RC23962 (C0605-012) May 22, 2006
Computer Science

IBM Research Report

Formalization, Verification and Restructuring of BPEL Models with Pi Calculus and Model Checking

Ke Xu[1,2], Ying Liu[2], Geguang Pu[2]

[1]Department of Automation
Tsinghua University
Beijing, China 100084

[2]IBM Research Division
China Research Laboratory
HaoHai Building, No. 7, 5th Street
ShangDi, Beijing 100085
China

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Formalization, Verification and Restructuring of BPEL Models with Pi Calculus and Model Checking

Ke Xu [2 1], Ying Liu [2], Geguang Pu [2]
[1] *Department of Automation, Tsinghua University, Beijing China 100084*
[2] *IBM China Research Laboratory, Beijing, China 100085*

Abstract

BPEL (Business Process Executable Language for Web Services) is an emerging standard for business application integration and B2B processing based on web services. As a popular specification for modeling and implementing business processes, building reliable and secure business application systems with BPEL becomes an important issue. In this work, BPEL models are automatically verified and analyzed with our Open Process AnaLyzer (OPAL) toolkit by formally capturing BPEL's semantics in Pi Calculus. The contribution of the work can be concluded in four points. First, the semantics of BPEL is fully formalized with Pi calculus; Second, the soundness of the formalization is validated and important properties are proved to be preserved in the formalization. Third, a concrete scenario is illustrated to show how model checking is applied to verify the reliability of BPEL model designs. Last, equivalence analysis in Pi calculus and model checking are combined to implement restructuring algorithms by which a BPEL model can be restructured for performance enhancement with OPAL.

1. Introduction

BPEL [1] is a de facto standard for business application integration and business-to-business processing based on web services. As a language for modeling both executable and abstract e-business processes, the effective design of e-business applications based on BPEL is a serious issue for the avoidance of unexpected behaviors and errors in e-business systems. Such a motivation comes from two aspects: (1) How to ensure the design of a BPEL model conforms to specific business constraints? Major business integration systems including Websphere Business Integrator of IBM support the direct transformation from a business conceptual model to a BPEL model as its implementation. It is thus important to verify that such a transformation is error-free, in that the resulting BPEL model captures all business requirements in the conceptual model without loss of information; (2) How to ensure the design of a BPEL model is semantically correct and reasonable in order to implement a reliable e-business application? Moreover, since adjusting process structures is a common behavior for business / IT consultants to improve the performance of a process design, how is it possible to instruct such adjustment through model analysis?

In fact, the urgent need for the solution of these problems in e-business world has already been recognized in [2] and [3]. In reliability-critical business domains like banking, it is critical that its core businesses must follow all business regulations and legal compliances in the industry. For example, a bank should comply with: "*An account can only be opened after detailed information of the customer is retrieved and his / her identity is successfully verified*" in order to prevent financial frauds. This fictional requirement is made based on the financial institution of U.S. Patriot Act (Sect 326). Without ensuring the requirement, not only the bank is vulnerable to economic losses, but also the business operation itself is illegal. We will provide this e-business model in detail with BPEL in section 4.3, and show how this model is verified and analyzed to avoid the violation of the above requirement.

The purpose of this paper is to address the previous two issues. We study the application of model checking [4] to automatically reason about the behavior of BPEL models with our Open Process AnaLyzer (OPAL) [5] toolkit. Critical issues in verifying BPEL models are addressed. Firstly, the semantics of BPEL is fully formalized with Pi Calculus [6]; secondly, algebraic properties and specific features in BPEL specification [1] are proved to be preserved in our formalism; thirdly, to reason about the reliability of BPEL models, it is shown how model checking is exploited for the automatic verification of BPEL models against desired properties; fourthly, as a further step it is also studied based on both equivalence analysis of Pi calculus and model checking how a BPEL model can be equivalently restructured to potentially gain a better performance.

The organization of the paper is as follows. Section 2 compares existing works on both BPEL formalization and model checking tools. In section 3, the full formalization of BPEL with Pi calculus is investigated. Section 4 proves that important laws and properties are well preserved in our formalism. Besides, we illustrate how model checking is used to analyze BPEL models with a concrete scenario. In section 5, transformation rules are deduced to implement restructuring algorithms that (semi-)automatically restructure a BPEL model. Section 6 concludes the paper.

2. Related Work and OPAL Methodology

It is already a general idea that formal models are necessary for complex Web Service Composition Languages (WSCLs) like BPEL [7]. In [8], the abstract operational semantics of BPEL is defined based on the Abstract State Machine (ASM) paradigm. Andreas, et al [9] translate BPEL into deterministic finite state

automata to support the matchmaking of state dependent services. Farahbod, et al [10] formally define the abstract executable semantics of BPEL based on Distributed Abstract State Machine (DASM). In [11], Petri-net is used to formalize BPEL constructs and several properties including dead transitions are checked. Our work differs from theirs in two aspects. (1) Pi calculus is chosen as the formal foundation of BPEL. Though BPEL is claimed to be based on Pi calculus, yet little work "actually provides solid semantics and analysis methods"[7]. As a process algebra with mobility and compositionability, previous work has already proved it a competent formal composition language for web service composition [12]. This is why Pi calculus is applied in this paper. (2) Our work goes beyond formal BPEL formalization and validating the formalisms with general properties like deadlocks. It investigates the topics of semantic validation of our BPEL formalization, temporal verification and also the restructuring of BPEL models with a concrete application scenario.

As far as model checker for Pi calculus is concerned, Mobility Workbench (MWB) [13] and HD-Automata Laboratory (HAL) [14] are two important ones. Their theoretical foundations, however, differ to each other in significant ways. MWB analyzes Pi calculus processes based on Dam's proof system for model checking mobile processes [15]. HAL, on the other hand, transforms Pi calculus into an automata based on Pi calculus's early transition semantics. Its verification is based on the integration of JACK model checker. Our OPAL toolkit follows the latter methodology. Figure 1 shows an overview of its architecture.

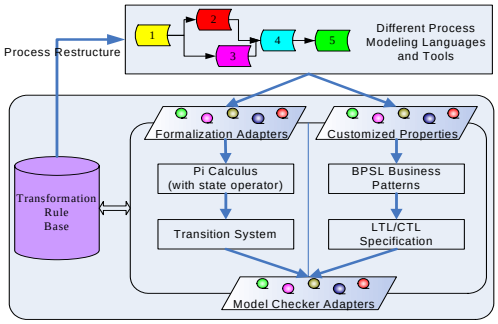

Fig. 1. Architecture of OPAL Toolkit.

Different from HAL, OPAL focuses itself more on the usability and scalability issue in order to make it practical to introduce the rigid model checking into e-business domain. It further addresses the following business domain specific verification problems: (1) By usability, OPAL supports the automatic formalization of different business process models (including BPEL) with Pi Calculus. It also enables the translation of Pi calculus processes to finite state transition system, during which deadlocks and redundant activities in a process are detected. (2) Property specification is a serious obstacle for applying model checking in e-business domain [5] since common logical formulae are too complex to use and understand. OPAL solves the problem by providing a visualized Business Property Specification Language (BPSL) [5], with intuitive business property patterns. (3) By scalability, OPAL provides an open integration environment for various business process modelers including BPEL Designer [16], WBI, etc (through formalization adapter) and different model checkers like NuSMV [17] (through model checker adapter). Separate topics including BPSL and OPAL are addressed in our separate work [5]. This paper primarily focuses on the formalization, verification and restructuring of BPEL models with OPAL.

3. Formalizing BPEL with Pi Calculus

3.1. An Overview of Pi Calculus and BPEL

The version of Pi-calculus used here is the polyadic Pi calculus [6] with the allowance of negation "¬". Its syntax is summarized as follows.

$$P ::= \sum_{i=1}^{n} \pi_i.P_i \mid new\ x\ P \mid !P \mid P \mid Q \mid \phi P \mid A(y_1,...,y_n) \mid 0$$

$$\pi_i ::= x'<y> \mid x(y) \mid \tau \qquad \phi ::= [x=y] \mid \phi \wedge \phi \mid \neg\phi$$

The simplest entities of Pi calculus are names (in lowercase) and processes (in uppercase). Processes can evolve by performing actions. *x'<y>* is an **output** action which sends name *y* via *x* and *x(z)* is an **input** action which receives a name via *x*. In **composition** *P*|*Q*, process *P* and *Q* proceeds independently and interacts via shared names. A **sum** $\pi_1.P_1 + \ldots + \pi_n.P_n$ is a non-deterministic choice of process execution. In **restriction** (*new x*) *P*, the usage of name *x* is bounded to *P*. **Replication** *!P* is an infinite composition *P* |*P* | …. ϕP is a process that is guarded by a Boolean expression ϕ for **name matching**. For example, *[x=y]P* means that *P* can be performed if name *x* is the same with *y*. *A(y1,…, yn)* is a parameterized process **identifier**, with its parameters to be the free names of the process.

On the other hand, BPEL can be regarded as a business workflow language designed for web services. A complete list of relevant model elements in BPEL is presented in the following.

Activity	*Activity*	*Structure*	*Structure*	*Scope*
Receive	**Empty**	**Sequence**	**Pick**	**Compensation**
Reply	**Termination**	**Switch**	**Flow**	**FaultHandler**
Invoke	**Correlation**	**While**	**Link**	**Throw**
Assign				**Scope**

Since everything is a process in Pi calculus, detailed semantics of BPEL will all be formally captured as Pi processes in the next sub-sections. Consequently, the **composition** of these Pi processes forms the specification of the BPEL application (e.g. the one in 4.3) that is composed by these elements.

3.2. Formalizing BPEL Activity Constructs

3.2.1. Variable

BPEL contains program variables which are assigned with values. Here the semantics of a program variable holding a value of *x* can be defined as a 'storage location' by a storage register as follows:

Variable(x) = Reg(x)

Reg(x) = put(y).Reg(y) + get'<x>.Reg(x)

The above formalization means that the stored value *x* of the variable can be read from the storage location via action *get'<x>*, and a new value *y* can be written into the location via *put(y)*.

3.2.2. Basic Semantics of BPEL Activity Constructs

The basic activities of BPEL (*Receive, Reply, Invoke, Assign, Empty*) define how message communication, service invocation and variable assignment are done. The basic formalization of these activities is in the following. More complex semantics like message correlation, global termination, etc are addressed laler.

Receive(start, γ, put, done) = start.γ(v).put'<v>.done'
Reply(start,get, γ ,done)=start.get(v).γ'<v>.done'
Invoke(start,get, γ ,put, done) =
start.get(v). γ' <v>.γ(w).put' <w>.done'
Assign(start,get,put,done) =
new c (start.get(v).c' <v> | c(x).put' <x>.done')
Empty(start,done) = start.done'

'*link name*', '*partner name*' and '*operation name*' are three elements in BPEL activities and they often appear at the same time. Therefore, here they are denoted as a unified name '*γ*'. The parterLinks and data sharing in these BPEL activities are mapped to the input and output prefixes of '*γ*', '*get*', '*put*'. In addition, two special names '*start*' and '*done*' are used to indicate common internal communications in a BPEL process.

3.2.3 Message Correlation in BPEL Activities

Typically, multiple conversations exist in multiparty business interactions. BPEL uses correlation sets to identify different conversation and route messages to the correct service instance. Each participant process in a correlated message exchange can serve either as an initiator or as a follower of the exchange. The initiator is responsible for initiating the value of the correlation set, which can be thought of as an alias for the identity of the business process instance. To implement this feature, the "alias for the identity of the business process instance" can be represented by a restricted name in Pi calculus. It is the initiator's responsibility to receive and store this restricted name in correlation set so that the follower can use it as a private channel to communicate with the correct process of service instance. Take asynchronized message exchange as an example, the correlation semantics can be further encoded into the *Receive* / *Reply* activity in 3.2.2 as:

$$Receive(start,\gamma,put,done) = start.\gamma(v,identity).new\ ack(put'<v>.ack'|init'<identity>.ack'|ack.ack.done')$$

$$Reply(start,\gamma,get,done) = start.get(v).new\ chan(retr'<chan>.chan(identity).identity'.\gamma'<v>.done')$$

$$CorrelationSet(identity,init,retr) = retr(chan).chan'<identity> + init(identity).CorrelationSet(identity,init,retr)$$

The *CorrelationSet* is implemented as a variable. *Reply* will use the *identity* initiated by *Receive* as a private channel to guard the sending of *v* to a correct service.

3.2.4 Global Termination of Activity

In BPEL, activity instance is interrupted and forced to terminate if a terminate activity or a fault is reached. To implement this, three steps can be followed. First, a private termination channel represented by a restricted name (*term*) should be added to each activity. Each activity registers its private *term* channel once it is instantiated and listens to this channel for **receiving** the **termination** (*term*) of its execution. Secondly, an InstanceManager (*InstMgr*) is implemented to record the *term* channels of each activity. The register (*reg*) and remove (*rm*) channel is used for the registration of removal of the *term* channels and head channel (*h*) is used to retrieve the first *term* channel stored in the manager. Note that in our real implementation, a max number of the *term* channel must be set to control the size of the *InstMgr* to make its behavior remain ***finite***.

$$InstMgr_0(reg,rm,empty,h) = reg(term_1).$$
$$InstMgr_1(reg,rm,empty,h,term_1) + empty(t,f).t'.InstMgr_0(reg,rm,empty,h)$$
$$InstMgr_n(reg,rm,h,term_1,...,term_n) = (reg(term_{n+1}).$$
$$InstMgr_{n+1}(reg,rm,empty,h,term_1,...,term_{n+1}) + rm(term_j).$$
$$InstMgr_{n-1}(reg,rm,empty,h,term_1,...,term_{j-1},...,term_{j+1},term_n) +$$
$$empty(t,f).f'.InstMgr_n(reg,rm,empty,h,term_1,...,term_n) + h(chan).$$
$$chan'<term_1>.InstMgr_{n-1}(reg,rm,empty,h,term_2,...,term_n)$$
$$InstMgr = InstMgr_0 \qquad n \geq 1, 1 \leq j \leq n$$

Thirdly, the termination activity should recursively retrieve the *term* channels from *InstMgr* and force the termination of each instantiated activity.

$$Termination(start,empty,h) = new\ chan\ t\ f(start.empty'<t,f>.(f.h'<chan>.chan(term).term'.(start'|Termination) + t.0))$$

3.3 Formalizing BPEL Structure Constructs

3.3.1 Basic Semantics of BPEL Structure Constructs

BPEL Structures imply different control relations (e.g. sequence, choice, etc) between BPEL activities. Define function *fn(P)* to be the set of all free names in a Pi Calculus process identified by *P*; denote also $start_P$ to be the *start* name of *P* (see 3.2.2). Consequently, the basic semantics of the five structures are defined as:

Seq (fn(P),fn(Q)) = new $start_q$ *({*$start_q$*/done}P | Q)*
Switch(b1,fn(P),b2,fn(Q)) =
[b1]P + [¬ b1 ∧ b2]Q + [¬ b1 ∧ ¬ b2] Empty
While(b,fn(P))=[b](Seq(fn(P),fn(While)) + [¬b]Empty
Pick (γ_1*, fn(P1),* γ_2*, fn(P2), put)=*
(new c(γ_1 *(v).put' <v>.c'|c.P1)) +*
(new c(γ_2 *(v').put' <v'>.c'| c.P2))*
Flow (fn(P),fn(Q),done) =
new ack ((new $done_1${ $done_1$*/done}P | *$done_1$*.ack')|*
(new $done_2$ *{*$done_2$*/done}Q|*$done_2$*.ack')|ack.ack.done')*

In the formalization, {*start/done*} is a substitution in Pi calculus, which means that name *done* is replaced by *start* so that an internal interaction can be formed between *P* and *Q*. Note the above *Switch* implies that when several branching conditions hold at the same time, the branches are taken in the order in which they appear, which follows exactly the semantics of BPEL.

3.3.2 Synchronizing with Links

The synchronization dependencies between activities are expressed by *link* in BPEL. Activity with incoming links will not start until the following three conditions are satisfied: (1) Its preceding activity is completed; (2) The status of all its incoming links has been determined, e.g., by *bpws:getLinkStatus* function; (3) The join condition of the activity is true, otherwise a standard *joinfailure* is thrown. Let *BPELAct* represent any

activity formalized in 3.2.2, and let *Links* be all links that targets to this activity, the feature is formalized as.

$$Link_i(fn(EvalTrans_i), ack, nack) = new\ pos, neg(EvalTrans_i | (pos.ack' + neg.nack'))$$

$$EvalTrans_i(done_{in}, pos, neg, neglink_{in}) = done_{in}.pos' + neglink_{in}.neg'$$

$$Links(fn(Link_i), done_{links}, eliminate) = new\ ack\ nack(Link_1 | ... | Link_n | \underbrace{ack.(... .(ack}_{n}.done_{links}' + nack.eliminate') ... + nack.eliminate')) \quad i = 1, ..., n$$

$$ActivityWithLinks(done_{preceding}, fn(links), evalJoin, neglink_{out}) = done_{preceding}.(done_{links}.new\ t\ f(evalJoin' < t, f >.(t.BPELAct + f.Throw(joinfailure)))) + eliminate.(neglink_{out}' | ... | neglink_{out}'))$$

Here $done_{in}$ indicates the proper termination of the source activity of the link, $done_{preceding}$ indicates the proper termination of preceding activity (if there is any, or otherwise it is simply removed). Note when a death-path is detected (e.g. if a branch in a *Switch* is not selected), the above formalization also captures the propagation of negative tokens (neg_{link}) through the outgoing links of an activity which conforms to the BPEL specification of links. This ensures that the execution of activities in a death-path is eliminated. The implementation of *Throw* can be referred in 3.4.1.

3.3.3 Manipulation of Timeout

The *Pick* activity is used to block and wait for the arrival of a suitable message or a *timeout* alarm to go off. Since the semantics of picking a suitable message is specified in the previous section, the manipulation of timeout alarm in *Pick* activity is addressed here. In the following formalism, *starttimer* is used to trigger the evaluation of deadline conditions by a proper timer. The omitted part '…' is the corresponding formalism of *Pick* in 3.3.1. An extra *timeout* input is added to listen to the competition for the arrival of a timeout event.

$$Pick(..., starttimer, deadline) = new\ timeout\ (starttimer' < deadline, timeout >.(..... + timeout.done'))$$

To simulate the observation of time with Pi calculus, the following pseudo-implementation of a Timer is provided. Here *wait* indicates the elapse of a default unit of time if the deadline expression is evaluated to be false, otherwise a timeout will be sent out immediately.

$$Timer(starttimer, deadline, timeout, eval) = new\ wait\ (starttimer(deadline, timeout).\ (Timer | TimeEval(deadline, timeout, eval))$$

$$TimeEval(deadline, timeout, eval) = new\ t\ f(eval' < deadline, t, f >.\ (f.(wait' | wait.TimeEval) + t.timeout'))$$

To better specify the real time aspects in BPEL, a timed version of Pi calculus [18] can also be used instead.

3.4 Formalizing BPEL Scope Constructs

3.4.1 Fault Handling and Compensation

Fault handling and compensation is an important issue in BPEL. However, its implementation in BPEL can be complex since it involves the concept of scope (refer to 3.4.2) in related to fault / compensation handlers. Take the fault handlers specified within an *Invoke* construct as an example. Since the fault / compensation handlers specified within a scope do not change dynamically, the names for all fault / compensation handlers within a scope can thus be pre-defined and stored in vectors (*FHandlers* and *Chandlers*). Therefore the feature of fault handling and compensation in BPEL is formalized in the following.

$$Invoke(\gamma, get, put, done, fault, \overrightarrow{type_i}) = get(v).\gamma' < v >.\ (\gamma(w).put'(w).done_{inv}' + fault(ftype).\ FaultHandling(ftype, \overrightarrow{type_i}, rethrow, done))$$

$$FaultHandling(ftype, \overrightarrow{type_i}, rethrow, done) = [ftype = type_1]BPELAct_1 + ... + [ftype = type_n]BPELAct_n + \sum_{i \in I, i \neq 1..n} [ftype = type_i](fready_{fname}' < ftype > | compensate_{cname_1}'. ..... .compensate_{cname_n}')\ (cname_i \in \overrightarrow{CHandler_{currentscope}}, fname \in \overrightarrow{FHandler_{parentscope}})$$

In the above, the fault type (*ftype*) of $type_1$, … $type_n$ can be caught during the execution of *Invoke* and corresponding *BPELAct* will be launched. If the fault handler for a fault is missed (e.g. for types other than $type_1$, … $type_n$), two things will happen. One is that the fault handler will *rethrow* this fault to the next enclosing parent scope through *fready* channel; the other is that it will try to invoke all compensation handlers for immediately enclosed scope. Note that in the process of *FaultHandling*, the output of compensate channel is in a sequential order such that it is possible for the invocation of compensation handlers to be in the reverse order of completion of the corresponding scopes [1]. Such formalization can be further combined with the one in 3.2.4 to implement the global termination feature in fault handling and compensation.

A Throw is responsible for the generation of a fault event and can be simply formalized as below.

$$Throw(fready, fault, ftype) = fready.fault' < ftype > .Throw$$

A *Compensationhandler (CompHandler)* is available for invocation only when its scope is completed normally. It can be invoked either explicitly by corresponding compensate activity or implicitly by the behavior of the implicit fault handler created by BPEL (which is the case of the above implementation). Therefore, the formalization of compensation handler is:

$$CompHandler(done, compensate) = done.compensate.new\ done_{BPELAct}(BPELAct | done_{BPELAct}.CompHandler)\quad (compensate \in \overrightarrow{CHandler_{CurSope}})$$

The action *done* is generated by the activity in the corresponding scope (in this case, it is the immediately enclosed scope of *Invoke* and we named it *CurScope*) and its reception in *CompHandler* indicates the normal completion of the scope. The *BPELAct* in the process of *FaultHandling* and *CompHandler* is the actual activity that is specified in the corresponding fault handler or compensation handler. The effective scope (in the next section) is extremely important in implementing the error handling feature of BPEL and must be correctly modeled when composing the *Throw*, *FaultHandling* and *CompHandler* by restricting free names (e.g. *fault*, names in *CHandler* and *FHandler*) in these processes.

3.4.2 Formalizing Scope in BPEL

In BPEL, scope is used to define an effective range of the usage of variables, compensation / fault handlers and other activities. Considering all these BPEL constructs can be related to an effective scope, we

collect their free names in a scope with a predefined function, *GetNames(s)*, where *s* is the scope. Hence the restriction operator 'new' us used to restrict the access to these elements according to their effective scope.

Scope ($\overline{restnames}$) =new GetNames(s) (P1 | P2 | …| Pn)

where *Pi* (*i=1,…,n*) can be a Pi process for any activity, structure, or scope constructs defined in this paper, and $\overline{restnames}$ is defined as a free name set of

(fn(P1)∪fn(P2)∪……∪fn(Pn))/GetNames(s)

4. Model Checking BPEL Processes

In this section, our above formalizations will be validated first with model checking to ensure that they do not violate the semantics of BPEL 1.1 specification, and important algebraic laws are well-preserved. Based on the validation, we can thus continue to use model checking to verify a BPEL application against ad-hoc user desired properties to ensure its reliability.

4.1 Correctness Validation of BPEL Formalization

It is critical to prove that the formalization in section 3 conform exactly to the semantics of BPEL in order to raise the confidence in their correctness. To do that, BPEL specifications can be hand coded as logical formulae and model checking is applied with our OPAL toolkit to check the previous formalizations against BPEL specification. Although it is difficult (if not impossible) to strictly prove the completeness of the correctness checking, model checking with its tool support does provide an effective way to verify any correctness criterion that one has in his / her mind.

With the size limitation, an important entry in the BPEL specification, "Synchronizing with links" in 3.3.2, is chosen as the example. There are two important aspects in validating the formalization of "links". One is "synchronization", which means the three conditions in 3.2.2 must be satisfied before an activity with incoming links can start; the other is "propagation", which is used to model Death-Path-Elimination in BPEL. To apply OPAL for the validation, the above semantics are specified with temporal logics of LTL [4] or CTL [4].

First we check the reachability of the *BPELAct*: Note *neglink*, *evalJoin* are **names** in the formalism of 3.2.2.

G ! neglink -> F evalJoin / LTL Spec 1*

It means that when negative links never happen, the action for evaluating join conditions (*evalJoin*) will eventually be executed in our formalization of link. In turn, if the evaluated result (*t* is received after *evalJoin*) is true, the corresponding *BPELAct* can be started.

Second, we check the "synchronization" semantics:

!E [(!done$_{preceding}$ | !done$_{in1}$ | ... | !done$_{inn}$) U evalJoin]
/ CTL Spec 2*

It ensures that there is no computation path in the formalization in which before the possible execution of *evalJoin*, a "*done*" action has never been executed.

Last, we check the "propagation" semantics:

AG (neglink$_{in}$ -> AF neglink$_{out}$) / CTL Spec 3*

This shows that whenever a negative link NL is evaluated, it will be propagated to all outgoing links of the activity whose incoming links contain NL.

The validation procedure is simple and automatic. OPAL takes both the BPEL script (which in this case is a single *link* construct) and the above formulae in LTL and CTL as inputs. With the result of 3.3.2, OPAL is able to automatically transform the link construct into Pi formalization. After this step is done, OPAL detects that there are no deadlocks and redundant actions in the formalization by itself. Meanwhile it also enables the integration of different model checkers for model checking LTL/CTL on the *link* construct. For example, in our current implementation, a NuSMV2 adapter is available in OPAL so that OPAL can automatically transform the above two inputs to the language of NuSMV2 [17] and invoke the validation on the engine. With this capability, the validation result shows that our formalization satisfies all of the above three formulae (all verified to be true by NuSMV2). More detailed application scenarios of OPAL can be found in the following sections where OPAL will also be used for the verification and structural analysis of BPEL models.

4.2 Property Preservation in BPEL Formalization

Aside from the validation, interesting laws are also given in this section to show that common algebraic properties are satisfied in our formalization. These laws are deduced based on the weak bi-simulation (denoted as '~'), which is a useful approach to show the equivalence between different Pi calculus processes. The identified laws (in 4 groups) are listed below.

Group 1: showing permutation laws are well-preserved

[Law 1] Sequence(fn(P),Sequence(fn(Q), fn(R))) ~
Sequence(Sequence(fn(P), fn(Q)), fn(R))

[Law 2] Flow(fn(P), fn(Q)) ~ Flow(fn(Q), fn(P))

[Law 3] Flow(fn(P), Flow(fn(Q), fn(R))) ~
Flow(Flow(fn(P), fn(Q)), fn(R))

Group 2: showing associative laws are well-preserved

[Law 4] Sequence(fn(Switch(b, fn(P),¬b, fn(Q))),fn(R))
~ Switch(b, fn(Sequence(fn(P),fn(R))),
¬b, fn(Sequence(fn(Q),fn(R))))

[Law 5] Flow(fn(Switch(b, fn(P),¬b,fn(Q))), fn(R),done)
~ Switch(b, fn(Flow(fn(P), fn(R))),
¬b ,fn(Flow(fn(Q), fn(R),done)))

Group 3: *branches in Switch are taken in the order in which they appear, and only one branch is taken*

[Law 6] Switch(b1 , fn(P), b2 , fn(Q)) ~
Switch(b2 ,fn(Q), b1 , fn(P)),
if b1 and b2 does not hold at the same time

[Law 7] Switch(b1 , fn(P), b2 , fn(Q)) ~ P
if b1 and b2 hold at the same time

[Law 8] Switch(b , fn(P), ¬b, fn(Q)) ~ P

Group 4: *showing the property of Empty*

[Law 9] Sequence(EmptyAct, fn(P)) ~ P

[Law 10] Flow(EmptyAct, fn(P)) ~ P

Here we only give the formal proof of Law 1 and 9 as examples. Other proofs are omitted here since they have strong similarity with these example proofs.

Proof 1:

RHS = new s_1 ({s_1/done} new s ({s/done}P | s.Q) | s_1.R)
= new s_1 (new s ({s/done}P | { s_1/done} s.Q)| s_1.R))
= new s_1 s ({s/done}P | { s_1/done}start.Q) | s_1.R)

/ structural congruence of Pi calculus*

LHS = new s({s/done}P | s. new s_1 ({s_1/done}Q| s_1.R))

= new s_1 s ({s'/done}P | s_1.({s/done]Q | s.R))

/ structural congruence of Pi calculus*

RHS ~ LHS ⇒ Law 1 holds. □

Proof 9:

LHS = new s ({s/done} Empty | P)

= new s (start.s'.0 | P) / Def. of EmptyAct*

~ P / the start name of P is 's';*

RHS ~ LHS ⇒ Law 9 holds. □

4.3 Verification Scenario of BPEL Process

To truly ensure the reliability of e-business applications, we further verify specific BPEL models against ad-hoc business requirement from user. Recall the banking example mentioned in the introduction, its realistic BPEL implementation is illustrated in figure 2 (though in the current implementation of OPAL, BPEL Designer [16] is the integrated BPEL modeler, figure 2 still illustrates our application in a general graphical form to make it independent of specific BPEL model designer). Though the size of the model is moderate, it contains most BPEL constructs including: *sequence* (dashed arcs), *link* (arcs), *flow* (rounded rectangle), *switch* (dashed rectangle), *basic activity* (rectangle), *variable* (column with mid-arrows) and *compensation* (octagon). The Pi calculus semantics for this complete banking BPEL process can be directly got by the composition of corresponding formalization of the model elements based on the results in section 3.

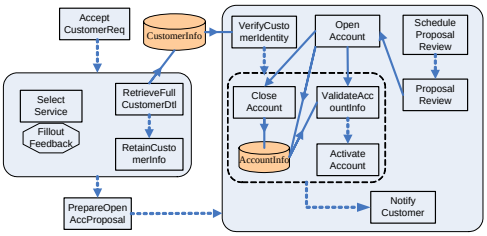

Fig. 2. BPEL Process Example – Account Opening.

The BPEL model represents the operation process of opening an account for specific customers. However, before actually implementing the application, it is necessary to ensure that the design of the BPEL model satisfies specific requirements of the process owner. Let us recall the requirement mentioned in the introduction:

Strong Security Protocol: *An account can only be opened after detailed information of the customer is retrieved and his / her identity is successfully verified.*

To verify the application against this requirement, first the BPEL model is automatically formalized by OPAL based on section 3. OPAL then translates the Pi formalizations into a state transition system based on its early operational semantics during which no deadlocks or unreachable activities in the process are found. The translated transition system contains totally 4498 reachable state, based on which OPAL will further symbolically merge the states that are associated with undistinguishable processes in order to reduce the state space. This step takes 3 seconds and the final transition system contains 1912 states and 4758 transitions. On the other hand, the requirement can be interpreted in the following basic set of LTL/CTL formulae:

G (! faultselect) -> G (RetrieveFullCustomerDtl ->

F VerifyCustomerIdentity) / LTL Spec 1*

! E [(! RetrieveFullCustomerDtl |

! VerifyCustomerIdentity) U OpenAccount]

/ CTL Spec 2*

With NuSMV2, the verification of the two formulae by OPAL takes 28 seconds. Results show the model successfully follows the first specification, indicating if exceptions in activity *SelectService* is not considered (*G (! faultselect)*), then whenever the customer detail is retrieved, his identity will be verified. But the model fails the second one. Counter example shows that there is a possible execution path on which the customer identity has not been verified when the account is opened. Thus a pitfall in the BPEL design is found that may lead to violation of the requirement. Some **snapshots** for OPAL toolkit can also be found in [19].

However, the reality is that checking a customer's identity is time consuming and in many cases, people in charge of the identity checking and the account opening works in different department. Therefore, restricting the account opening strictly after his identity checking may result in an inefficient operational process. The above strong security requirements should thus be relaxed.

Weak Security Protocol: *An account can only be opened after detailed information of the customer is retrieved. In the case when his / her identity is not successfully verified, the corresponding opened account must be closed as compensation.*

Similarly, the above requirement is interpreted into the following set of LTL/CTL formulae:

G ! (faultselect | joinfailure) ->

G (RetrieveFullCustomerDtl ->

(F VerifyCustomerIdentity & F OpenAccount))

/ LTL Spec 3*

!E [(! RetrieveFullCustomerDtl) U OpenAccount]

/ CTL Spec 4*

!E [(!OpenAccount) U CloseAccount]

/ CTL Spec 5*

G (! joinfailure) -> G (verificationfailed ->

F CloseAccount) / LTL Spec 6*

Specification 4 is a relaxation of specification 2 (since now *VerifyCustomerIdentity* does not need to execute strictly before *OpenAccount*). Specification 5 and 6 express the compensation requirement: if failures in the evaluation of join conditions in links are not considered (*G (! joinfailure)*), the identity verification failure will lead to the closing of an opened account. "*verificationfailed*" here is the name used to indicate a branch in the *Switch* structure. It took OPAL up to 62 seconds to finish verifying the four formulae. The results show that although the BPEL process model fails to follow the strong security protocol, it does satisfy the weak version of the security requirement.

5. BPEL Process Restructure

Now that a BPEL model is verified, this section addresses the further issue of "how to *equivalently and*

correctly restructure a BPEL model to potentially enhance its performance?". In this section, equivalence relation is defined with Pi calculus based on which important laws for BPEL structures are discovered. Restructure algorithm is then implemented in OPAL to support the automatic restructure of BPEL models.

5.1 State Equivalence and Restructure Rules

Common strong/weak bisimulation in Pi calculus orders strict (observable) behavior equivalence between two systems. However, in business domain it is too ideal to ask two processes to be exactly the same to qualify their equivalence. E.g., in the above account opening application, two BPEL models can be regarded as equivalent if their execution results in both the insertion of a correct customer record in the variable of "*CustomerInfo*" and a correct account record in "*AccountInfo*". More specifically, two e-business processes can often be regarded as equivalent if they run the same business task and generate the same result. Denote ***PUT(P)*** and ***GET(P)*** to be the set of name parameters of *put(v)* and *get(v)* in Pi process *P*; denote $\rightarrow^*$ to be multiple transitions fired by any action. Therefore, the following state equivalence relation of two BPEL models is defined with Pi calculus:

Definition 1: A process *P* can terminate with environment *Env*, if: $(P \mid Env) \rightarrow^* done' \mid Env'$.

Definition 2: Suppose *V* is a set of variables defined in BPEL, *P1* and *P2* are two processes. Then *P1* and *P2* are state equivalent on variable *V* in *Env*, denoted as $P1 \approx_{s,v,Env} P2$, if: (1) $P1 \mid V \mid Env \rightarrow^* done' \mid V^* \mid Env^*$ and $P2 \mid V \mid Env \rightarrow^* done' \mid V^{**} \mid Env^{**}$; (2) $V^* = V^{**}$.

Definition 2 specifies that two BPEL models are state equivalent, if they can terminate in the same environment and result in the same set of variables. As syntax sugar, denote *P;Q* and *P||Q* as abbreviation for *Seq(fn(P),fn(Q))* and *Flow(fn(P),fn(Q),done)* in section 3.3; The relaxation of state equivalence can lead to the following interesting results.

Property 1: $(P1;P2) \mid Env \rightarrow^* done' \mid Env^*$, iff $P1 \mid Env \rightarrow^* done' \mid Env^{**}$ *and* $P2 \mid Env^{**} \rightarrow^* done' \mid Env^*$.

Law 5.1: $P1;P2 \approx_{s,v,Env} P2;P1$ is satisfied if PUT(P1) ∩ GET(P2) = ∅ ∧ PUT(P2) ∩ GET(P1)= ∅ ∧ PUT(P1) ∩ PUT(P2)= ∅, where *P1;P2* can terminate with (*V* | *Env*) and PUT(Pj) / GET(Pj) are previously defined.

Proof: From the side condition, we know only *P1* or *P2* can exclusively change the values of *V*. As *P1;P2* can terminate with (*V*|*Env*), from Property 1 we have $P1 \mid V \mid Env \rightarrow^* done' \mid V^{**} \mid Env^{**}$ and $P2 \mid V^{**} \mid Env^{**} \rightarrow^* done' \mid V^* \mid Env^*$. Besides from the side condition, we have $V=V^{**}$ or $V^{**}=V^*$. Consequently, $P2;P1 \mid V \mid Env \rightarrow^* done' \mid V^* \mid Env^{**}$' as well. Therefore, based on state equivalence, we have $(P1;P2) \approx_{s,v,Env} (P2;P1)$□

Law 5.1 actually means that for two basic BPEL activities *P1* and *P2* in section 3.2.2, their sequential execution in *Seq* and parallel execution in *Flow* is not distinguished by state equivalence. In another word, when comparing two BPEL models by whether they run the same business tasks and generate the same results, sequential execution of activities can be equivalently replaced by a parallel execution. Therefore, it is possible to dig out more concurrencies in a process to more efficiently run those activities whose execution may otherwise be unnecessarily guarded. This important law makes us keep conducting the following state equivalent rules for transforming BPEL models.

Rule 1: $P1;P2 \approx_s P1 \,||\, P\_2$ if PUT(P1) ∩GET(P2) = ∅ ∧ PUT(P2) ∩GET(P1)= ∅ ∧ PUT(P1) ∩PUT(P2)= ∅.

Rule2.1: $(P1||P2);P3 \approx_s (P1;P3)||P2$ if PUT(P3)∩ GET(P2)=∅ ∧ PUT(P2)∩GET(P3)= ∅ ∧ PUT(P3)∩PUT(P2)= ∅.

Rule2.2: $(P1||P2);P3 \approx_s P1||(P2;P3)$ if PUT(P3)∩ GET(P1) = ∅ ∧ PUT(P1)∩GET(P3)= ∅ ∧ PUT(P3)∩PUT(P1)= ∅.

Rule 3: $(P1;(P2||P3) \approx_s (P1;P2)||P3$ if PUT(P3)∩ GET(P1) = ∅ ∧ PUT(P1) ∩GET(P3)= ∅ ∧ PUT(P3) ∩PUT(P1)= ∅.

These rules can be recursively applied to different sequential structures in BPEL model for discovering hidden concurrencies. Restructuring result of a BPEL model based on these rules can be ensured to be state equivalent to the original one. Besides by combining model checking, we can also further impose property preservation constraints on the resulted process.

5.2 Restructuring Algorithm and Its Application

A restructuring algorithm (as shown in figure 3) is thus implemented in OPAL to (semi)automatically apply the above rules to restructure BPEL models.

Input: P1;P2;P3; …;Pn;NULL which are stored in variable *str* (NULL represents the terminator)
Initial: ***cp*** and ***np*** denotes the *current pointer* and *new element pointer* initially to be 1 and 0, respectively (pointing to the position where activity P2 and P1 is located); ***str*** is a data structure to store the current sequential BPEL process structure; ***c*** is an upper bound for retry times; ***props*** is a set of pre-defined properties that the input BPEL process must satisfy.
Procedure: *Transp*(*np*, *cp*, *str*, *c*, *props*)
For (i=0; i < c; i++)
  repeat
    case: *np* and *cp* both point to basic activities
      if (satisfyRule1)
        apply Rule 1, update *str*;
        np points the new element; *cp* = *np* + 1;
      else *np* = *cp*; *cp* = *cp* + 1;
    case: *np* does not point to a basic activity AND *cp* < *np*
      if (satisfyRule3)
        apply Rule 3, update *str*; *np* points the new element;
        if (*np* != 1) *cp* = *np* - 1; **else** *cp* = *np* + 1;
      else *np* = *cp*; *cp* = *cp* + 1;
    case: *np* does not point to a basic activity AND *cp* > *np*
      if (satisfyRule2.2)
        apply Rule2.2, update *str*; *np* points the new element;
        call *Transp*(pointer right to *np*, pointer right to *np*+1, right structure of *str*, *c*, *props*);
        if (*np*!=1) *cp* = *np* - 1; **else** *cp* = *np* + 1;
      else if (satisfyRule2.1)
        apply Rule2.1, update *str*; *np* points the new element;
        call *Transp*(pointer left to *np*, pointer left to *np*+1, left structure *str*, *c*, *props*);
          if (np!=1) cp = np - 1; else cp = np + 1;
      else *np* = *cp*; *cp* = *cp* + 1;
  until (*cp*=NULL)
  if (*Check*(P1;P2;P3; …;Pn, *props*) = true)
    /*Model checking if the model still satisfies props
  Report Success; **Terminate**;
End For

Fig. 3. Restructuring Algorithm in OPAL.

The idea of the algorithm is concluded as follows. First the BPEL model is traversed by its basic activities. Each activity and its predecessors are matched with the previous rules for potential transformations. After the traverse is over, model checking is then applied to verify that the restructured model satisfies the pre-defined set of property specifications (if there is any). If the verification fails, the rule matching process is retried until a retry upper bound is reached.

To illustrate the application of the algorithm, let us revisit the BPEL model in 4.3. In this model, the core business operations of account opening and customer identity verification cannot start until the open account proposal is prepared. Meanwhile, this preparation can only start after the first flow structure is completed. This makes the *PrepareOpenAccProposal* activity a potential bottleneck in the process and lead to a low utility of resources (people who prepares the proposal).

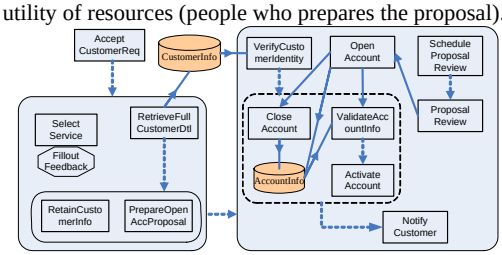

Fig. 4. Restructured Result with OPAL.

Figure 4 shows the restructure result of OPAL based on the algorithm, in which *PrepareOpenAccProposal* is now concurrently executed with *RetainCustomerInfo*. The result is not only state equivalent to the model in figure 2 (because *PrepareOpenAccProposal* has no incoming links and does not share variable with other activity), but also preserves the ***weak security protocol*** in section 4.3. However, a problem here is that updating a sequential process to a parallel one may not definitely lead to performance improvement since the execution logic of a business process is also determined by other factors like data dependency, policies, etc. Therefore, it must be clear that the result of the algorithm is only a guide for possible restructuring. Business consultants can manually adjust the restructured results based on the synergy of multiple factors.

6. Conclusion

The contribution of this paper includes three parts. First, the semantics of BPEL is fully formalized with Pi calculus. It is proved that algebraic laws and important properties abstracted from BPEL 1.1 specification are also well preserved in the formalization. Second, it is shown how model checking is applied for the automatic validation and verification of BPEL models with our OPAL toolkit. The verification result of a concrete BPEL model explains that the formalization of BPEL with Pi calculus has realistic values. Thirdly, based on the definition of state equivalence for Pi calculus, transformation rules are conducted to restructure a BPEL process such that the resulted process satisfies the same set of property specifications and fulfills the same business goals as the original one. But meanwhile it can potentially gain better performance. A restructure algorithm is implemented based on these rules in OPAL toolkit and tested on a concrete BPEL model. It illustrates that the combination of equivalence analysis of Pi calculus processes and the model checking is valuable in solving real business problems.

References

[1] T. Andrews, F. Curbera, et al. “Business Process Execution Language for Web Services 1.1”, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, 2003.

[2] W. L. Wang, Z. Hidvegi, et al. “E-process design and assurance using model checking, Computer”, 33(10), 2000, pp. 48-53.

[3] B. Anderson, J. Hansen, et al. “Model checking for design and assurance of e-Business processes”, Decision Support Systems, 39(3), 2005, pp. 333-344.

[4] E. M. Clarke, O, Jr. Grumberg, D. A. Peled. *Model Checking*. MIT Press, Cambridge, Mass, 1999.

[5] K. Xu, Y. Liu, et al. “BPSL Modeler - Visual notation language for intuitive business property reasoning”. Proc the 5th International Workshop on Graph Transformation and Visual Modeling Techniques, 2006, In Press.

[6] R. Milner. *Communicating and Mobile Systems: the Pi calculus*. Cambridge University Press, Cambridge, 1999.

[7] W.M.P Aalst. “Pi calculus versus Petri nets: Let us eat humble pie” rather than further inflate the Pi hype”, Technical Report, Eindhoven University of Technology, 2004. URL

[8] F. Roozbeh, G. Uwe, et al. “Abstract Operational Semantics of the Business Process Execution Language for Web Services”. Technical Report CMPT2004-03, Simon Fraser University, 2003.

[9] W. Andreas, F. Peter, et al. “Transforming BPEL into annotated deterministic finite state automata for service discovery”. Proc. IEEE International Conference on Web Services, 2004, pp.316-323.

[10] R. Farahbod, U. Glässer et al. “Specification and validation of the business process execution language for web services”. Lecture Notes in Computer Science, 3052, 2004, pp. 78-94.

[11] S. Karsten, S. Christian. “A Petri net semantic for BPEL4WS-validation and application”, Proc. 11th Workshop on Algorithms and Tools for Petri Nets, 2004, pp. 1-6.

[12] M. Lumpe, F. Achermann, et al. “A formal language for composition”. In Foundations of Component-based Systems, Cambridge University press, 2000, pp. 69-90.

[13] B. Victor. “A verification tool for the polyadic Pi calculus”. Ph.D. Thesis, Uppsala University, Sweden, 1994.

[14] G.L. Ferrari, S. Gnesi, et al. “A Model Checking Verification Environment for Mobile Processes”. ACM Trans. Software Engineering and Methodology, 12(4), 2003, pp. 440-473.

[15] M. Dam. “Model checking mobile processes”. Proc. 4th International Conference on Concurrency Theory, 1993, pp. 22-36

[16] BPEL Designer,

[17] A. Cimatti, E. Clarke, et al. “NuSMV 2: an OpenSource tool for symbolic model checking”. Lecture Notes in Computer Science, 2404, 2002, pp. 359-364.

[18] K. Xu, L. C. Liu, C. Wu. “Well-timed Pi Calculus and Its Approach to Equivalency and Schedulability Analysis”. Computer Integrated Manufacturing System, 2006, In Press

[19] K. Xu, Y. Liu, et al. “BPSL Modeler - Visual notation language for intuitive business property reasoning”. Technical Report, IBM, RC23830(C0512-005), 2005.

Comment:

Comment: