Communication Papers of the Federated Conference on Computer Science and Information Systems pp. 241–247
DOI: 10.15439/2018F148
ISSN 2300-5963 ACSIS, Vol. 17

Assertional Reasoning for Concurrent and Communicating BPEL-like Programs

Longfei Zhu
Key Laboratory of Ministry of Public Security, Zhejiang Police College, Zhejiang, China
Email: zhulongfei@zjjcxy.cn

Qiwen Xu §
Faculty of Science and Technology, University of Macau, Macao SAR, China
Email: qwxu@umac.mo

Huibiao Zhu
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China
Email: hbzhu@sei.ecnu.edu.cn

***Abstract*—This paper studies verification of programs similar to BPEL4WS (BPEL), the latter being a de facto standard for the web services composition and orchestration. Traditionally, in verification of concurrent and distributed programs, the model was either based on shared variables or message passing and was studied separately. BPEL-like programs have features that are present in both models: several flows within one service can be executed in parallel and they can access shared variables, whereas several services communicate by message passing. Therefore, it is natural that for verification of BPEL-like programs, the verification methods developed for shared variables and message passing be integrated. In this paper, we combine the proof rules for shared variable programs from Owicki and Gries, the proof rules for CSP like programs from Apt, Francez and de Roever, together with proof rules for compensation and fault handling, to cover all major features of BPEL. An operational semantics is presented and the proof rules can be justified over that. Examples are provided to show the feasibility of verification framework.**

***Index Terms*—BPEL, Hoare logic, shared variables, message passing.**

I. Introduction

WEB services and other web-based applications have been becoming more and more important in practice. Various web-based business process languages have been introduced, such as XLANG [1], WSFL [2], BPEL4WS (BPEL) [3] and StAC [4], which are designed for the description of services composed of a set of processes across the Internet. Their goal is to achieve the universal interoperability between applications by using web standards, as well as to specify the technical infrastructure for carrying out business transactions. BPEL4WS (BPEL) is the OASIS standard for web services composition and orchestration. To support long-running transactions, it provides the ability to define fault and compensation handling. In addition, BPEL allows several flows executing in parallel in a service, and several services running concurrently. Due to the interesting features of BPEL programs mentioned above, the verification of BPEL programs is challenging.

Much research has been done on verification of concurrent and distributed programs. Typically, the model is either based on shared-variables or message passing. Owicki and Gries [5], and Apt, Francez and de Roever [6], respectively extended Hoare logic to concurrent programs with shared variables and distributed programs with message passing. BPEL-like programs have features that are present in both models: several parallel flows within one service can access shared variables, whereas several concurrent services communicate by message passing. Therefore, it is natural that for verification of BPEL like programs, the verification methods developed for shared variables and message passing be integrated. In this paper, we combine the proof rules for shared variable programs from Owicki and Gries, the proof rules for CSP like programs from Apt, Francez and de Roever, together with proof rules for compensation and fault handling, to cover all major features of BPEL.

The remainder of this paper is organized as follows. Section 2 introduces a language based on BPEL together with an operational semantics. In section 3, we provide the verification rules, including the rules for dealing with compensation, fault handling, parallel flows through shared variables, and multiple services through message passing. A few simple examples are given to illustrate the rules. Section 4 concludes the paper with a discussion.

II. An Operational Model

In this section, we present the operational semantics of a BPEL-like language, based on the work in [7] and [8].

A. The Syntax of BPEL

Our language contains the following categories of syntactic elements:

$$\begin{array}{lll} BA & ::= & \texttt{skip} \mid x := e \mid \texttt{rec}\ a\ x \mid \texttt{rep}\ a\ e \mid \texttt{throw} \\ A & ::= & BA \mid g \circ A \mid A;\ A \mid A \lhd b \rhd A \mid b * A \\ & & \mid A \parallel A \mid A \sqcap A \mid \texttt{undo}\ n \mid \{A\,?\,A,\ A\}_n \\ W & ::= & (A, \cdots, A) \end{array}$$

where:

- The category BA stands for the basic activity. Activity $x := e$ assigns the value of e to variable x. Activity `skip` behaves the same as $x := x$. A variable may be shared among parallel flows within one service.
  In order to implement the communications between concurrent services, two statements are introduced, i.e.,

§ Qiwen Xu is corresponding author.

`rec` a x and `rep` a e. Activity `rec` a x represents the receiving of a value through channel a and storing in x. To avoid complications, we assume variable x is not shared by parallel flows. If the information is needed by another flow, it has to be copied to another variable first. Sending a message is represented by `rep` a e. Activity `throw` indicates that the program encounters a fault.

- The category A stands for the activities within one service. Several constructs are similar to those in traditional programming languages. $A; B$ stands for sequential composition. $A \lhd b \rhd B$ is the conditional construct and $b * A$ is the iteration construct. $A \sqcap B$ stands for the nondeterministic choice. $g \circ A$ awaits the Boolean condition g to be set true. $\{A\,?\,C,\, F\}_n$ stands for the scope based compensation statement, where n stands for the scope name, A, C and F for the primary activity, compensation program and fault handler correspondingly. If A terminates successfully, program C is installed in the compensation list for later compensating. On the other hand, if A encounters a fault during its execution, the fault handler F will be activated. Further, the compensation part C does not contain scope activity. Statement "`undo` n" activates the execution of the programs with scope name n.
  A service may contain one or several flows running in parallel. We use the notation $A \parallel B$ to stand for two such flows.
- The category W stands for the coordination of several concurrent services. Such a set of services is denoted by $(A_1, \cdots, A_n)$, and their communication is modelled by message passing.

B. An Operational Model

For the operational semantics of BPEL, its transitions are of the two types.

$$C \longrightarrow C' \quad \text{or} \quad C \xrightarrow{a.m} C'$$

where C and C' are the configurations describing the states of an execution mechanism before and after a step respectively. The first type is used to denote non-communication transitions. The second type is used to represent communication between concurrent services where a is the channel and m is the message that is passed.

A configuration is expressed as $\langle P, \sigma, Cp\rangle$, where

(1) The first component P is a program that remains to be executed.
(2) The second element σ is the state for all the variables.
(3) The third element Cp stands for a compensation set; i.e., containing the scope names whose compensation parts need to be executed. Cp can contain several copies of the same element. Therefore, it is in fact a bag. For a scope n, the compensation program is denoted by $C(n)$. When statement `undo n` is executed, $C(n)$ will be invoked.

For the program P in configuration $\langle P, \sigma, Cp\rangle$, it can either be a normal program or one of the following special forms:

ε : A program has terminated successfully. We use ε to represent the empty program.
$\boxtimes$: A program has encountered a fault and stops at the faulty state, represented by a special symbol $\boxtimes$.

C. Transition Rules

Transition rules are presented below.

(1) Basic Commands

Firstly we list the operational semantics for basic commands. The execution of $x := e$ assigns the value of expression e to variable x, and leaves other variables unchanged.

$$\langle x := e,\ \sigma,\ Cp\rangle \longrightarrow \langle \varepsilon,\ \sigma[x \mapsto e(\sigma)],\ Cp\rangle$$

For communication commands, statement `rec a x` receives message m through channel a. The received message will be stored in variable x.

$$\langle \texttt{rec a x},\ \sigma,\ Cp\rangle \xrightarrow{a.m} \langle \varepsilon,\ \sigma[x \mapsto m],\ Cp\rangle$$

`rep a e` stands for the sending of `e` on channel a, and the message is $e(\sigma)$ when sent in state σ.

$$\langle \texttt{rep a e},\ \sigma,\ Cp\rangle \xrightarrow{a.e(\sigma)} \langle \varepsilon,\ \sigma,\ Cp\rangle$$

`throw` encounters a fault after activation, while leaving all variables and the compensation set unchanged.

$$\langle \texttt{throw},\ \sigma,\ Cp\rangle \longrightarrow \langle \boxtimes,\ \sigma,\ Cp\rangle$$

`undo` n invokes the compensation program corresponding to scope name n.

$$\langle \texttt{undo}\ n,\ \sigma,\ Cp\rangle \longrightarrow \langle C(n),\ \sigma,\ Cp \setminus n\rangle, \quad \text{where } n \in Cp$$

Here function $C(n)$ represents the program whose name is n (i.e, the scope name). $Cp \setminus n$ represents that scope name n is removed once from Cp.

(2) Sequential Constructs

For sequential composition $P; Q$, if P does not encounter a fault, the transition rules are the same as usual. Below in this section, $\xrightarrow{\beta}$ denotes either a communication or non-communication transition.

$$\frac{\langle P,\ \sigma,\ Cp\rangle \xrightarrow{\beta} \langle P',\ \sigma',\ Cp'\rangle \ \textit{and}\ P' \neq \varepsilon, \boxtimes}{\langle P; Q,\ \sigma,\ Cp\rangle \xrightarrow{\beta} \langle P'; Q,\ \sigma',\ Cp'\rangle}$$

$$\frac{\langle P,\ \sigma,\ Cp\rangle \xrightarrow{\beta} \langle \varepsilon,\ \sigma',\ Cp'\rangle}{\langle P; Q,\ \sigma,\ Cp\rangle \xrightarrow{\beta} \langle Q,\ \sigma',\ Cp'\rangle}$$

If P encounters a fault during its execution, $P; Q$ also encounters a fault during its execution.

$$\frac{\langle P,\ \sigma,\ Cp\rangle \xrightarrow{\beta} \langle \boxtimes,\ \sigma',\ Cp'\rangle}{\langle P; Q,\ \sigma,\ Cp\rangle \xrightarrow{\beta} \langle \boxtimes,\ \sigma',\ Cp'\rangle}$$

The usual await statement $g \circ P$ waits for the Boolean guard g to be set true.

$$\langle g \circ P,\ \sigma,\ Cp\rangle \longrightarrow \langle P,\ \sigma,\ Cp\rangle, \text{if} \quad g(\sigma)$$

$P \sqcap Q$ either behaves like P or like Q. The choice between

them is nondeterministic.

$$\langle P \sqcap Q,\ \sigma,\ Cp\rangle \longrightarrow \langle P,\ \sigma,\ Cp\rangle$$
$$\langle P \sqcap Q,\ \sigma,\ Cp\rangle \longrightarrow \langle Q,\ \sigma,\ Cp\rangle$$

The conditional $P \lhd b \rhd Q$ starts process P if the value of b is true. Otherwise it executes Q instead.

$$\langle P \lhd b \rhd Q,\ \sigma,\ Cp\rangle \longrightarrow \langle P,\ \sigma,\ Cp\rangle, \text{if} \quad b(\sigma)$$
$$\langle P \lhd b \rhd Q,\ \sigma,\ Cp\rangle \longrightarrow \langle Q,\ \sigma,\ Cp\rangle, \text{if} \quad \neg b(\sigma)$$

The transition rules for iteration are similar to conditional.

$$\langle b * P,\ \sigma,\ Cp\rangle \longrightarrow \langle P; b * P,\ \sigma,\ Cp\rangle, \text{if} \quad b(\sigma)$$

$$\langle b * P,\ \sigma,\ Cp\rangle \longrightarrow \langle \varepsilon,\ \sigma,\ Cp\rangle, \text{if} \quad \neg b(\sigma)$$

(3) Parallel Flows

Now we consider the transition rules for parallel composition. First we define a function $\mathbf{par}(P, Q)$, which can be used in defining the transition rules for parallel composition. Let

$$\mathbf{par}(P,Q) =_{df} \begin{cases} \varepsilon & \text{if } P = \varepsilon \wedge Q = \varepsilon \\ \boxtimes & \text{if } P = \boxtimes \wedge Q = \boxtimes \\ & \vee P = \boxtimes \wedge Q = \varepsilon \\ & \vee P = \varepsilon \wedge Q = \boxtimes \\ P \parallel Q & \text{otherwise} \end{cases}$$

It indicates the program status for two parallel flows after executing a transition. If both components are in the empty states, the whole service is also in the empty state. If both are in the faulty states, or one is in the faulty state and another one is in the empty state, then the whole service is also in faulty state. If one flow performs a transition, the whole service can also perform the transition.

$$\frac{\langle P, \sigma, Cp\rangle \xrightarrow{\beta} \langle P', \sigma', Cp'\rangle}{\langle P \parallel Q, \sigma, Cp\rangle \xrightarrow{\beta} \langle \mathbf{par}(P', Q), \sigma', Cp'\rangle}$$

$$\frac{\langle Q, \sigma, Cp\rangle \xrightarrow{\beta} \langle Q', \sigma', Cp'\rangle}{\langle P \parallel Q, \sigma, Cp\rangle \xrightarrow{\beta} \langle \mathbf{par}(P, Q'), \sigma', Cp'\rangle}$$

(4) Scope

For scope $\{A?\, C,\ F\}_n$, if the primary activity A performs a transition which does not lead to the faulty state, the whole scope can also perform the successful transition of the same type.

$$\frac{\langle A,\ \sigma,\ Cp\rangle \xrightarrow{\beta} \langle A',\ \sigma',\ Cp'\rangle \text{ and } A' \neq \boxtimes}{\langle \{A?\, C,\ F\}_n, \sigma, Cp\rangle \xrightarrow{\beta} \langle \{A'?\, C,\ F\}_n, \sigma', Cp'\rangle}$$

When the primary activity has been terminated, the compensation program is added into the compensation set. This is represented by the following rule.

$$\langle \{\varepsilon?\, C,\ F\}_n,\ \sigma,\ Cp\rangle \longrightarrow \langle \varepsilon,\ \sigma,\ Cp \cup \{n \to C\}\ \rangle$$

On the other hand, if the primary activity performs a transition leading to the faulty state, the fault handler in the scope will be activated.

$$\frac{\langle A,\ \sigma,\ Cp\rangle \xrightarrow{\beta} \langle \boxtimes,\ \sigma',\ Cp'\rangle}{\langle \{A?\, C,\ F\}_n, \sigma, Cp\rangle \xrightarrow{\beta} \langle F, \sigma', Cp'\rangle}$$

(5) Communicating Services

A collection of concurrent services is represented as $W = (P_1,\ P_2,\ \cdots,\ P_n)$, and we use σ_i and Cp_i to denote the state and compensation set of service P_i respectively.

If one service does non-communication transitions, the whole system can also do a transition of the same type.

$$\frac{\langle P_i,\ \sigma_i,\ Cp_i\rangle \longrightarrow \langle P_i',\ \sigma_i',\ Cp_i'\rangle}{\langle W,\ \sigma,\ Cp\rangle \longrightarrow \langle W',\ \sigma',\ Cp'\rangle}$$

where $W' = (P_1, P_2, \cdots, P_i', \cdots, P_n)$, $\sigma = (\sigma_1, \sigma_2, \cdots, \sigma_i, \cdots, \sigma_n)$, $\sigma' = (\sigma_1, \sigma_2, \cdots, \sigma_i', \cdots, \sigma_n)$ and $Cp' = (Cp_1, Cp_2, \cdots, Cp_i' \cdots, Cp_n)$.

If two services involve the communication via the same channel, the whole system also does the communication via the channel.

$$\frac{\langle P_i,\ \sigma_i,\ Cp_i\rangle \xrightarrow{a.m} \langle P_i',\ \sigma_i',\ Cp_i'\rangle, \langle P_j,\ \sigma_j,\ Cp_j\rangle \xrightarrow{a.m} \langle P_j',\ \sigma_j',\ Cp_j'\rangle}{\langle W,\ \sigma,\ Cp\rangle \longrightarrow \langle W',\ \sigma',\ Cp'\rangle}$$

where $W' = (P_1, \cdots, P_i', \cdots, P_j', \cdots, P_n)$, $\sigma' = (\sigma_1, \cdots, \sigma_i', \cdots, \sigma_j', \cdots, \sigma_n)$, $Cp' = (Cp_1, \cdots, Cp_i', \cdots, Cp_j', \cdots, Cp_n)$.

III. Verification Rules

In this section, we study the verification rules for the BPEL-like programs.

A. Correctness Formula

The verification rules are in the form of a Hoare triple:

$$\{p\}\, S\, \{q\}$$

here S stands for the program, p and q stand for the precondition and the postcondition respectively. If the program S is started in a state that satisfies p, after the execution, postcondition q should be satisfied.

To deal with the two typical features of BPEL, i.e., fault handling and compensation, we introduce two variables ok and $comp$.

B. General Rules

Boolean variable ok is used to identify whether a program is in the faulty state or not. For a configuration $\langle P, \sigma, Cp\rangle$, ok is true if and only if $P \neq \boxtimes$. Since the initial configuration is never faulty, we have the following general rule

OK-rule

$$\frac{\{p \wedge ok\}\, S\, \{q\}}{\{p\}\, S\, \{q\}}$$

ok may be false in the postcondition, indicating that the current system has encountered faults in the execution.

The other general rule is the usual consequence rule:
Consequence-rule

$$\frac{p \Rightarrow p_1,\ \ \{p_1\}\, S\, \{q_1\},\ \ q_1 \Rightarrow q}{\{p\}\, S\, \{q\}}$$

C. Rules for Basic Commands

(1) Assignment:
The rule for assignment is the same as in the traditional Hoare logic and ok is true in the postcondition.

$$\{p[e/x]\}\, x := e\, \{p \wedge ok\}$$

(2) throw:
For `throw`, it immediately enters into the faulty state while leaving the states unchanged.

$$\frac{r \text{ does not contain variable } ok}{\{r\}\, \texttt{throw}\, \{\neg ok \wedge r\}}$$

To verify communicating processes, Apt, Francez and de Roever [6] suggested the verification be divided into two phases. The first phase is the "local verification" for each process, and the second phase is the "cooperation test" where the local verification of the processes are checked to be matching.

(3) Replying:
Obviously, sending a message does not change the state

$$\{p\}\, \texttt{rep}\ a\ y\, \{p\}$$

ok actually holds in the postcondition, but we can deduce this fact by applying the OK-rule.

(4) Receiving:

$$\frac{q \Rightarrow ok}{\{p\}\, \texttt{rec}\ a\ x\, \{q\}}$$

This rule at first would look odd, as the postcondition can be anything (in our context, as long as ok is true). Whether the postcondition is really valid is checked in the cooperation test.

The rule for conditional choice is the same as the traditional one.

(5) Conditional choice:

$$\frac{\{p \wedge b\}\, S_1\, \{q\},\ \{p \wedge \neg b\}\, S_2\, \{q\}}{\{p\}\, S_1 \lhd b \rhd S_2\, \{q\}}$$

(6) Sequential Composition

For sequential composition, there are two rules. The first rule stands for the case that the first program successfully terminates. The second rule indicates that the first program encounters fault during its execution.

Rule 1:

$$\frac{r \Rightarrow ok,\ \{p\}\, A\, \{r\},\ \ \{r\}\, B\, \{q\}}{\{p\}\, A\, ;\, B\, \{q\}}$$

Rule 2:

$$\frac{r \Rightarrow \neg ok,\ \{p\}\, A\, \{r\}}{\{p\}\, A\, ;\, B\, \{r\}}$$

(7) Iteration

For simplicity, we only present the rules for partial correctness.

Rule 1:

$$\frac{\{p \wedge b\}\, S\, \{p\}}{\{p\}\mathbf{while}\ b\ \mathbf{do}\ S\{p \wedge \neg b\}}$$

Rule 2:

$$\frac{q \Rightarrow \neg ok,\ \{p \wedge b\}\, S\, \{q\}}{\{p\}\mathbf{while}\ b\ \mathbf{do}\ S\{q\}}$$

D. Scope and Compensation

A compensation may be installed several times, so we introduce a function $comp$ to record that. More specifically, for a scope n, we use $comp.n$ to stand for the number that the compensation program has been installed. For the compensated program named n, we use function $C(n)$ to represent it.

For scope, the verification rules are divided into two cases.

(1) Scope

The first rule deals with the case that the primarily activity A can successfully terminate. The compensation program C is installed.

Rule 1:

$$\frac{\{p\}\, A\, \{q[comp.n + 1/comp.n]\},\ \ q \Rightarrow ok}{\{p\}\, \{A?C, F\}_n\, \{q\}}$$

The second rule handles the case that A encounters the fault. The fault handler will be triggered.

Rule 2:

$$\frac{\{p\}\, A\, \{r \wedge \neg ok\},\ \ \{r\}\, F\, \{q\}}{\{p\}\, \{A?C, F\}_n\, \{q\}}$$

(2) Compensation

For `undo` n, the compensation program $C(n)$ will be executed. In addition, it has the effect of reducing $comp.n$ by 1. Therefore, in the precondition of $C(n)$, the number of the recorded program named n should be one less.

$$\frac{\{p[comp.n + 1/comp.n]\}\, C(n)\, \{q\}}{\{p\}\, \texttt{undo}\ n\, \{q\}}$$

Example 1 Consider the program below.

$\{x := x + 1?x := x - 1, \texttt{skip}\}_n$;
$\{x := x + 2?x := x - 2, \texttt{skip}\}_m$;
`undo` m;
`undo` n

By applying the verification rules, we can obtain the following proof outline:

$\{ok \wedge x = 0 \wedge comp.n = 0 \wedge comp.m = 0\}$
$\{x := x + 1?x := x - 1, \texttt{skip}\}_n$;
$\{ok \wedge x = 1 \wedge comp.n = 1 \wedge comp.m = 0\}$
$\{x := x + 2?x := x - 2, \texttt{skip}\}_m$;

$$\{ok \wedge x = 3 \wedge comp.n = 1 \wedge comp.m = 1\}$$
$$\texttt{undo}\ m\ ;$$
$$\{ok \wedge x = 1 \wedge comp.n = 1 \wedge comp.m = 0\}$$
$$\texttt{undo}\ n$$
$$\{ok \wedge x = 0 \wedge comp.n = 0 \wedge comp.m = 0\}$$

in which the verification of undo m is supported by

$$\{ok \wedge x = 3 \wedge comp.n = 1 \wedge comp.m + 1 = 1\}$$
$$x := x - 2$$
$$\{ok \wedge x = 1 \wedge comp.n = 1 \wedge comp.m = 0\}$$

and of undo n by

$$\{ok \wedge x = 1 \wedge comp.n + 1 = 1 \wedge comp.m = 0\}$$
$$x := x - 1$$
$$\{ok \wedge x = 0 \wedge comp.n = 0 \wedge comp.m = 0\}$$

In this example, the compensation programs completely undo the effect of the forward activities, so it should be expected that final postcondition is exactly the same as the initial precondition.

E. Parallel Flows

In one service, several flows may be executed in parallel and information is exchanged via shared variables. In the classic verification method due to Owicki and Gries [5], the central concept is the interference freedom. Intuitively, it means that assertions in the local proofs of one process should not be invalidated by the execution of a parallel process. Suppose $\{p\}\,S\,\{q\}$ is a Hoare triple in the local verification for the statement S, statement T from another process is said to be interference free to $\{p\}\,S\,\{q\}$ if the following two conditions are satisfied:

(1) $\{\exists ok.p \wedge pre(T)\}\,T\,\{\exists ok.p\}$
(2) $\{\exists ok.q \wedge pre(T)\}\,T\,\{\exists ok.q\}$

where $pre(T)$ is the precondition of T. Note the interference freedom is concerned with the shared program variables, and hence ok is removed from the assertions by the quantification. Adopting the parallel rule to our setting, the postcondition is modified to take into account the faulty states.

$$\frac{\{p_i\}\,S_i\,\{q_i\}\ \text{are interference-free}}{\{p_1 \wedge p_2\}\,S_1 \parallel S_2\,\{Merge(q_1, q_2)\}}$$

where $Merge(q_1, q_2) =_{df} \exists ok_1, ok_2 \bullet q_1[ok_1/ok] \wedge q_2[ok_2/ok] \wedge ok = ok_1 \wedge ok_2$. This combines the two postconditions, for the information about local variables and compensation, and the parallel flow is in the faulty state if at least one component is in the faulty state.

Example 2 Let $S_1 =_{df} x := x + 1\ ;\ \texttt{throw}$, $S_2 =_{df} x := x + 2$.

For S_1, we have the following local proof outline

$$\{x = 0\}$$
$$\{ok \wedge (x = 0 \vee x = 2)\}$$
$$x := x + 1$$
$$\{ok \wedge (x = 1 \vee x = 3)\}$$
$$\texttt{throw}$$
$$\{\neg ok \wedge (x = 1 \vee x = 3)\}$$

For S_2, the proof outline is

$$\{x = 0\}$$
$$\{ok \wedge (x = 0 \vee x = 1)\}$$
$$x := x + 2$$
$$\{ok \wedge (x = 2 \vee x = 3)\}$$

For interference freedom test, we need to check assertions $(x = 0 \vee x = 2)$ and $(x = 1 \vee x = 3)$ in S_1 are not invalidated by $x := x + 2$ in S_2, whereas $(x = 0 \vee x = 1)$ and $(x = 2 \vee x = 3)$ in S_2 are not invalidated by $x := x + 1$ in S_1. Formally, this is shown by the following

$$\begin{array}{llll}
\{(x = 0 \vee x = 2) \wedge (x = 0 \vee x = 1)\} & x := x + 2 & \{x = 0 \vee x = 2\} \\
\{(x = 1 \vee x = 3) \wedge (x = 0 \vee x = 1)\} & x := x + 2 & \{x = 1 \vee x = 3\} \\
\{(x = 0 \vee x = 1) \wedge (x = 0 \vee x = 2)\} & x := x + 1 & \{x = 0 \vee x = 1\} \\
\{(x = 2 \vee x = 3) \wedge (x = 0 \vee x = 2)\} & x := x + 1 & \{x = 2 \vee x = 3\},
\end{array}$$

which are all trivial. By the rule for parallel flows, we have

$$\{x = 0\}\,S_1 \parallel S_2\,\{\neg ok \wedge x = 3\}$$

F. Communicating Services

Different services do not share variables and communicate by passing messages. The central concept in the method developed by Apt, Francez and de Roever [6] is the cooperation test. It checks that the postcondition of an input command is indeed ensured by the sending command. The Hoare triples of two matching communication pairs

$$\{p_1\}\,\texttt{rec}\ a\ x\,\{q_1\}$$
$$\{p_2\}\,\texttt{rep}\ a\ e\,\{q_2\}$$

cooperate, if the following is true

$$\{\exists ok.\, p_1 \wedge p_2\}\,x := e\,\{\exists ok.\, q_1 \wedge q_2\}$$

For a set of services, the proof outlines cooperate if the Hoare triples of every two matching communication pairs does. Note the assertions in the verification of each service may contain ok and $comp$, and we rename them as ok_i and $comp_i$ to avoid conflicts among different services, and arrive at the following rule for services

$$\frac{\text{proof of } \{p_i\}\,P_i\,\{q_i\}\ \text{cooperate},\ i = 1, 2, \ldots, n}{\{p_1 \wedge p_2 \wedge \cdots \wedge p_n\}\,(P_1, P_2, \ldots, P_n)\,\{q_1' \wedge q_2' \wedge \cdots \wedge q_n'\}}$$

where $q_i' = q_i[ok_i/ok, comp_i/comp]$.

Example 3 Let $P_1 =_{df} x_1 := 0; \texttt{rep}\ a\ (x_1 + 1)$, $P_2 =_{df} \texttt{rec}\ a\ x_2; \texttt{rep}\ b\ (x_2 + 2)$, $P_3 =_{df} \texttt{rec}\ b\ x_3$.

For P_1, we have the following proof outline

$$\begin{array}{c}\{\texttt{true}\}\\ \{ok\}\\ x_1 := 0\\ \{ok \wedge x_1 = 0\}\\ \texttt{rep}\ a\ (x_1 + 1)\\ \{ok \wedge x_1 = 0\}\end{array}$$

For P_2,

$$\begin{array}{c}\{\texttt{true}\}\\ \{ok\}\\ \texttt{rec}\ a\ x_2\\ \{ok \wedge x_2 = 1\}\\ \texttt{rep}\ b\ (x_2 + 2)\\ \{ok \wedge x_2 = 1\}\end{array}$$

For P_3,

$$\begin{array}{c}\{\texttt{true}\}\\ \{ok\}\\ \texttt{rec}\ b\ x_3\\ \{ok \wedge x_3 = 3\}\end{array}$$

There are two matching communication pairs. For cooperation test, we need to check

$$\begin{array}{l}\{x_1 = 0\}\ \ x_2 := x_1 + 1\ \ \{x_2 = 1\}\\ \{x_2 = 1\}\ \ x_3 := x_2 + 2\ \ \{x_3 = 3\}\end{array}$$

which are all trivial. It follows that

$$\begin{array}{c}\{\texttt{true}\}\\ (P_1, P_2, P_3)\\ \{ok_1 \wedge ok_2 \wedge ok_3 \wedge x_1 = 0 \wedge x_2 = 1 \wedge x_3 = 3\}\end{array}$$

IV. Conclusion

There has been some work on applying formal methods to web services. An operational semantics of StAC (Structured Activity Compensation) [9], another business process modeling language where compensation acts as one of its main features, has also been studied in [4]. StAC and the B method has been combined in [10] to describe business transactions. Bruni *et al.* [11] have studied the transaction calculi for Sagas. The long-running transactions were discussed and a process calculi was proposed in [12] in the context of a Java API, namely the Java Transactional Web Services. Laneve and Zavattaro [13] explored the application of π-calculus in the formalization of the semantics of the transactional construct of BPEL. They also studied a standard pattern of Web Services composition using π-calculus. For verifying the properties of long-running transactions, Lanotte *et al.* [14] have explored their approach in a timed framework, where a Communicating Hierarchical Timed Automata was developed. Model checking techniques have been applied in the verification of properties of long-running transactions.

In comparison, there has been little work on deductive reasoning of BPEL-like programs. As far as we know, Luo *et al.* [15] were the first to study a Hoare logic for BPEL-like programs. The work has not covered concurrent behaviours. Parallelism in one service has been considered in [8], and the rely/guarantee [16] approach to verifying shared variable programs is adopted. The same approach (instead of rely/guarantee, usually named as assumption/commitment) for message passing, although also available, e.g., see [17] for a survey, is more difficult to use. Therefore, in this paper, we decide to adopt the earlier cooperation test approach from Apt, Francez and de Roever. To be consistent in the style, the method of interference freedom test from Owicki and Gries is adopted to deal with shared variables.

In this paper, we focus on the deductive reasoning of BPEL-like programs in one unified framework, especially the verification of concurrent communicating BPEL programs. Verification methods developed for shared variables and message passing are integrated. To deal with the compensation and fault handling of web services and facilitate the verification, we introduce ok and present the corresponding rules. There are a few minor technical improvements over [8] in the way ok is used. Examples are provided to show the feasibility of verification framework.

Acknowledgment

This work is supported in part by National Key R&D Program of China (No. 2017YFC0803700), Macao Science and Technology Development Fund under the EAE project (No. 072/2009/A3), Ministry of Public Security of China (No. 2017GABJC16), Natural Science Foundation of China (No. 61602177 and No. 61402176). The authors would like to thank J.W. Sanders, C. Ma and X. Liu for discussions.

References

[1] S. Thatte, *XLANG: Web Service for Business Process Design*. Microsoft, 2001, http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.html.

[2] F. Leymann, *Web Services Flow Language (WSFL 1.0)*. IBM, 2001, http://www-3.ibm.com/software/solutions/webservices/pdf/WSDL.pdf.

[3] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, M. Satish Thatte, and S. Weerawarana, *Business Process Execution Language for Web Service*, 2003, http://www.siebel.com/bpel.

[4] M. J. Butler and C. Ferreira, "An operational semantics for StAC, a language for modelling long-running business transactions," in *Proc. COORDINATION 2004: 6th International Conference on Coordination Models and Languages, Pisa, Italy, February 24–27, 2004*, ser. Lecture Notes in Computer Science, vol. 2949. Springer-Verlag, 2004, pp. 87–104.

[5] S. S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I," *Acta Informatica*, vol. 6, pp. 319–340, 1976. doi: 10.1007/BF00268134. [Online]. Available: https://link.springer.com/article/10.1007/BF00268134

[6] K. R. Apt, N. Francez, and W. P. D. Roever, "A proof system for communicating sequential processes," vol. 2, no. 3, pp. 359–385. doi: 10.1145/357103.357110. [Online]. Available: http://dl.acm.org/citation.cfm?id=357110

[7] Z. Qiu, S. Wang, G. Pu, and X. Zhao, "Semantics of BPEL4WS-Like fault and compensation handling," in *Proc. FM 2005: International Symposium of Formal Methods Europe, Newcastle, UK, July 18–22, 2005*, ser. Lecture Notes in Computer Science, vol. 3582. Springer-Verlag, 2005, pp. 350–365.

[8] H. Zhu, Q. Xu, C. Ma, S. Qin, and Z. Qiu, "The rely/guarantee approach to verifying concurrent bpel programs," in *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7504. Springer, 2012. doi: 10.1007/978-3-642-33826-7_12 pp. 172–187. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-33826-7_12

[9] M. J. Butler and C. Ferreira, "A process compensation language," in *Proc. IFM 2000: 2nd International Conference on Integrated Formal Methods, Dagstuhl Castle, Germany, November 1–3, 2000*, ser. Lecture Notes in Computer Science, vol. 1945. Springer-Verlag, 2000, pp. 61–76.

[10] M. J. Butler, C. Ferreira, and M. Y. Ng, "Precise modelling of compensating business transactions and its application to BPEL," *Journal of Universal Computer Science*, vol. 11, no. 5, pp. 712–743, 2005.

[11] R. Bruni, H. C. Melgratti, and U. Montanari, "Theoretical foundations for compensations in flow composition languages," in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. ACM. doi: 10.1145/1040305.1040323. ISBN 978-1-58113-830-6 pp. 209–220. [Online]. Available: http://doi.acm.org/10.1145/1040305.1040323

[12] R. Bruni, G. L. Ferrari, H. C. Melgratti, U. Montanari, D. Strollo, and E. Tuosto, "From theory to practice in transactional composition of web services," in *Formal Techniques for Computer Systems and Business Processes*, ser. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg. doi: 10.1007/11549970_20 pp. 272–286. [Online]. Available: https://link.springer.com/chapter/10.1007/11549970_20

[13] C. Laneve and G. Zavattaro, "Web-pi at work," in *Proc. TGC 2005: International Symposium on Trustworthy Global Computing, Edinburgh, UK, April 7–9, 2005*, ser. Lecture Notes in Computer Science, vol. 3705. Springer-Verlag, 2005, pp. 182–194.

[14] R. Lanotte, A. Maggiolo-Schettini, P. Milazzo, and A. Troina, "Design and verification of long-running transactions in a timed framework," *Science Computer Programming*, vol. 73, no. 2-3, pp. 76–94, 2008. doi: 10.1016/j.scico.2008.07.001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642308000774

[15] C. Luo, S. Qin, and Z. Qiu, "Verifying bpel-like programs with hoare logic," in *Proc. TASE 2008: 2nd IEEE International Symposium on Theoretical Aspects of Software Engineering*. Nanjing, China: IEEE Computer Society, June 2008, pp. 151–158.

[16] C. B. Jones, "Tentative steps toward a development method for interfering programs," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 4, pp. 596–619, 1983. doi: 10.1145/69575.69577. [Online]. Available: http://dl.acm.org/citation.cfm?id=69577

[17] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, and M. P. J. Zwiers, *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge Tracts in Theoretical Computer Science 54, Cambridge University Press, 2001.