reply ⟩
          ⟨/faultHandler ⟩
        ⟨/process ⟩
        ⟨process Assessor⟩
          ⟨sequence ⟩
10.         ⟨receive ⟩ . . . ⟨/ receive ⟩
11.         ⟨assign ⟩ . . . ⟨/ assign ⟩
12.         ⟨reply ⟩ . . . ⟨/ reply ⟩
          ⟨/ sequence ⟩
        ⟨/process ⟩
        ⟨process Approver⟩
          ⟨sequence ⟩
13.         ⟨receive ⟩ . . . ⟨/ receive ⟩
14.         ⟨assign ⟩ . . . ⟨/ assign ⟩
15.         ⟨reply ⟩ . . . ⟨/ reply ⟩
          ⟨/ sequence ⟩
        ⟨/process ⟩
        ⟨process Customer⟩
          ⟨invoke ⟩
16.         inputVariable=. . .
17.         outputVariable=. . .
          ⟨/invoke ⟩
          ⟨faultHandler ⟩
18.         ⟨catch ⟩ . . . ⟨/ catch ⟩
          ⟨/faultHandler ⟩
        ⟨/process ⟩
```

Figure 2: The *loanapproval* example

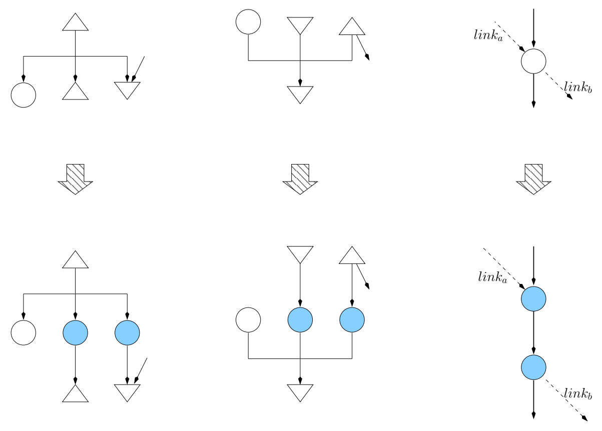


Figure 3: Rewriting `fork`, `join`, and `link`

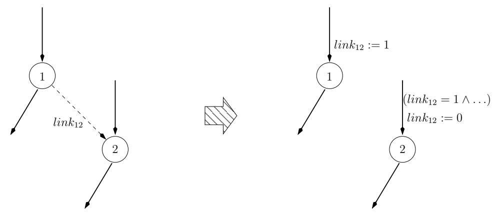


Figure 4: Rewriting the `link` nodes

such that $n.pc = pc_i$ if and only if some $e \in n.E_{out}$ is executed by thread t_i. In this case, we say node n belongs to thread t_i. We also use $n.tid$ to denote the id of the thread that n belongs to.

Note that the sets *Thread*, P, and $n.pc$ are not associated with the original CPG. In the following section, we propose a heuristic algorithm to discover a candidate set of threads and PC variables, and assign a PC variable to each node.

4.1 Thread Discovery and PC Variable Assignment

We assume that, in any CPG, thread creation and termination operations (corresponding to `fork` and `join`) are always nested; that is, if thread B is `fork`ed from thread A, then thread B should `join` back before thread A terminates. Whenever a CPG is produced from a BPEL process, this assumption is guaranteed to hold. The assumption significantly simplifies the algorithm for assigning PC variables. We can simply perform a Depth-First Search (DFS) of the graph, and assign new PC variables only when visiting the following nodes: (1) entry node of a process, and (2) every successor of a `fork` node, except for the first one. All other nodes, including the first successor of a `fork` node, belong to the same thread as their predecessor nodes. A `join` node belongs to the thread of a predecessor node that has the smallest thread index.

Algorithm 1 ASSIGN_PCVAR(G)

```
1: for each node $n \in N$ do
2:     $n.pc$ = NULL;
3:     $n.visited$ = 0;
4: end for
5: numPC = 1;
6: $P$ = {};
7: ASSIGN_PCVAR_DFS( $G$.entryNode );
```

Algorithm 2 ASSIGN_PCVAR_DFS(n_1)

```
1:  if $n_1.visited$ == 0 then
2:      $n_1.visited$ = 1;
3:      if $n_1.pc$ == NULL then
4:          $n_1.pc$ = numPC; // assigning a PC var
5:          $P = P \cup \{pc_{\texttt{numPC}}\}$;
6:          numPC = numPC +1;
7:      end if
8:      for each $e \in n_1.E_{out}$ do
9:          $n_2 = e.tgt$;
10:         if $n_2 \in N_f$ then
11:             if $e$ is the first edge in $n_1.E_{out}$ then
12:                 $n_2.pc = n_1.pc$;
13:             end if
14:         else if $n_2 \in N_j$ then
15:             if $n_2.pc$ == NULL or $n_2.pc > n_1.pc$ then
16:                 $n_2.pc = n_1.pc$;
17:             end if
18:         else
19:             if $n_2.pc$ == NULL then
20:                 $n_2.pc = n_1.pc$;
21:             end if
22:         end if
23:         ASSIGN_PCVAR_DFS( $n_2$ );
24:     end for
25: end if
```

The pseudo code is presented in Algorithms 1 and 2, where ASSIGN_PCVAR is the entry point of the procedure. In this procedure, $numPC$ (which is initialized to 1) represents the total number of PC variables (and, hence, the total number of threads) in the graph. The auxiliary field $n.visited$ is used to keep track of the visited nodes during the DFS. We store the result of this computation, that is, to which thread each node belongs, at $n.pc$. Recall that for $n \in N$, $n.pc$ stores the PC variable pc_i.

4.2 Disjunctive Transition Relation Encoding

In symbolic model checking, the model is represented as a tuple $\langle I, T, X\rangle$, where I is the symbolic representation of the initial states and T is the symbolic representation of the transition relation. X is the set of state variables, and X' contains the next-state copies of variables in X. Specifically, $X = \mathit{Var} \cup \{lk_a \mid e_a \in E_l\} \cup P \cup \{sel\}$, which includes BPEL variables, link variables, PC variables, and an auxiliary variable *sel* that we later use to model the interleaving semantics explicitly. Assume that the CPG has a unique entry node $n_1 \in N$, then I is defined as $(n_1.pc = n_1.id) \wedge \bigwedge_{pc_i \neq n_1.pc}(pc_i = \bot)$. In the initial state, all the yet-to-be-created threads have a PC value $\bot$. Furthermore, after a thread terminates through execution of a `join` edge, its PC value becomes $\bot$ again. The interleaving semantics is imposed by using a non-deterministic scheduler variable called sel, whose domain is the set of thread indices in the CPG. An edge $e \in E$, whose source node belongs to thread t_i ($e.src.pc = pc_i$), is executed only when $(sel = i)$. Also note that when an edge $e \in E$ is executed, state variables that are not assigned new values on this edge must retain their current values. According to the synchronous communication semantics, we model the synchronous execution of $ch!exp$ and $ch?x$ as if there is an assignment $x' := exp$ (the next-state value of x is current value of exp) added to T. This assignment (synchronous execution of send and receive) happens only when both threads are ready to communicate—when the PC of the send thread is at the source node of the send edge and the PC of the receive thread is at the source node of the receive edge. If one thread is ready for a send $ch!exp$ but the other thread is not yet ready for the corresponding receive $ch?x$ (or vice versa), no transition from these two threads will be executed since there is no corresponding transition formula in T.

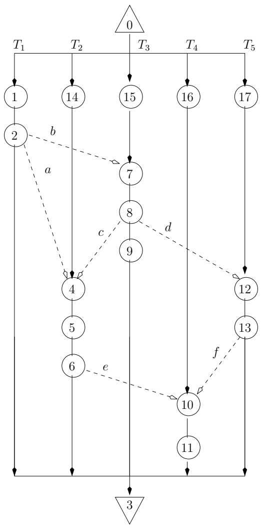
In the sequel, we use $e.X_W$ to denote the set of state variables being written to by an edge e. When an action $e.act$ is an `assignment`, e.g., $v := exp$, variable v belongs to $e.X_W$. When $e.act$ is a `receive`, e.g., $ch?x$, variable x belongs to $e.X_W$.

Now we present the algorithm for building the disjunctive transition relation T. Let $T = \bigvee_{e \in E} T_e$ where T_e is the transition relation for an individual edge $e \in E$. We start by iterating through the set E of CPG edges. The pseudo code of this procedure is given in Algorithm 3. Here $E_{visited} \subseteq E$ denotes the set of already visited edges. For each edge $e \in E$, we use $X_{visited} \subseteq X$ to denote the set of state variables that are assigned, either explicitly through actions or implicitly through control flow transition. We do not add a transition formula to T for send edges – these formulae are added when processing the corresponding receive edges.

Algorithm 3 ENCODE_INTERLEAVING(G)

```
1:  $T :=$ false;
2:  $E_{visited} := \{\ \}$;
3:  for each edge $e : (n_1, n_2, g, \alpha)$ in $(E \setminus E_{visited})$ do
4:      $E_{visited} := E_{visited} \cup \{e\}$;
5:      $X_{visited} := \{n_2.pc\} \cup e.X_W$;
6:      $T_s := (sel = n_1.tid)$;
7:      $T_e := (n_1.pc = n_1.id \wedge g \wedge n_2.pc' = n_2.id)$;
8:      if $n_1 \in N_f$ then
9:          for each edge $e_1 \in n_1.E_{out}, e_1 \neq e$ do
10:             $E_{visited} := E_{visited} \cup \{e_1\}$;
11:             $X_{visited} := X_{visited} \cup \{e_1.tgt.pc\}$;
12:             $T_e := T_e \wedge e_1.tgt.pc' = e_1.tgt.id$;
13:         end for
14:     else if $n_2 \in N_j$ then
15:         for each edge $e_2 \in n_2.E_{in}, e_2 \neq e$ do
16:             $E_{visited} := E_{visited} \cup \{e_2\}$;
17:             $X_{visited} := X_{visited} \cup \{e_2.src.pc\}$;
18:             $T_e := T_e \wedge e_2.src.pc = e_2.src.id \wedge e_2.src.pc' = \bot$;
19:         end for
20:     end if
21:     if $\alpha$ is assignment $v := exp$ then
22:         $T_e := T_e \wedge v' = exp$;
23:     else if $\alpha$ is receive $ch?x$ then
24:         if $\exists e_3 \in E, e_3.act = ch!exp$ then
25:             $E_{visited} := E_{visited} \cup \{e_3\}$;
26:             $X_{visited} := X_{visited} \cup \{e_3.tgt.pc\}$;
27:             $T_e := T_e \wedge x' = exp$;
28:             $T_e := T_e \wedge e_3.src.pc = e_3.src.id \wedge e_3.tgt.pc' =$
                $e_3.tgt.id$;
29:         else
30:             $T_e := T_e \wedge x' = *$;
31:         end if
32:     else if $\alpha$ is send $ch!exp$ then
33:         continue
34:     end if
35:     $T_e := T_e \wedge \Sigma_{v \in (X \setminus X_{visited})} v' = v$;
36:     $T := T \vee T_s \wedge T_e$;
37: end for
38: return $T$;
```

As an example, the result of applying the Algorithm 3 to part of the CPG in Figure 1 is shown in Figure 5.



$(n_0, n_1, \mathsf{true}, -)$	$sel = 1 \wedge pc_1 = 0 \wedge pc'_1 = 1 \wedge pc'_2 = 14 \wedge pc'_3 = 15 \wedge pc'_4 = 16 \wedge pc'_5 = 17 \wedge \ldots$
$(n_0, n_{14}, \mathsf{true}, -)$	$(visited)$
$(n_0, n_{15}, \mathsf{true}, -)$	$(visited)$
$(n_0, n_{16}, \mathsf{true}, -)$	$(visited)$
$(n_0, n_{17}, \mathsf{true}, -)$	$(visited)$
$(n_1, n_2, \mathsf{true}, -)$	$sel = 1 \wedge pc_1 = 1 \wedge pc'_1 = 2 \wedge (lk'_a = 1 \wedge lk'_b = 1) \wedge \ldots$
$(n_2, n_3, \mathsf{true}, -)$	$sel = 1 \wedge pc_1 = 2 \wedge pc_2 = 6 \wedge pc_3 = 9 \wedge pc_4 = 11 \wedge pc_5 = 13 \wedge pc'_1 = 3 \wedge pc'_2 = \perp \wedge \ldots$
$(n_4, n_5, \mathsf{true}, -)$	$sel = 2 \wedge pc_2 = 4 \wedge pc'_2 = 5 \wedge \ldots$
$(n_5, n_6, \mathsf{true}, -)$	$sel = 2 \wedge pc_2 = 5 \wedge pc'_2 = 6 \wedge (lk'_e = 1) \wedge \ldots$
$(n_6, n_3, \mathsf{true}, -)$	$(visited)$
$(n_7, n_8, \mathsf{true}, -)$	$sel = 3 \wedge pc_3 = 7 \wedge pc'_3 = 8 \wedge (lk'_c = 1 \wedge lk'_d = 1) \wedge \ldots$
$(n_8, n_9, \mathsf{true}, -)$	$sel = 3 \wedge pc_3 = 8 \wedge pc'_3 = 9 \wedge \ldots$
$(n_9, n_3, \mathsf{true}, -)$	$(visited)$
$(n_{10}, n_{11}, \mathsf{true}, -)$	$sel = 4 \wedge pc_4 = 10 \wedge pc'_4 = 11 \wedge \ldots$
$(n_{11}, n_3, \mathsf{true}, -)$	$(visited)$
$(n_{12}, n_{13}, \mathsf{true}, -)$	$sel = 4 \wedge pc_5 = 12 \wedge pc'_5 = 13 \wedge (lk'_f = 1) \wedge \ldots$
$(n_{13}, n_3, \mathsf{true}, -)$	$(visited)$
$(n_{14}, n_4, \mathsf{true}, -)$	$sel = 2 \wedge pc_2 = 14 \wedge pc'_2 = 4 \wedge (lk_a = 1 \wedge lk_c = 1) \wedge \ldots$
$(n_{15}, n_7, \mathsf{true}, -)$	$sel = 3 \wedge pc_3 = 15 \wedge pc'_3 = 7 \wedge (lk_b = 1) \wedge \ldots$
$(n_{16}, n_{10}, \mathsf{true}, -)$	$sel = 4 \wedge pc_4 = 16 \wedge pc'_4 = 10 \wedge (lk_e = 1 \wedge lk_f = 1) \wedge \ldots$
$(n_{17}, n_{12}, \mathsf{true}, -)$	$sel = 5 \wedge pc_5 = 17 \wedge pc'_5 = 12 \wedge (lk_d = 1) \wedge \ldots$

Figure 5: Encoding of the interleaving semantics for the sample CPG

4.3 Monolithic Symbolic Verification

For a BPEL process without any external service invocation, we can model the process as a CPG, build a monolithic symbolic representation and check its correctness by symbolic reachability analysis. For a composite web service in which a BPEL process invokes a set of externally defined BPEL processes, we can build a single CPG that includes all participating BPEL processes by adding a new *entry node* which is a `fork`, with outgoing edges to the entry nodes of all participating processes; at the same time, adding a new *exit node* which is a `join`, with incoming edges from the exit nodes of all participating processes. In the monolithic verification model, all the variables are treated as global variables and the model is treated as a closed system.

Given such a verification model, we can apply the standard symbolic fixpoint algorithm in model checking [22] for reachability analysis. Let R be the set of reachable states from I in the model; we start with $R = I$, and repeatedly compute $R \cup post(T, R)$, where $post(T, R)$ is the set of successor states of R. $post(T, R)$ is defined as $(\exists X.R(X) \wedge T(X, X'))[X/X']$, where $\exists X.$ denotes existential quantification on X, and $[X/X']$ denotes renaming X' with X. Maintaining the entire reachable state set R^i at every iteration i is costly. However, to detect convergence of this fixpoint computation, the algorithm needs to store the already reached states (in order to stop as soon as $R^{i+1} = R^i$). Let R^{i-1} and R^i be two reachable state sets at two consecutive steps; the set $R^i \setminus R^{i-1}$ is called the *frontier set* [26]. In computing $R^{i+1}, post(T, R^i \setminus R^{i-1})$ can be used instead of $post(T, R^i)$ to speed up the computation, if the set $(R^i \setminus R^{i-1})$ has a smaller symbolic representation.

In this paper, we apply a specialized symbolic search strategy called REACH_FRONTIER to improve reachability fixpoint computation [29]. It uses an augmented *frontier set* to detect convergence. In reachability computation, a frontier set consists of all the new states reached at the previous iteration; that is, $F^0 = I, F^i = post(T, F^{i-1}) \setminus F^{i-1}$. When the CPG is an acyclic graph, fixpoint computation can stop when F^i becomes empty. In the presence of cycles, we can identify a set of back edges $E_{back} \subseteq E$ in the CPG, whose removal will make the graph acyclic. Let $S_{back} = \bigvee_{e \in E_{back}} (e.src.pc = e.src.id)$ denote the state subspace associated with source nodes of the back edges. The set of already reached states that falls inside S_{back} is $S = R \cap S_{back}$; the emptiness of the set $(F \setminus R \cap S_{back})$ can be used to detect convergence.

We identify back edges in the CPG by a DFS starting from the entry node. If the CPG is acyclic, the post-order of DFS gives a topological order and all edges are from lower ranked nodes to higher ranked nodes. If the CPG has cycles, E_{back} is identified as the set of edges from higher ranked nodes to lower ranked nodes (w.r.t. the post-order of the DFS) and we label them as back edges. The removal of these edges makes the CPG a directed acyclic graph.

Our new reachability procedure in Algorithm 4 takes as parameters the state subspace Err (the error states) as well as S_{back} associated with tail blocks of back edges E_{back}. We use the set S to represent the subset of already reached states that fall inside S_{back}. The procedure terminates whenever the standard fixpoint procedure terminates. The procedure returns `false` if a violation is found, i.e., there exists some $s \in Err$ that is reachable; the procedure returns `true` if it is proven that no violation exists. In the later case, we say a proof is found.

Algorithm 4 REACH_FRONTIER(T,I,Err,S_{back})

```
 1: $F = I$;
 2: $S = I \cap S_{back}$;
 3: while $F \neq \emptyset$ do
 4:     if $(F \cap Err) \neq \emptyset$ then
 5:         return false;
 6:     end if
 7:     $F = (post(T, F) \setminus F) \setminus S$;
 8:     $S = S \cup (F \cap S_{back})$;
 9: end while
10: return true;
```

The symbolic analysis presented in this section is well suited for analyzing individual BPEL processes, for which the CPG sizes are often small. When applied to a composite web service, however, such a monolithic verification method often suffers from the well-known state space explosion problem, similar to the prior methods using automata product construction. In the next section, we present a modular verification method, which analyzes BPEL processes individually to compute their summaries before composing them together to verify the entire system.

5. MODULAR VERIFICATION OF WEB SERVICE COMPOSITION

We summarize both the safe invoking context and the behavior of a process in terms of its incoming and outgoing messages. The safe invoking context (pre-condition) is computed using a backward symbolic analysis starting from the set Err of *error states*, followed by projecting the result to the *incoming messages*. The behavior of a process (post-condition) is computed using a forward symbolic analysis starting from the initial states, followed by projecting the result to the set of *incoming messages* and *outgoing messages*. Assume that in a composite web service, the main process is interacting with some external processes. We shall show that, under certain conditions, by composing the summaries of the external processes, the reachability analysis of the main process itself is as precise as the monolithic analysis of the entire system.

We focus on checking safety properties, e.g., no violation of assertions in the BPEL source code. A process is *safe* if no assertion violation occurs during its execution (including all the embedded service invocations). A process invocation is safe if no assertion violation occurs during the execution of the invoked process.

For the following analysis, we assume that there is a unique pair of send and receive edges for each channel. This constraint implies that each message can only be written by one process once. If a channel is shared by multiple pairs of send and receive edges, we rewrite by creating multiple copies of the channel. (The above constraint may not be satisfied by some types of BPEL processes in practice; we discuss this limitation in Sec 5.3.) Consider a process modeled by a CPG $\langle N, \mathit{Var}, \mathit{Chl}, E\rangle$. For $e \in E$, if $e.act = ch!exp$ or $e.act = ch?v$, we use $e.ch$ to denote ch. With each $ch \in \mathit{Chl}$, we associate a message msg_{ch}. $M_{out} = \{msg_{e.ch} \mid e \in \mathit{Send}\}$ denotes the set of outgoing messages of the process, and $M_{in} = \{msg_{e.ch} \mid e \in \mathit{Receive}\}$ denotes the set of incoming messages of the process.

For a process P, we use *P.Pre* to represent its safe invoking context, which is a constraint over the variables in M_{in}. *P.Pre* is the condition under which invoking P is guaranteed not to cause assertion failure inside P. We use *P.Post* to represent the expected outcome of executing process P. *P.Post* is the precise relation of the variables in M_{in} and the variables in M_{out}. The summary of an individual process P is defined as a tuple $\langle M_{in}, M_{out}, \mathit{Pre}, \mathit{Post}\rangle$.

In the following sections, we shall show how to compute the summary of an individual process. Note that the summary of each process is computed separately, without the consideration of (the summaries of) other processes; that is, when computing the invoker process, we do not need the summaries of the invoked processes. Later, we shall show how to compose the summaries of all the involved processes in analyzing the entire system. Finally, we would like to point out that the assertions may appear in any of the involved processes; it does not affect the order in which processes are summarized. We shall explain that, by conducting a local analysis of the main process using the summaries of the other involved processes, we can detect assertion violations in any of the involved process, and the verification result is as precise as a monolithic analysis of the entire system.

5.1 Computing Process Summaries

Both *P.Pre* and *P.Post* are computed using process-local symbolic fixed point computations. A local transition system $\langle I, T, X\rangle$ for process P is constructed as follows. $X = M_{in} \cup M_{out} \cup X_m$ is the set of state variables, where X_m is the set of state variables that we have defined for a monolithic analysis of P. The transition relation T is computed by Algorithm 3 with slight changes: (1) read incoming messages (replace line 30 with $T_e := T_e \wedge x' = msg_{ch}$), and (2) write outgoing messages (replace line 33 with $T_e := T_e \wedge msg'_{ch} = exp$). Note that a process-local transition system does not inline the invoker/invoked processes or their summaries.

We define *P.Pre* as the conjunction of a set of *safe* constraints over (the incoming messages of) the receive edges in P:

$$\bigwedge_{e \in \mathit{Receive}} \exists M_{out}.\exists X_m.(S_{pre} \wedge e.src.pc = e.src.id),$$

where $S_{pre} \wedge e.src.pc = e.src.id$ is the set of safe states associated with the source node of a receive edge.

Algorithm 5 computes S_{pre} as a greatest fixed point via a backward analysis starting from the set Err of error states in P. S_{pre} is the weakest pre-condition of the set of local assertion conditions. In the pseudo code, we use S_{tmp} to represent the set of states that can reach Err. $S_{rec} = \bigvee_{n \in N}(n.pc = n.id)$, where $N = \{e.src \mid e \in \mathit{Receive}\}$, is the state subspace associated with the set of source nodes of receive edges. $pre(T, S)$ implements the computation of pre-condition of S with respect to the transition relation T. The algorithm iteratively cuts states that may lead to Err until a fixed point is reached.

Algorithm 5 REACH_PRE(T,Err,S_{rec})

```
1: $S_{tmp} := Err$;
2: $S_{pre} := \mathsf{true} \setminus (S_{tmp} \cap S_{rec})$;
3: $S'_{tmp} := pre(T, S_{tmp})$;
4: while $S'_{tmp} \setminus S_{tmp} \neq \emptyset$ do
5:     $S_{pre} := S_{pre} \setminus (S'_{tmp} \cap S_{rec})$;
6:     $S_{tmp} := S'_{tmp}$;
7:     $S'_{tmp} := pre(T, S_{tmp})$;
8: end while
```

We define *P.Post* as the relation of all the outgoing messages in M_{out} and all the incoming messages in M_{in}. Since the incoming messages to process P do not change inside P, we can compute *P.Post* by conjoining the relations between individual outgoing messages and all the incoming messages as follows:

$$\bigwedge_{e \in \mathit{Send}} \exists M_{out} \setminus \{msg_{e.ch}\}.\exists X_m.(S_{post} \wedge e.tgt.pc = e.tgt.id),$$

where $S_{post} \wedge e.tgt.pc = e.tgt.id$ is the set of reachable states associated with the target node of a send edge. Algorithm 6 computes S_{post} as a least fixed point via a forward analysis. S_{post} is the set of states reachable from the intersection set of initial states I and S_{pre}.

In the pseudo code, we use $S_{send} = \bigvee_{n \in N}(n.pc = n.id)$, where $N = \{e.tgt \mid e \in \mathit{Send}\}$, to denote the state subspace associated with the target nodes of send edges. Since S_{pre} depends solely on the incoming messages in M_{in} (which do not change inside P), the result can be computed by conjoining S_{pre} with the reachable states from I at the end (line 9).

5.2 Composing Process Summaries

Now we describe how to utilize process summaries to achieve modular verification of the entire system. Consider a system of multiple interacting processes, each of which may contain some assertion checks; our modular analysis proceeds as follows: (1) we arbitrarily pick a process P, and for the remaining processes, we perform the process-local summarization as described in the previous section; (2) after computing the summaries of all processes except P, we derive a *joint summary*; and (3) when building the process-local transition relation for process P, we impose the *joint summary* as a set of additional constraints (to be described in the

Algorithm 6 REACH_POST(T,I,S_{send}, S_{pre})

```
1: $S_{tmp} := I$;
2: $S_{post} := S_{tmp} \cap S_{send}$;
3: $S'_{tmp} := post(T, S_{tmp})$;
4: while $S'_{tmp} \setminus S_{tmp} \neq \emptyset$ do
5:     $S_{post} := S_{post} \cup (S'_{tmp} \cap S_{send})$;
6:     $S_{tmp} := S'_{tmp}$;
7:     $S'_{tmp} := post(T, S_{tmp})$;
8: end while
9: $S_{post} := S_{post} \cap S_{pre}$;
```

remainder of this section). A joint summary of a set of processes is defined as $\langle M_{IN}, M_{OUT}, PRE, POST\rangle$, where $M_{IN} = \bigcup_P P.M_{in}$, $M_{OUT} = \bigcup_P P.M_{out}$, $PRE = \bigwedge_P P.Pre$, $POST = \bigwedge_P P.Post$.

The transition system $\langle I, T, X\rangle$ of process P, which utilizes the summary $\langle M_{IN}, M_{OUT}, PRE, POST\rangle$, is constructed as follows. $X = M_{IN} \cup M_{OUT} \cup X_m$, where X_m is the set of process-local state variables. T is constructed as in Algorithm 3 but the segment from line 21 to 34 is replaced with the segment from line 1 to 20 in Algorithm 7. When building the transition relation of process P, each process invocation is treated as if it is a single transition step.

Algorithm 7 COMPOSE_SUMMARY_PATCH(G)

```
1: if $\alpha$ is assignment $v := exp$ then
2:     $T_e := T_e \wedge v' = exp$;
3: else if $\alpha$ is receive $ch?x$ then
4:     if $\exists e_3 \in E, e_3.act = ch!exp$ then
5:         $E_{visited} := E_{visited} \cup \{e_3\}$;
6:         $X_{visited} := X_{visited} \cup \{e_3.tgt.pc\}$;
7:         $T_e := T_e \wedge x' = exp$;
8:         $T_e := T_e \wedge e_3.src.pc = e_3.src.id \wedge e_3.tgt.pc' = e_3.tgt.id$;
9:     else if $msg_{ch} \in M_{OUT}$ then
10:        $T_e := T_e \wedge x' = msg_{ch} \wedge POST$;
11:    else
12:        $T_e := T_e \wedge x' = *$;
13:    end if
14: else if $\alpha$ is send $ch!exp$ then
15:    if $msg_{ch} \in M_{IN}$ then
16:        $X_{visited} := X_{visited} \cup msg_{ch}$;
17:        $msg'_{ch} = exp$;
18:        Assert ($e.tgt.pc = e.tgt.id \wedge PRE$);
19:    end if
20: end if
```

At line 10 in Algorithm 7, we impose the constraint (over msg_{ch}) derived from the invoked process. The encoding ensures that the value of x after $ch?x$ is properly defined. At line 18, we add the assertion to ensure that interaction with the external process is safe. If the assertion is not satisfied, it means that an assertion is violated by the external process. For messages that do not correspond to any processes that has been summarized (e.g., due to missing BPEL source code), we adopt a conservative encoding to allow all possible values, i.e., at line 12, the value of x in $ch?x$ becomes nondeterministic.

EXAMPLE 1. *Consider the two processes in Figure 6, where P_A invokes P_B and there is an assertion inside P_A:* `assert`$(y = 1)$ *at n_4. The corresponding CPGs are given in Figure 6. There are two different ways of applying our modular analysis method. For the purpose of checking assertion violation, our analysis remains sound and complete in both cases.*

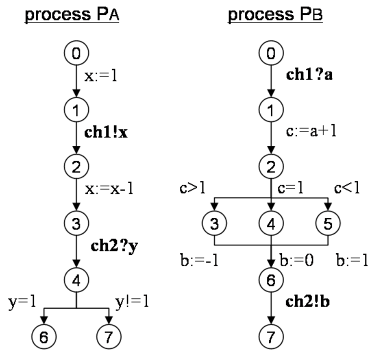


Figure 6: An example for process summary and composition

Case 1: We summarize process P_B first, and then verify process P_A by composing the summary of P_B. The summary of P_B is $\langle\{msg_{ch1}\}, \{msg_{ch2}\}, \texttt{true}, Post\}\rangle$, where $P_B.Post$ is

$$\begin{array}{l}(msg_{ch1} > 0 \wedge msg_{ch2} = +1)\vee \\ (msg_{ch1} = 0 \wedge msg_{ch2} = 0)\vee \\ (msg_{ch1} < 0 \wedge msg_{ch2} = -1)\end{array}$$

Note that $P_B.Post$ is the precise relation between the incoming messages and the outgoing messages. After composing the summary of P_B, the transition relation of process P_A is

$$\begin{array}{lcl}n_0 \rightarrow n_1 & : & (pc = 0 \wedge pc' = 1) \wedge (x' = 1) \\ n_1 \rightarrow n_2 & : & (pc = 1 \wedge pc' = 2) \wedge (msg'_{ch1} = x) \\ n_2 \rightarrow n_3 & : & (pc = 2 \wedge pc' = 3) \wedge (x' = x - 1) \\ & & \wedge(msg'_{ch1} = msg_{ch1}) \\ n_3 \rightarrow n_4 & : & (pc = 3 \wedge pc' = 4) \wedge (y' = msg_{ch2} \wedge P_B.Post) \\ \dots & & \end{array}$$

One can verify that when $pc = 4$, the assertion condition $(y = 1)$ always holds.

Case 2: Alternatively, we can summarize process P_A, and then verify process P_B by composing the summary of P_A. The summary of P_A is $\langle\{msg_{ch2}\}, \{msg_{ch1}\}\ , msg_{ch2} \neq 1, msg_{ch1} = 1\}\rangle$. The transition relation of process P_B is

$$\begin{array}{lcl}n_0 \rightarrow n_1 & : & (pc = 0 \wedge pc' = 1) \wedge (a' = msg_{ch1}) \\ & & \wedge(msg_{ch1} = 1) \\ n_6 \rightarrow n_7 & : & (pc = 6 \wedge pc' = 7) \wedge (msg'_{ch2} = b) \\ \dots & & \end{array}$$

While composing the summary of P_A, we shall add an assertion to P_B at node n_7; that is, `assert`$(msg_{ch2} \neq 1)$ at n_7. One can verify that when $pc_1 = 7$, the condition $(msg_{ch2} \neq 1)$ always holds, meaning that there is no assertion violation inside P_A.

5.3 Modularity of Message-passing Processes

Given a transition system $\langle I, T, X\rangle$, where T is in disjunctive form, we say $x \in X$ is a *symbolic constant* if $x' = x$ holds for all disjunctive transitions in T. When we conduct a process-local analysis of P, the incoming messages to P are symbolic constants. (P may change its local copies, but not the incoming messages themselves.) Let us prove that, in Algorithm 6, we can compute the set of states reachable from $I \wedge S_{pre}$ precisely, by conjoining S_{pre} at the end of the reachability analysis (line 9).

Let $C = \{x \mid x \in X, x \text{ is a symbolic constant}\}$ and $I(X)$ denote a constraint over variables in X. By definition, $T(X, X')$ is equal to $T'(C, Y, Y') \wedge C' = C$, where $Y = X \setminus C$ and $T' = \exists C'.T$.

We have the following property:

$$post^*(T, I(C) \wedge I(X)) \equiv I(C) \wedge post^*(T, I(X)).$$

This property states that, as long as variables in C are symbolic constants, one can precisely compute the fixed point of reachable states form $I(C) \wedge I(X)$ by simply conjoining $I(C)$ with the fixed point of reachable states from $I(X)$. Note that for a BPEL process, the reachable states from $I(X)$ can be calculated in advance, and serve as a summary.

We prove the property as follows:

$$\begin{array}{ll}
 & post(T, I(C) \wedge I(X)) \\
\equiv & (\exists X.I(C) \wedge I(X) \wedge T(X, X'))[X/X'] \\
\equiv & (\exists Y.\exists C.I(C) \wedge I(C,Y) \wedge T(C,Y,C',Y'))[C/C'][Y/Y'] \\
\equiv & (\exists Y.((\exists C.(I(C) \wedge I(C,Y) \wedge T'(C,Y,Y') \wedge C' = C)) \\
 & [C/C']))[Y/Y'] \\
\equiv & (\exists Y.((I(C') \wedge I(C',Y) \wedge T'(C',Y,Y'))[C/C']))[Y/Y'] \\
\equiv & (\exists Y.(I(C) \wedge I(C,Y) \wedge T'(C,Y,Y')))[Y/Y'] \\
\equiv & I(C) \wedge ((\exists Y.I(C,Y) \wedge T'(C,Y,Y'))[Y/Y']) \\
\equiv & I(C) \wedge ((\exists Y.((\exists C.I(C,Y) \wedge T'(C,Y,Y') \wedge C' = C) \\
 & [C/C']))[Y/Y']) \\
\equiv & I(C) \wedge (\exists Y.\exists C.I(C,Y) \wedge T(C,Y,C',Y'))[C/C'][Y/Y'] \\
\equiv & I(C) \wedge (\exists X.I(X) \wedge T(X,X'))[X/X'] \\
\equiv & I(C) \wedge post(T, I(X))
\end{array}$$

By induction on the iterations of $post$ function, we have:

$$post^*(T, I(C) \wedge I(X)) \equiv I(C) \wedge post^*(T, I(X)).$$

Consider that a process P_A sends a message msg_1 to process P_B, and P_A receives a message msg_2 as the reply. To analyze P_A, one needs to know the values of msg_2 w.r.t. msg_1. This is obtained by computing $post^*(T_B, I(msg_1) \wedge I_B)$, where T_B is the process-local transition relation and I_B is the initial states of P_B. Since msg_1 is never modified by P_B, according to the property described above, one can compute $I(msg_1) \wedge post^*(T_B, I_B)$ instead of computing $post^*(T_B, I(msg_1) \wedge I_B)$. In this case, $post^*(T_B, I_B)$ can be computed separately without any information about P_A. During the analysis of P_A, we simply conjoin $I(msg_1)$ with the summary of P_B. This is done implicitly in Algorithm 7 (line 10).

Since we assume that there is a unique send and receive pair for each channel, the value of each message is preserved after it is written. As long as the fixed point of reachable states is computed precisely in Algorithm 5 and 6 (no widening is involved), *PRE* is the most permissive environment assumption of summarized processes and *POST* is the precise summary of the relation between M_{IN} and M_{OUT}.

Finally, for BPEL processes having multiple sends and receives on the same channel, we have to explicitly enumerate the possible ways of pairing up one send and one receive and assign each pair a unique channel. When a naive algorithm is used, in the worst case, we will end up creating too many possible CPGs. This is a limitation of our CPG based symbolic analysis. Furthermore, if a send/receive activity is within a loop, we need to unwind the loop fully before applying the CPG based analysis.

In this paper, we perform composite model checking [3, 29, 30] on systems having unbounded integers. In the integer domain, the precise fixed point of reachable states is not always computable. Although we can compute the precise reachable states in most of the practical examples, there are cases where the fixed point computation may not converge. In such cases, we may apply approximation techniques such as widening [7] to enforce convergence and get a conservative summary.

6. EXPERIMENTS

We developed a prototype tool that implements the techniques proposed in this paper. The tool consists of three major components: (1) Translator for BPEL+WSDL to CPG, (2) Composite Symbolic Model Checker, (3) Summarizer/Modular Verifier.

We first parse BPEL and WSDL documents and extract necessary control and data flow information. Then we construct the corresponding concurrent process graph (CPG). Depending on type of verification used (*monolithic* or *modular* verification) we build the symbolic transition relations for the processes. After constructing the symbolic encoding, we employ a symbolic model checker for reachability analysis. We use a composite symbolic representation with an underlying symbolic library [3] that is able to handle models with both boolean and unbounded integer variables. The composite symbolic representation is used for symbolic model checking with state sets represented over Binary Decision Diagrams and Polyhedra over linear constraints [30, 29].

In monolithic model checking, all BPEL processes are put into a single CPG, and then the symbolic reachability analysis is applied to the resulting symbolic transition system. In modular verification, we first compute summaries of the processes. While analyzing a process, we compose its transition system with the summaries of the other processes and then use symbolic reachability analysis.

We conducted experiments on two public benchmarks: loan approval and travel agency. The results are shown in Table 2 and Table 3, respectively. The second column shows the result of monolithic analysis. Columns 3-6 shows the results of analyzing each individual process in modular analysis. In the result row, *P* indicates that a proof is found and assertions are not violated, while *NA* indicates that the analysis is not able to terminate. *S* indicates that a summary is generated. The CPU time is given in seconds (s). The memory usage is given in mega bytes (MB). The ITRs row shows the number of iterations of the fixpoint computation.

In the monolithic analysis of *loan approval*, the reachable states of all concurrent processes converged at the 32-th iteration of the fixpoint computation, and a proof that none of the assertions are violated was found. In the modular analysis, we summarize assessor, approver, and customer processes before analyzing the root process, approval. Although there is some overhead in summarizing the first three processes, by applying these summaries to the root process, we were able to reduce the runtime by 90% (from 1227.2 seconds to 124.5 seconds) and reduce the memory usage by 40% (from 810 MB to 490 MB).

In the monolithic analysis of *travel agency*, the computation ran out of memory at the 57-th fixpoint iteration. Neither a proof nor a violation was found. In the modular analysis, we successfully proved the correctness of root process (VTA) by first summarizing processes hotel, flight, and user. Compared to the monolithic analysis, modular verification of VTA took only fraction of the effort required for the (incomplete) monolithic analysis (99.96% reduction in time and 78% reduction in memory).

Our experimental results on these two BPEL specifications show that our modular verification method outperforms the conventional monolithic verification technique both in terms of execution time and memory usage, resulting in a more scalable verification approach.

7. CONCLUSION

We have proposed a modular verification method for composite web services written in BPEL. Our approach is based on an efficient symbolic encoding to model the interleaving semantics of BPEL processes, which includes both shared variable multi-threading within a process and synchronous/asynchronous message

	Monolithic Verification	Modular Verification			
		Approval	Assessor	Approver	Customer
Result	P	P	S	S	S
Time (s)	1227.2	124.5	0.1	0.1	0.1
Memory (MB)	810	490	289	290	290
ITRs	32	16	10	10	5

Table 2: Analyzing the loan approval benchmark.

	Monolithic Verification	Modular Verification			
		VTA	Hotel	Flight	User
Result	NA	P	S	S	S
Time (s)	18947	814	13.5	13.4	34.6
Memory (MB)	1663	363	273	363	284
ITRs	57	55	23	22	30

Table 3: Analyzing the travel agency benchmark.

passing among processes. We present a novel method for analyzing and summarizing concurrently running processes, together with a modular verification framework that utilizes the summaries for scalable verification of the entire system.

8. REFERENCES

[1] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In *Computer Aided Verification*, pages 521–525. Springer, 1998. LNCS 1427.

[2] D. Beyer, A. Chakrabarti, and T. A. Henzinger. Web service interfaces. In *International Conference on World Wide Web*, pages 148–159. ACM, 2005.

[3] T. Bultan, R. Gerber, and C. League. Composite model-checking: verification with type-specific symbolic representations. *ACM Trans. Softw. Eng. Methodol.*, 9(1):3–50, 2000.

[4] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design*, 2004.

[5] R. Cleaveland and S. Sims. The ncsu concurrency workbench. In *Computer Aided Verification*, pages 394–397. Springer, 1996. LNCS 1102.

[6] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346. Springer, 2003. LNCS 2619.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977.

[8] Z. Duan, A. J. Bernstein, P. M. Lewis, and S. Lu. A model for abstract process specification, verification and composition. In *International Conference on Service-Oriented Computing*, pages 232–241. ACM, 2004.

[9] Z. Duan, A. J. Bernstein, P. M. Lewis, and S. Lu. Semantics based verification and synthesis of BPEL4WS abstract processes. In *International Conference on Web Services*, pages 734–737. IEEE Computer Society, 2004.

[10] J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *Principles of Programming Languages*, pages 1–11, 2000.

[11] J. Klein F. Leymann D. Roller F. Curbera, Y. Goland and S. Weerawarana. Business process execution language for web services, version 1.1. 2003.

[12] R. Farahbod, U. Glässer, and M. Vajihollahi. An abstract machine architecture for web service based business process management. In *Business Process Management*, pages 144–157. Springer, 2005. LNCS 3812.

[13] C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3):153–183, 2005.

[14] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Compatibility verification for web service choreography. In *International Conference on Web Services*, pages 738–741, 2004.

[15] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *International Conference on World Wide Web*, pages 621–630. ACM, 2004.

[16] X. Fu, T. Bultan, and J. Su. Model checking XML manipulating software. In *International Symposium on Software Testing and Analysis*, pages 252–262, 2004.

[17] S. Haddad, T. Melliti, P. Moreaux, and S. Rampacek. Modelling web services interoperability. In *International Conference on Enterprise Information Systems*, pages 287–295, 2004.

[18] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[19] G. J. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[20] K. Huynh. Analysis through reflction: walking the EMF model of BPEL4WS. Master's thesis, York University, Toronto, Canada, 2005.

[21] N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg. Analyzing interacting bpel processes. In *Business Process Management*, pages 17–32. Springer, 2006. LNCS 4102.

[22] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.

[23] S. Nakajima. Model-checking behavioral specification of BPEL applications. *Electr. Notes Theor. Comput. Sci.*, 151(2):89–105, 2006.

[24] S. Qadeer, S. K. Rajamani, and J. Rehof. Summarizing procedures in concurrent programs. In *Principles of Programming Languages*, pages 245–255. ACM, 2004.

[25] Z. Qiu, S. Wang, G. Pu, and X. Zhao. Semantics of BPEL4WS-like fault and compensation handling. In *International Symposium of Formal Methods*, pages 350–365. Springer, 2005. LNCS 3582.

[26] R. K. Ranjan, A. Aziz, R. K. Brayton, B. F. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification. May 1995.

[27] N. Sharygina K. Wallnau S. Chaki, J. Ivers. The comfort reasoning framework. In *Computer Aided Verification*, pages 164–169, 2005.

[28] N. Sharygina, S. Chaki, E. M. Clarke, and N. Sinha. Dynamic component substitutability analysis. In *International Symposium of Formal Methods Europe*, pages 512–528. Springer, 2005. LNCS 3582.

[29] Z. Yang, C. Wang, F. Ivančić, and A. Gupta. Mixed symbolic representations for model checking software programs. In *Formal Methods and Models for Codesign*, pages 17–24, July 2006.

[30] T. Yavuz-Kahveci and T. Bultan. A symbolic manipulator for automated verification of reactive systems with heterogeneous data types. *STTT*, 5(1):15–33, 2003.