A static compliance-checking framework for business process models

Y. Liu
S. Müller
K. Xu

Regulatory compliance of business operations is a critical problem for enterprises. As enterprises increasingly use business process management systems to automate their business processes, technologies to automatically check the compliance of process models against compliance rules are becoming important. In this paper, we present a method to improve the reliability and minimize the risk of failure of business process management systems from a compliance perspective. The proposed method allows separate modeling of both process models and compliance concerns. Business process models expressed in the Business Process Execution Language are transformed into pi-calculus and then into finite state machines. Compliance rules captured in the graphical Business Property Specification Language are translated into linear temporal logic. Thus, process models can be verified against these compliance rules by means of model-checking technology. The benefit of our method is threefold: Through the automated verification of a large set of business process models, our approach increases deployment efficiency and lowers the risk of installing noncompliant processes; it reduces the cost associated with inspecting business process models for compliance; and compliance checking may ensure compliance of new process models before their execution and thereby increase the reliability of business operations in general.

INTRODUCTION

Modern businesses face a broad number of challenges. While striving to please their customers, they must meet the expectations of their shareholders and remain profitable. Due to globalization and digitization, they are typically confronted with increased and highly dynamic competition. Consequently, investments in information technology (IT) have become a necessary condition to stay competitive and remain in business. As a result, many enterprises have recently shown a growing interest in business process management (BPM), which

refers to all activities performed by businesses to model, automate, optimize, monitor, and adopt their businesses processes.[1,2] Thus, there has been an increased acceptance and adoption of business process management systems (BPMSs) in order to efficiently support, execute, and monitor business processes.

The above development has been paralleled by a growing number of regulatory requirements imposed on businesses. Prominent examples in the United States are the Gramm-Leach-Bliley Act,[3] the Sarbanes Oxley Act (SOX),[4] and the USA Patriot Act.[5] While these United States regulations have very broad coverage, many industry-specific regulations—such as the international Basel II accord,[6] European Money Laundering Regulation,[7] and the Law of the People's Republic of China on the People's Bank of China[8]—have been enacted around the globe. Demonstrating compliance with legal requirements and international standards generally requires that affected companies document their business processes. Although many enterprises try to regard such documentation requirements as an opportunity to identify their informal processes and to render their execution more efficient, for large enterprises with thousands of different business processes, this alone represents a considerable challenge.

Enterprises operating in heavily regulated industries, such as financial services, health care, government, and national defense, are likely governed by a large number of regulatory requirements. As these requirements must be implemented and enforced by a multitude of internal business and IT controls, many regulations now recommend the use of respected standards, such as COBIT** (Control Objectives for Information and Related Technologies)[9] and ITIL** (Information Technology Information Library),[10] for the implementation of an enterprise IT system. These standards consist of well-defined abstract process definitions that can be tailored according to a company's individual needs.

Because of the increasing number of regulations and standards, enterprises need a comprehensive compliance-management approach, as discussed in Abrams et al.[11] and Giblin et al.[12] They need to be able to understand the implications of new regulations for their business and its processes. As business processes are increasingly managed using BPMSs, regulatory requirements that necessitate changes to the structure of particular workflows directly impact business process modeling. Thus, whenever a new regulatory requirement is enacted, a company needs to know what its impact is. Three effects are possible: existing business processes must be adapted or removed; new business processes must be introduced; or there is no impact because all business processes are already compliant with the new requirement.

Business processes that are automated through BPMS can be used to implement IT processes and controls, as defined by ITIL or COBIT, and thereby address existing regulations. The impact of new regulatory requirements, however, cannot be assessed using these frameworks. For large enterprises with thousands of business processes deployed on the BPMS and stored in specific repositories, the assessment of which existing process definitions comply with a new regulatory requirement is of utmost importance. In this paper, we describe an approach that allows for the static verification of business process models against a set of formally expressed regulatory requirements, which include constraints on the state and execution order of process activities. We call these formally expressed regulatory requirements *compliance rules*. Our approach helps a company with the identification of noncompliant business processes before their execution and, in the case of noncompliance, indicates the nature of the problem.

Potential benefits of automatic verification of business process models

To ensure compliance, the impact of each new regulatory requirement on existing business process models needs to be identified. Although BPM does not help here, our approach indicates which processes are compliant and which are not, hence providing a valuable tool to ensure that new requirements are incorporated into the company's process models. Hence, the benefit of our method is threefold:

1. Through automated verification of a large set of business process models, our approach increases efficiency during deployment and lowers the risk of implementing and activating noncompliant processes.

2. By automating a tedious task that must otherwise be done manually, our method reduces the cost associated with inspecting business process models for compliance.
3. Used as a tool during modeling of new business processes, our approach ensures compliance of new models before their execution and thereby increases the reliability of business operations in general.

Case study

We now introduce a case study, which we shall use as a running example throughout this paper. We assume the existence of a Chinese bank called *SimpleBank*, whose managers want to know whether its business operations conform to a set of relevant compliance rules. For the sake of simplicity, we focus only on SimpleBank's account-opening process, whose process definition is portrayed in *Figure 1*.

We further assume that at some point SimpleBank is confronted with a set of new compliance rules corresponding to the Rules for Anti-Money Laundering by Financial Institutions,[13] as published by the People's Bank of China. The two relevant articles are Article 11 and Article 13. Article 11 states in part: "When opening deposit accounts or providing settlement service for individual customers, financial institutions shall verify the customers' IDs and record the names and ID numbers." Article 13 states: "Financial institutions shall abide by relevant rules and report to the People's Bank of China and/or the State Administration of Foreign Exchange of any large-value transactions detected in the process of providing financial services to customers.

> Classification of large-value transactions shall be determined in line with relevant rules made by the People's Bank of China and the State Administration of Foreign Exchange on reporting of fund transactions.

For example, if a customer deposits a large amount of money into his account, the respective transaction must be reported. Compliance with this rule requires an adequate interpretation of *large value* according to the relevant rules made by the People's Bank of China and the State Administration of Foreign Exchange. For the sake of simplicity, we shall assume this is a parameter that can be flexibly adjusted.

Given the business process in Figure 1 and a set of compliance rules, we demonstrate in this paper how a set of well-defined model transformations enables the use of model-checking technology to verify whether the definition of a business process complies with a set of relevant compliance rules. We call our method *compliance checking*.

Overview of the compliance-checking method

Overall, our compliance-checking method includes six major steps (*Figure 2*).

In Step 1, we model our business processes using BPEL (Business Process Execution Language).[14] In Step 2, we use the visual BPSL[15] to specify relevant compliance rules. (In the section "Modeling business processes and compliance rules," we provide both the BPEL process model and the compliance rules formalized in BPSL (Business Property Specification Language) for the SimpleBank case study.) We transform the BPEL process model into a representation using pi-calculus[16] in Step 3.1. Then, in Step 3.2, the pi-calculus is transformed into a finite state machine (FSM). In Step 4, the BPSL compliance rules are transformed into linear temporal logic (LTL).[17] (Steps 3.1, 3.2, and 4 are described in the section "Model transformations.") Having thus formalized both the business processes and compliance rules, in Step 5 we use model-checking technology[18] to statically verify whether the business processes comply with the imposed regulations. In Step 6, counterexamples (i.e., execution orders of process activities that demonstrate how the compliance rules can be violated) are fed back to the business process layer to demonstrate how the compliance rules have been violated. (Details about the model checking, counterexample tracing, and specific optimization approaches for compliance checking are presented in the sections "Compliance-checking framework and case study results" and "Advanced features of the compliance-checking framework.")

MODELING BUSINESS PROCESSES AND COMPLIANCE RULES

Before introducing the details of our compliance-checking method, we briefly explain how to model business processes with BPEL and how to specify compliance rules with BPSL.

Business process modeling using BPEL

In the earlier section "Case study," we introduced a conceptual account-opening process for Simple-

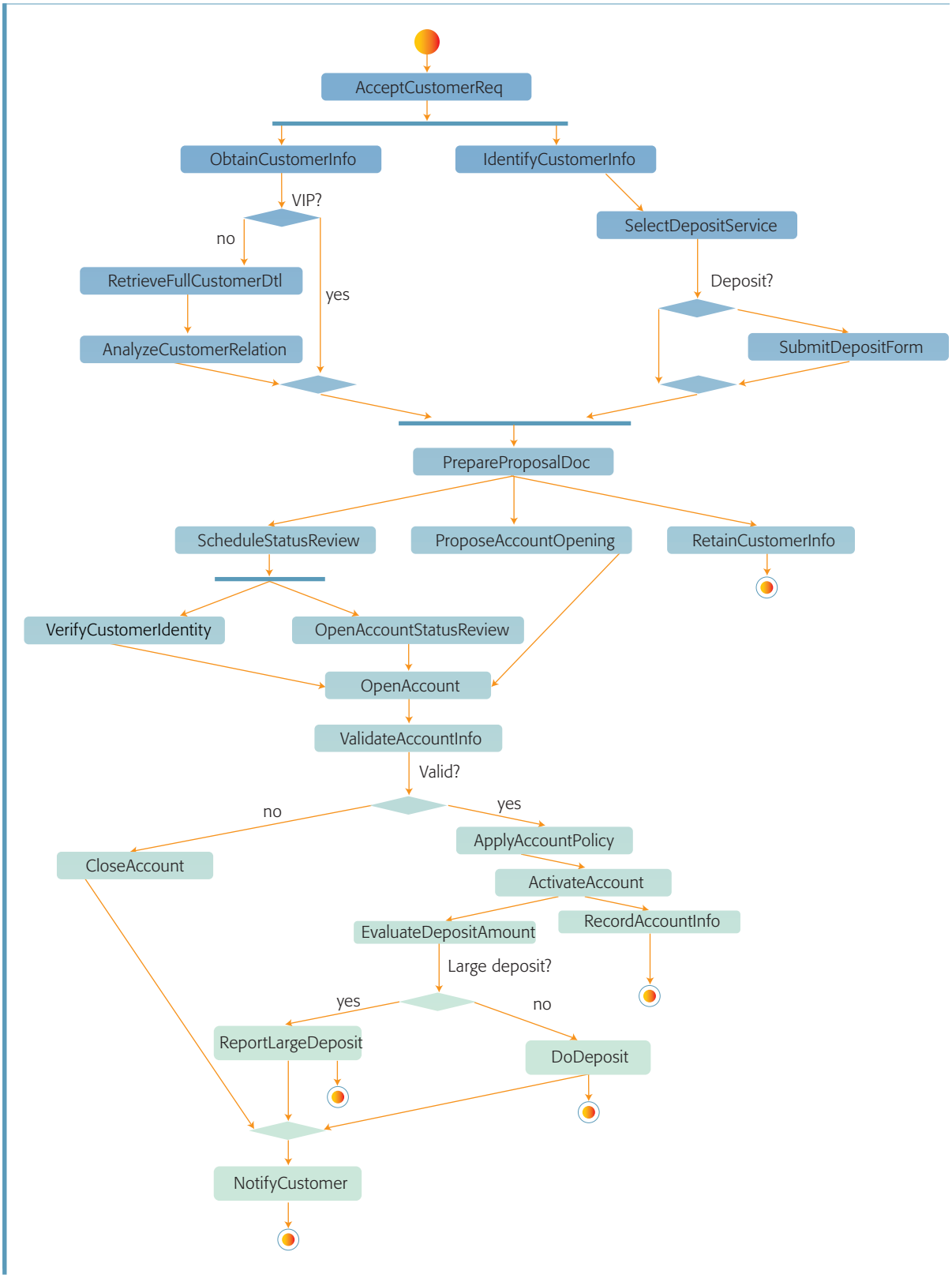

Figure 1
Account-opening process

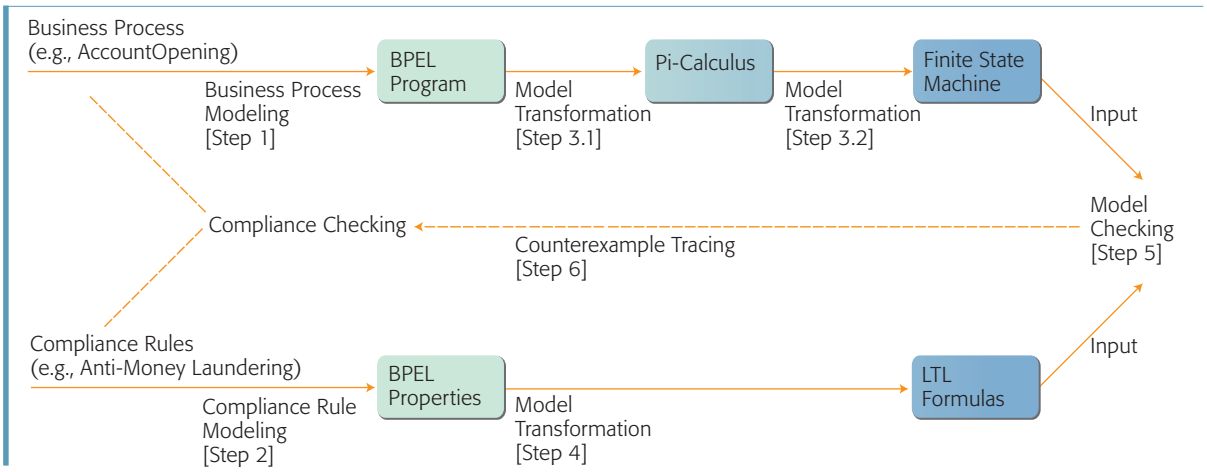

Figure 2
Compliance-checking method

Bank. Assuming that SimpleBank wants to take advantage of a BPMS to manage this process, the process should be specified with an executable business process modeling language. BPEL is such a language. It is a de facto standard for business application integration and business-to-business processing based on XML (Extensible Markup Language), with a specific focus on Web services. It synthesizes essential aspects of Web Services Flow Language (e.g., support for graph-oriented processes)[19] and XLANG[20] (e.g., structural constructs for processes) into one cohesive language to support implementing business processes in a natural manner. Although there is no formal proof that BPEL is powerful enough to express all requirements related to business processes, BPEL has been applied in many real customer cases and enjoys broad industry acceptance. Because of this and its characteristic features, BPEL has been the business process modeling language of choice in our compliance-checking method.

A BPEL process, also called a *BPEL program*, consists of four major elements: The *Variable* section defines the data variables used by the process, providing their definitions in terms of Web Services Description Language (WSDL) message types, XML schema simple types, or XML schema elements. The *PartnerLinks* section defines the different parties that interact with the business. The *FaultHandler* elements define the activities that must be performed in response to faults during process execution. The rest of the process definition contains the description of normal behavior with BPEL activities, including *BasicActivity*, *StructuredActivity*, and *ScopeActivity*. The BPEL program corresponding to the account-opening process is given in ***Figure 3***. (It shows only the behavior definition section, whose essence is self-explanatory.) The precise semantics of BPEL activities and variables are explained in the section "Model transformations."

As writing BPEL code manually is relatively cumbersome, process designers often model business processes visually by using a process modeling tool such as IBM WebSphere* Business Integration Modeler (WBI Modeler).[21] In addition to providing a visual user interface, these tools allow graphical process models to be exported as BPEL programs. We do not focus on the transformation from Unified Modeling Language (UML**) to BPEL here; refer to Mantell[22] for more detailed information. (The visual process modeling language provided by WBI Modeler is partially compatible with UML activity-diagram notation.)

Compliance rule modeling using BPSL

Temporal constraints in compliance rules can be specified formally with temporal logic formulas, such as LTL and computation tree logic (CTL). However, for people without a background in formal logic systems, temporal logic systems are rather difficult to understand and use. The purpose of BPSL is to provide a more intuitive formalism to

```
<bpws:sequence name="Sequence" ……>
    <bpws:receive createInstance="yes" name="AcceptCustomerReq"
                  partnerLink="ACQ" variable="variable_acq" ……/>
    <bpws:flow name="Flow" ……>
      <bpws:scope name="Scope" ……>
        <bpws:sequence name="HiddenSequence" ……>
          <bpws:invoke inputVariable="variable_oci"
            name="ObtainCustomerInfo" outputVariable="variable_oci" p……/>
          <bpws:switch name="VIP?" ……>
            <bpws:case ……>
              <bpws:empty name="EmptyAction2" ……/>
            </bpws:case>
            <bpws:otherwise>
              <bpws:sequence name="HiddenSequence1" ……>
                <bpws:invoke inputVariable="variable_rfcd"
         name="RetrieveFullCustomreDtl" outputVariable="variable_rfcd" ……/>
                  <bpws:invoke inputVariable="variable_acr"
                   name="AnalyzeCustomerRelation" outputVariable="variable_acr"
                       partnerLink="ACR" ……/>
              </bpws:sequence>
            </bpws:otherwise>
          </bpws:switch>
        </bpws:sequence>
      </bpws:scope>
      <bpws:sequence name="Sequence1" wpc:displayName="Sequence1" ……>
        <bpws:invoke inputVariable="variable_acr" name="IdentifyCustomerReq"
                     outputVariable="variable_acr" partnerLink="ICR" ……/>
        <bpws:invoke inputVariable="variable_sds" name="SelectDepositService"
                     outputVariable="variable_sds" partnerLink="SDS" ……/>
        <bpws:switch name="Deposit?" ……>
          <bpws:case ……>
            <bpws:invoke inputVariable="variable_sdf" name="SubmitDepositForm"
                         outputVariable="variable_sdf" partnerLink="SDF" ……/>
          </bpws:case>
          <bpws:otherwise>
            <bpws:empty name="EmptyAction" ……/>
          </bpws:otherwise>
        </bpws:switch>
      </bpws:sequence>
    </bpws:flow>
    <bpws:invoke inputVariable="variable_ppd" name="PrepareProposalDoc"
                 outputVariable="variable_ppd" partnerLink="PPD" ……/>
    <bpws:flow name="Flow1" ……>
      <bpws:links>
        <bpws:link name="Link2" ……/>
        <bpws:link name="Link3" ……/>
      </bpws:links>
      <bpws:invoke inputVariable="variable_vci" name="VerifyCustomerIdentity"
                   outputVariable="variable_vci" partnerLink="VCI" ……>
        <bpws:targets> <bpws:target linkName="Link3"/> </bpws:targets>
      </bpws:invoke>
      <bpws:invoke inputVariable="variable_ssr" name="ScheduleStatusReview"
                   outputVariable="variable_ssr" partnerLink="SSR" ……>
        <bpws:sources>
          <bpws:source linkName="Link2"/>
          <bpws:source linkName="Link3"/>
        </bpws:sources>
      </bpws:invoke>
```

Figure 3 (Part 1 of 2)
BPEL program for the account-opening process

```
<bpws:invoke inputVariable="variable_pao" name="ProposeAccountOpenning"
                  outputVariable="variable_pao" partnerLink="PAO" ……/>
      <bpws:invoke inputVariable="variable_rci" name="RecordCustomerInfo"
                  outputVariable="variable_rci" partnerLink="RCI" ……/>
      <bpws:invoke inputVariable="variable_oasr" name="OpenAccountStatusReview"
                  outputVariable="variable_oasr" partnerLink="OASR" ……>
        <bpws:targets> <bpws:target linkName="Link2"/> </bpws:targets>
      </bpws:invoke>
    </bpws:flow>
    <bpws:invoke inputVariable="variable_oa" name="OpenAccount"
                outputVariable="variable_oa" partnerLink="OA" ……/>
    <bpws:invoke inputVariable="variable_vai" name="ValidateAccountInfo"
                outputVariable="variable_vai" partnerLink="VAI" ……/>
    <bpws:switch name="Valid?" ……>
      <bpws:case ……>
        <bpws:sequence name="HiddenSequence3" ……>
          <bpws:invoke inputVariable="variable_aap" name="ApplyAccountPolicy"
                       outputVariable="variable_aap" partnerLink="AAP" ……/>
          <bpws:invoke inputVariable="variable_aa" name="ActivateAccount"
                       outputVariable="variable_aa" partnerLink="AA" ……/>
          <bpws:flow name="ParallelActivities" ……>
            <bpws:links>
              <bpws:link name="Link1" ……/>
              <bpws:link name="Link4" ……/>
            </bpws:links>
            <bpws:invoke inputVariable="variable_rai" name="RecordAccountInfo"
                         outputVariable="variable_rai" partnerLink="RAI" ……/>
            <bpws:invoke name="EvaluateDepositAmount" ……>
              <bpws:sources> <bpws:source linkName="Link1"/> </bpws:sources>
            </bpws:invoke>
            <bpws:switch name="LargeDeposit?" ……>
              <bpws:targets> <bpws:target linkName="Link1"/> </bpws:targets>
              <bpws:sources> <bpws:source linkName="Link4"/> </bpws:sources>
              <bpws:case wpc:id="66">
                <bpws:invoke inputVariable="variable_rld" name="ReportLargeDeposit"
                             outputVariable="variable_rld" partnerLink="RLD" ……/>
              </bpws:case>
              <bpws:otherwise>
                <bpws:invoke inputVariable="variable_dd" name="DoDeposit"
                             outputVariable="variable_dd" partnerLink="DD" ……/>
              </bpws:otherwise>
            </bpws:switch>
            <bpws:reply name="NotifyCustomer1" partnerLink="NC" variable="variable_nc" ……>
              <bpws:targets> <bpws:target linkName="Link4"/> </bpws:targets>
            </bpws:reply>
          </bpws:flow>
        </bpws:sequence>
      </bpws:case>
      <bpws:otherwise>
        <bpws:sequence name="HiddenSequence2" ……>
          <bpws:invoke inputVariable="variable_ca" name="CloseAccount"
                       outputVariable="variable_ca" partnerLink="CA" ……/>
          <bpws:reply name="NotifyCustomer" partnerLink="NC"
                      portType="wsdl:ProcessPortType" variable="variable_nc" ……/>
        </bpws:sequence>
      </bpws:otherwise>
    </bpws:switch>
</bpws:sequence>
```

Figure 3 (Part 2 of 2)
BPEL program for the account-opening process

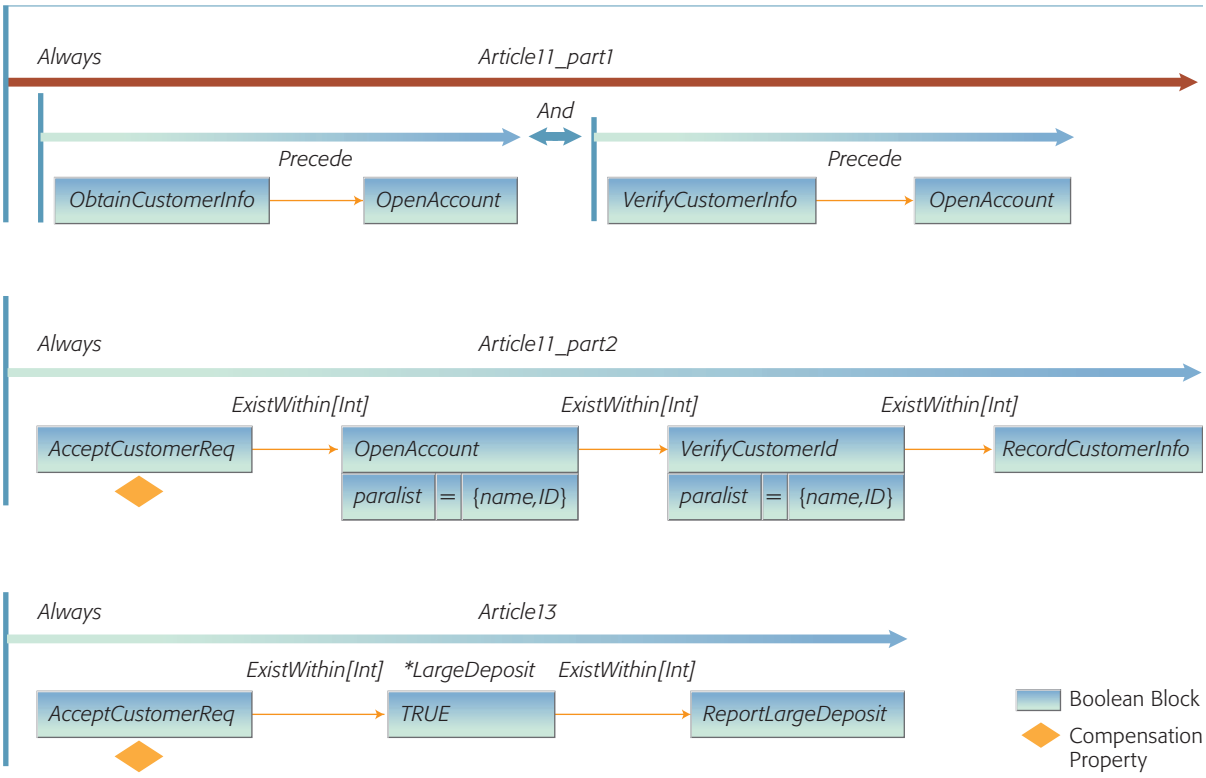

Figure 4
BPSL specification of Article 11 and Article 13

express such properties. Therefore, in our compliance-checking method, we use BPSL for specifying compliance rules.

Four main features of BPSL simplify the specification and understanding of temporal properties:

1. Obscure logical operators are replaced with an intuitive visual notation.
2. Recurring logical patterns from a business or regulatory domain are defined as dedicated operators.
3. Domain-specific templates (e.g., the property patterns[23]) can be predefined and reused in BPSL to help increase the efficiency of property specification.
4. BPSL has a direct semantic interpretation in both LTL and CTL.

In this paper, we focus more on LTL because we agree with the argument by Vardi[24] that the branching-time formalism of CTL is unintuitive to business analysts and does not support compositional reasoning, as does LTL. The complete syntax, semantics, and notation of BPSL have been described by Xu et al.[15] To ease understanding, in ***Figure 4*** we present the visual BPSL specification of Article 11 and Article 13 of our case study. (The precise semantics of these BPSL properties are explained in the next section.)

In Figure 4, Article 11 is specified by the first two BPSL properties, *Article11_part1* and *Article11_part2*, while Article 13 is specified by the third property, *Article13*. The rectangles denote Boolean blocks, which may represent the performing of business activities (e.g., *AcceptCustomerReq*, *ObtainCustomerInfo*, and *VerifyCustomerInfo*) or the processing of data (e.g., *names* and *IDs* in the *ParaList*). Annotated arrows are used as temporal operators to define the temporal dependency between the Boolean blocks. For example, the temporal operator *ExistWithin [inf]* specifies that the next Boolean block must hold within an infinite amount of time after the previous Boolean block holds. The combination of these temporal dependencies thus forms a visual temporal sequence in BPSL. There are simple and compound temporal

sequences. A simple temporal sequence specifies a chain of temporal relations among different Boolean blocks or any other BPSL properties (e.g., *Article11_part2* and *Article13* are simple ones). A compound temporal sequence is a logical combination of simple temporal sequences (e.g., *Article11_part1* is a compound temporal sequence that consists of two other simple temporal sequences combined by a logical *AND* relation).

The BPSL properties in Figure 4 thus possess the following semantics: Before a customer account may be opened, the customer information must first be obtained and verified (*Article11_part1*): Whenever a customer request for opening a deposit account is received, the customer information must be obtained and the customer name and ID must be later verified. Finally, the customer information must be recorded (*Article11_part2*). Whenever a customer request for opening a deposit account is received, the deposit must be checked as to its amount. If it is a large-value deposit, it must be reported later.

In previous work, Giblin et al.[12] developed REALM (regulations expressed as logical models), a metamodel and method to formally specify regulations. While REALM, which builds on real-time temporal object logic, is more expressive than BPSL, BPSL supports visual, and thus more intuitive, property specification. As REALM properties cannot be verified by existing model-checking algorithms, we use BPSL as the specification language in our static compliance-checking work.

MODEL TRANSFORMATIONS

In this section we introduce the essential model transformations of our compliance-checking method. To reuse existing model-checking algorithms, a series of transformations are performed. We first explain the transformation from BPEL to pi-calculus and the transformation from pi-calculus to FSM (see Steps 3.1 and 3.2 in Figure 2). Then we show how BPSL properties are transformed into LTL formulas (see Step 4 in Figure 2).

BPEL to pi-calculus

Some elementary knowledge of pi-calculus is necessary to understand the content of this section. Before introducing the transformation from BPEL to pi-calculus, we thus present an introduction to pi-calculus.

Pi-calculus

Pi-calculus[16] is a model of concurrent communicating processes that allows complex communication patterns to be modeled. We chose pi-calculus as the formal method to formalize BPEL programs. A detailed discussion justifying this choice can be found in the section "Related work."

The syntax of Milner's polyadic pi-calculus is as follows:

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

$$\pi_i ::= \overline{x} < y > | x(y) | \tau$$

$$\phi ::= [x = y] | \phi \wedge \phi | \neg\phi$$

The simplest entities of pi-calculus are *names* (denoted by lowercase) and *processes* (denoted by uppercase). Processes can evolve by performing *actions*. Syntactically, $\overline{x} < y >$ denotes an *output* action that sends name y via x, and $x(y)$ is an *input* action, which receives a name y via x. Further, τ is a silent *action* that expresses unobservable behavior. A *sum* $\pi_1.P_1 + \pi_2.P_2 + \cdots + \pi_n.P_n$ denotes a nondeterministic choice of process execution. In the *restriction new x P*, the scope of name x is bound to P. In the *composition* $P|Q$, the processes P and Q can proceed independently and can interact by means of shared names. The *replication* $!P$ can be thought of as an infinite composition $P|P|\cdots$ of processes. Finally, ϕP represents a process that is guarded by a Boolean expression ϕ evaluated by name *matching*.

Transforming BPEL to pi-calculus

The transformation from BPEL to pi-calculus requires a semantic translation of BPEL. The overall semantics are very complex. Having described the complete transformation from BPEL to pi-calculus in our previous work,[25] in this paper we present only the important basic and structured activities.

BPEL contains program variables to which values can be assigned. We adopt the approach proposed by Jacobs and Piessens[26] to formally define a programming variable as a *storage location*. Accordingly, a variable holding a value of x (*Variable(x)*) is defined by a register (Reg) as follows (the expression $=_{def}$ is used as a definition symbol):

$$Variable(x) =_{def} Reg(x)$$

$$Reg(x) =_{def} put(y).Reg(y) + \overline{get} < x > .Reg(x)$$

The above formalization means that the stored value

x of the variable can be read from the storage location by using $\overline{get}<x>$, and a new value y can be written into the location with $put(y)$.

The basic BPEL activities (*Receive, Reply, Invoke, Assign, Empty,* and *Termination*) define how message communication, service invocation, and variable assignment are done in a process model. *Link name, Partner name,* and *Operation name* are three elements in the activities of BPEL, and they often appear at the same time. Therefore, here the three elements are denoted as a unified name: ℓ. The partner links and data sharing in these BPEL activities are mapped to the input and output prefixes of ℓ, *get*, and *put* in pi-calculus. In addition, two special names, *start* and *done*, are used to indicate common internal communication in a BPEL process.

$$Receive(start, \ell, put, done) =_{def}$$
$$start.\ell(v).\overline{put} < v >. \overline{done}$$
$$Reply(start, get, \ell, done) =_{def}$$
$$start.get(v).\overline{\ell} < v >. \overline{done}$$
$$Invoke(start, get, \ell, put, done) =_{def}$$
$$start.get(v).\overline{\ell} < v >.\ell(w)\overline{put} < w >. \overline{done}$$
$$Assign(start, get, put, done) =_{def}$$
$$new\ c(start.get(v).\overline{c} < v > | c(x).\overline{put} < x >. \overline{done})$$
$$Empty(start, done) =_{def}\ start.\overline{done}$$

There are some challenges in the transformation, such as the handling of *timeout, synchronizing with links, message correlation, global termination of activity*, and *fault handling and compensation*. We discussed how to solve these problems in another paper.[25] Because they are not critical to understanding our compliance-checking method, we do not cover them in this paper. We take *Receive*(*start, ℓ,put,done*) as an example to explain the semantics of the pi-calculus formalization. The process of *Receive* contains some free names (e.g., *put* and *done*), which are defined as the communication channels of this process. The names *start* and *done* are used to start and terminate the activity. The communication among different activities through these two channels thus forms the control flow of the BPEL process. When the *Receive* process is triggered by its start channel, an input action of $\ell(v)$ is enacted to receive a message through a specific partner link. The output action of $\overline{put}<v>$ is then enacted to put the received message (i.e., v) into the corresponding variable. Finally, the action $\overline{done}$ is enacted to indicate the termination of the *Receive* activity and to trigger another activity in the BPEL process.

Structured activities imply different control relations between the executions of activities in BPEL. We let function *fn* define a mapping from a pi-calculus process to a set of free names, where *fn(P)* indicates the set of all free names contained in process *P*. Consequently, the semantics of the structured activities *Sequence, Switch, While, Pick*, and *Flow* are defined as the following:

$$Sequence(fn(P), fn(Q)) =_{def}$$
$$new\ start(\{start/done\}P|Q)$$
$$Switch(b_1, fn(P), b_2, fn(Q)) =_{def}$$
$$[b_1]P + [\neg b_1 \wedge b_2]Q + [\neg b_1 \wedge \neg b_2]Empty$$
$$While\ (b, fn(P)) =_{def}$$
$$[b]Sequence(fn(P), fn(While)) + [\neg b]Empty$$
$$Pick(\ell, fn(P_1), \ell', fn(P_2), put) =_{def}$$
$$(new\ c(\ell(v).\overline{put} < v >. \overline{c}|c.P_1)) +$$
$$(new\ c(\ell'(v').\overline{put} < v' >. \overline{c}|c.P_2))$$
$$Flow(fn(P), fn(Q), done) =_{def}$$
$$new\ ack((new\ done'(\{done'/done\}P|$$
$$done'.\overline{ack}))|$$
$$(new\ done''(\{done''/done\}Q|$$
$$done''.\overline{ack}))|ack.ack.\overline{done})$$

In the formalization, $\{start/done\}$ is a basic name substitution operation, which means that the name *done* is replaced by *start* so that an internal interaction can occur between processes *P* and *Q*. The formalization of *Switch* above implies that in the case when several branching conditions hold at the same time, the branches are taken in the order in which they appear, which reflects exactly the semantics of BPEL in its specification. *Pick* is a nondeterministic choice between process *P* and *Q*. *Switch, While, Sequence,* and *Flow* are self-explanatory.

Scope is an important concept in BPEL for defining an effective scope of the definition and use of variables, compensation handlers, fault handlers, and other activities. As these BPEL elements can all be associated with a scope, we collect all the free names in a scope with a predefined function,

GetNames(s), where *s* is the scope. Therefore, the restriction operator *new* can be used to restrict the access to these elements according to their effective scope. The following is the formalization of *Scope*:

$$Scope(\overline{restnames}) =_{def}$$
$$new\, GetNames(s)\,(P_1|P_2|\cdots|P_n)$$

P_i $(i=1,\cdots,n)$ can be a pi-calculus process of basic activities, structured activities, variables, fault handlers, or compensation handlers. Furthermore, $\overline{restnames}$ is defined as a free name set:

$$(fn(P_1)\cup fn(P_2)\cup\cdots\cup fn(P_n))/GetNames(s)$$

Because everything is a process in pi-calculus, the semantics of a BPEL process can be formalized as the *composition* of the above pi processes for all considered BPEL activities:

$$Process = Variable_1|\cdots|Variable_m|Activity_1|\cdots|Activity_n$$

Pi-calculus to FSM

The application of model checking to verify business process models against compliance rules requires the transformation of business process models into a formalism that can be accepted by a model-checking algorithm. Most current model-checking tools require FSM as an input format. We thus first translate from BPEL to pi-calculus and then from pi-calculus to FSM. Having the processes formalized in pi-calculus as an intermediary formalism provides us the opportunity to apply other verification techniques, such as structural verification (including, for example, deadlock detection) and bisimulation. Furthermore, it also helps the future integration of more practical and efficient model-checking algorithms and tools into our compliance-checking approach. Despite the detour through pi-calculus, the static verification result is the same as if the process models had been transformed into FSM directly.

The transformation from pi-calculus to FSM yields the total behavior of the BPEL process by reducing the pi processes into their corresponding state spaces. Previous works by Ferrari et al.[27] and Pistore[28] have already shown the approach and feasibility of how to correctly transform finitary pi-calculus processes into the corresponding finite state automata such that a wide range of powerful formal verification techniques can be smoothly reused in the case of mobile processes. In our compliance-checking method, we exploit the results obtained by Ferrari et al.[27] to transform a pi process into a corresponding FSM, based on the early operational semantics of pi-calculus. The mapping between the early operational semantics[29] of pi-calculus to the state transitions in FSM is presented as a set of transformation rules. For better understanding, each transformation rule is illustrated with an example (see ***Figure 5***).

For each transition triggered by an action π in pi-calculus, three attributes are used to record the necessary information of π. The *action* attribute records its port name *port(π)*; the *paralist* attribute records the set of parameters *para(π)* passed through the port; and the *type* attribute is one of the values of *input, output,* or *tau,* which indicates that π is an input, output, or an invisible action in pi-calculus.

The choice of pi-calculus as the mediation between high-level business process models and very formal models yields several benefits. Specifically, state space generation from pi processes and not directly from the business process model provides us with the following advantages:

- While transforming from the pi process to FSM, two properties can be directly checked: deadlock and redundant activities of the BPEL process. A deadlock exists if there is a pi process that cannot generate its state space before it changes to an empty process ("0"). If a pi process cannot communicate with any other pi process, then there is a redundant activity in the corresponding BPEL process.
- Using pi-calculus and FSM as the formal models of business processes renders the business process verification method independent of a specific model checker. Thus, more practical and efficient model checkers can be integrated into our compliance-checking method.

BPSL to LTL

Though BPSL supports semantic mapping to both LTL and CTL, in this section we explain the mapping from BPSL specifications to LTL formulas based on the three compliance rules given earlier in the section "Compliance rule modeling using BPSL" as examples. Before discussing the mapping from BPSL to LTL, an overview of LTL is given as preliminary information.

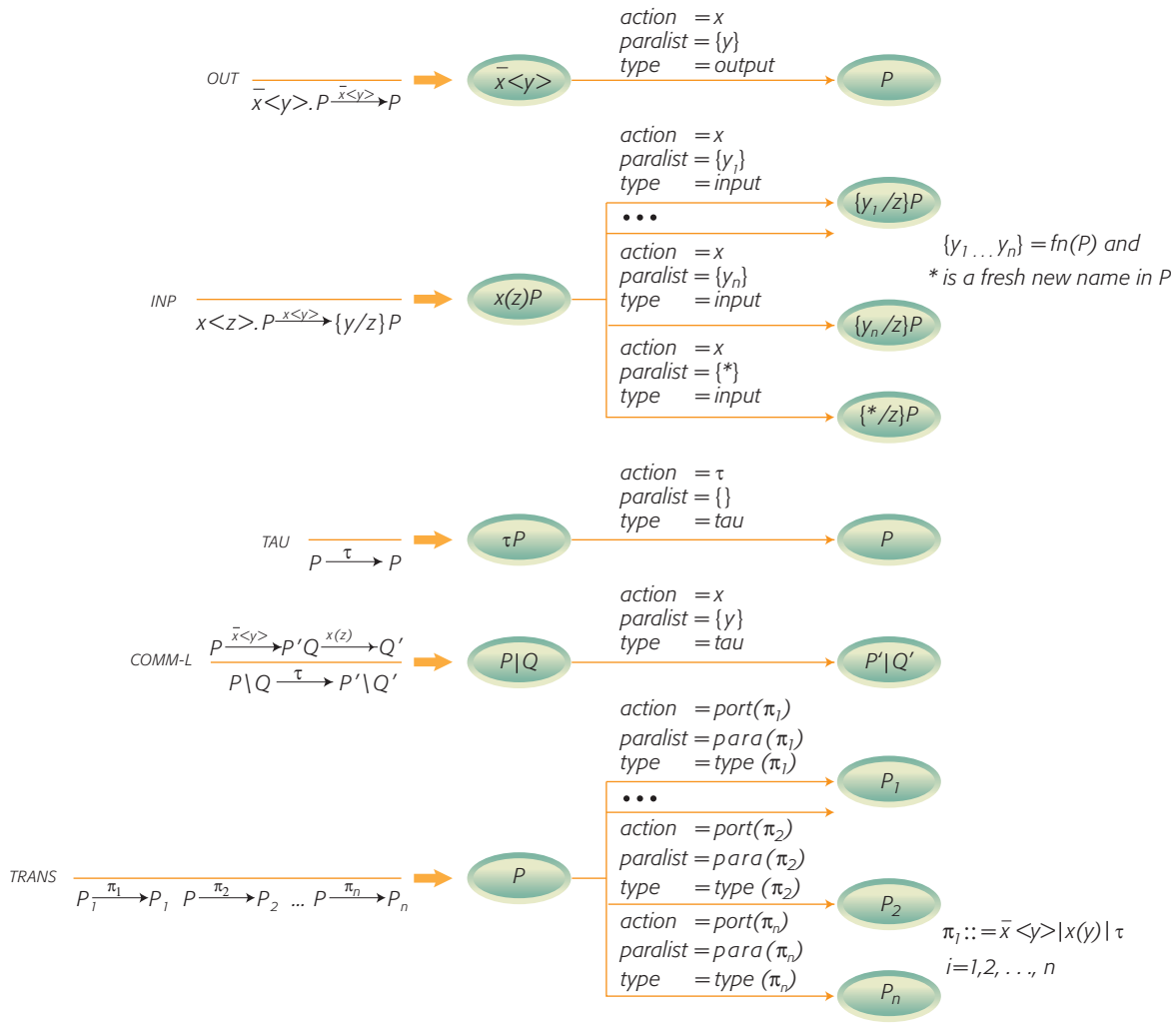

Figure 5
Transformation rules from pi-calculus to FSMs

LTL

LTL[17,18] is a widely used language for specifying temporal properties of software or hardware designs. In LTL, time is treated as if each moment has just one possible future. Accordingly, linear temporal formulas are interpreted over linear state sequences, and we regard them as describing the behavior of a single computation of a system.

LTL uses formulas of the form **A***f*, where: first, **A** is the universal path quantifier, which means that *f* has to be true on all possible paths in the future; and second, *f* is a path formula (which is true along a specific path) in which the only state subformulas permitted are atomic propositions. More precisely:

If $p \in AP$ (*AP* is a nonempty set of atomic propositions), then *p* is a path formula.

If *f* and *g* are path formulas, then $\neg f$, $f \vee g$, $f \wedge g$, **X***f*, **F***f*, **G***f*, and *f* **U** *g* are path formulas.

The four basic operators of *X*, *F*, *G*, and *U* are explained informally as follows:

- *X* ("next time"): a formula (*f*) is true in the second state of the path.
- *F* ("eventually"): a formula (*f*) will be true at some future state on the path.
- *G* ("always"): a formula (*f*) is true at every state on the path.

- *U* ("until"): there is a state on the path where the second formula (*g*) is true, and at every successive state on the path the first formula (*f*) is true.

Take the formula *G* (*AcceptCustomerReq* → *F VerifyCustomerId*) as an example. This formula is true in a computation precisely if every state in the computation in which *AcceptCustomerReq* holds is followed by some state in the future in which *VerifyCustomerId* holds. Here, *AcceptCustomerReq* and *VerifyCustomerId* represent atomic propositions.

Mapping from BPSL to LTL

Referring to Figure 4, we start with *Article11_part2*. As explained, *Article11_part2* (and *Article13*) is a simple temporal sequence that specifies the temporal relation among a sequence of Boolean blocks or other BPSL properties along paths in which time advances monotonically. The name of each Boolean block in *Article11_part2* represents a business activity being performed and the data associated with it. For example, the Boolean block *AcceptCustomerReq* is interpreted as *action* = *AcceptCustomerReq*, and the Boolean block of *ObtainCustomerInfo* with the parameter list (*ParaList*) of name and ID is interpreted as *action* = *ObtainCustomerInfo* & *ParaList* = {*name, ID*}. Note the diamond symbol under the Boolean block of *AcceptCustomerReq*. This is a special feature of BPSL named *compensation property* (*cp*). A *cp* specifies that when the Boolean block associated with the *cp* does not hold, the whole BPSL property is still deemed to be correct if *cp* holds. For this example, the diamond stands for a shortcut to the *cp* being True. It specifies that *Article11_part2* will be evaluated only when a customer request is accepted (because it is always allowable for *AcceptCustomerReq* not to hold). The key word *Always* in the formula of *Article11_part2* is a global temporal operator, which has a direct interpretation in LTL as the *G* operator.

Four types of global temporal operators are supported by BPSL: (*Possible*) *Always,* (*Possible*) *Eventually, Repeat,* and *Never.* (*Possible*) *Always* is equal to *G* or *AG*(*EG*); (*Possible*) *Eventually* is equal to *F* or *AF*(*EF*). *Repeat* and *Never* mean that the temporal sequence must hold at least *n* times or that it must not hold at all. Different Boolean blocks are associated with the temporal operators in BPSL. For example, in ***Figure 6A*** *ExistWithin [inf]* specifies that after a customer request is accepted (*AcceptCustomerReq*), the customer information including the customer name and ID must be obtained. Here *inf* is the scope parameter for the operator *ExistWithin* with the value of *infinity*. Seventeen stereotypes of temporal operators with different semantics are supported in BPSL to specify temporal relations in different situations. While some of the temporal operators have a direct mapping to LTL temporal operators (e.g., *Next [3]* for *XXX* and *AllWithin [inf]* for *G*), others can be used to express rather complex temporal relations in a simple and compact manner (e.g., the *MultiWithinOnEvt [scope] [n]* operator specifies the scenario that a Boolean block must hold for *n* times when a certain *event* occurs within the *scope*). Consequently, the final LTL formula corresponding to *Article 11_part2* is:

$$G(action = AcceptCustomerReq \rightarrow F((action = ObtainCustomerInfo \,\&\, Paralist = \{name, ID\}) \,\&\, F(action = VerifyCustomerIdentity \,\&\, Paralist = \{name, ID\}) \,\&\, F(action = RecordCustomerInfo)))$$

The mapping of this formula to the BPSL notations is shown in Figure 6A.

As to *Article13*, two new elements need to be introduced. The rectangle *TRUE* indicates a Boolean block that will always hold. The notation of *LargeDeposit,* on the other hand, is a post-condition associated with this Boolean block. In BPSL, a post-condition specifies whether it is necessary to further evaluate the rest of the temporal sequence after a Boolean block. For example, in *Article13*, the post-condition of *LargeDeposit* specifies that *ReportLargeDeposit* will (only) be performed after a *LargeDeposit* is detected (i.e., *action* = *LargeDeposit* → *F (action* = *AcceptCustomerReq)*). Consequently, the final LTL formula corresponding *to Article13* is:

$$G(action = AcceptCustomerReq \rightarrow F(action = Large\,Deposit \rightarrow F(action = ReportLargeDeposit)))$$

The mapping of this formula to the BPSL notations is shown in ***Figure 6B***.

Finally, *Article11_part1* is captured using a compound temporal operator which specifies the logical relation (*And*) between two simple temporal sequences. Here a new temporal operator *Precede*

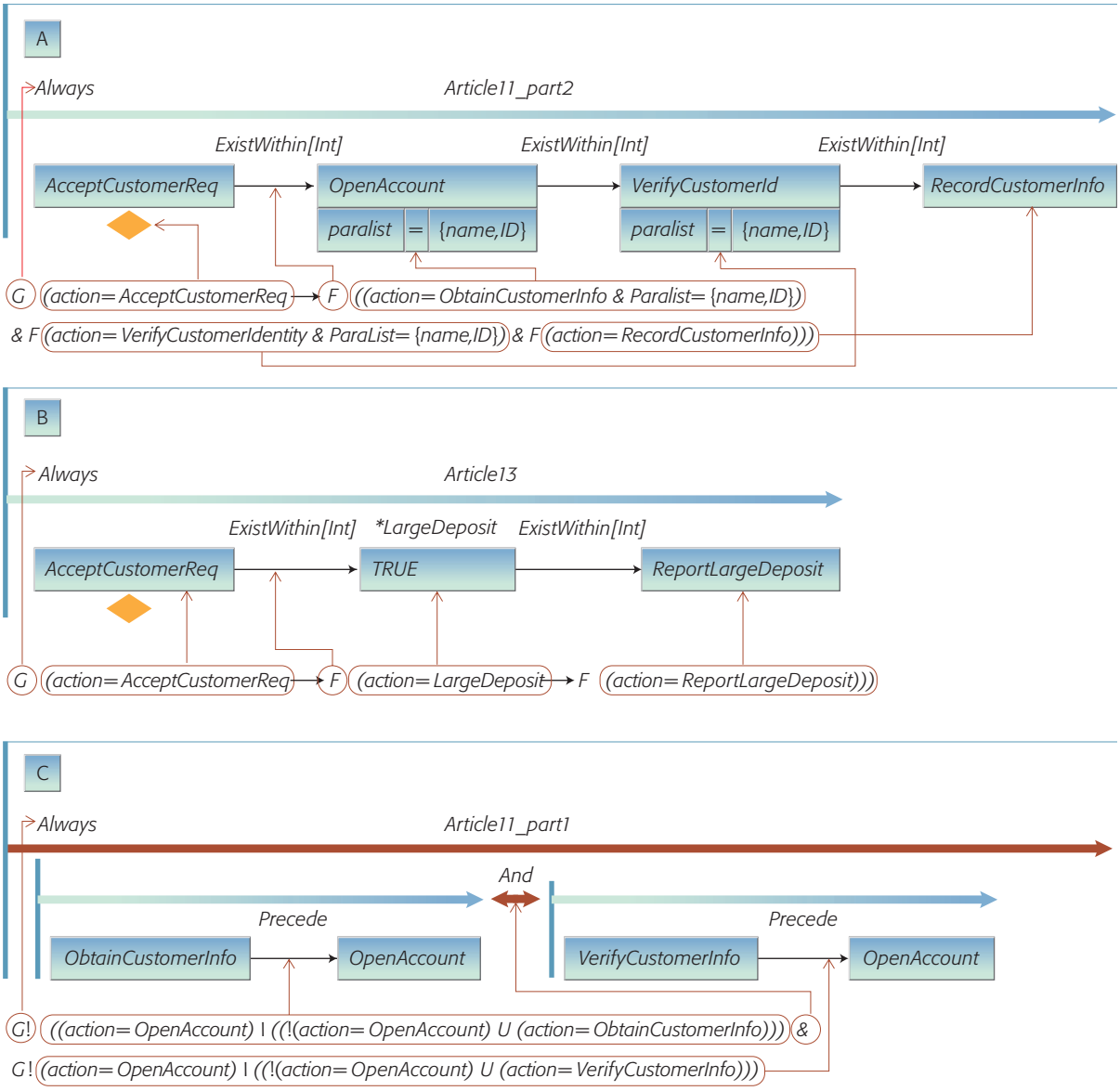

Figure 6
Mapping from BPSL to LTL: (A) for *Article11_part2*, (B) for *Article13*, and (C) for *Article11_part1*

appears. The *Precede* relation (*a Precede b*) specifies that either *b* never occurs or that there is no occurrence of *b* before *a* holds, that is,

$$G!(action = OpenAccount)|$$
$$(!(action = OpenAccount) \cup$$
$$(action = ObtainCustomerInfo)).$$

Consequently, the final LTL formula corresponding to *Article11_part1* is:

$$G!(action = OpenAccount) |$$
$$((!(action = OpenAccount) \cup$$
$$(action = ObtainCustomerInfo))) \&$$
$$(!(action = OpenAccount) |$$
$$((!(action = OpenAccount) \cup$$
$$(action = VerifyCustomerInfo))))$$

The mapping of this formula to the BPSL notations is shown in ***Figure 6C***. The above mapping follows

exactly the semantics of BPSL[15] and thus ensures its correctness.

COMPLIANCE-CHECKING FRAMEWORK AND CASE STUDY RESULTS

With FSM and LTL, model-checking technology can be used to verify whether the business process complies with the compliance rules. We briefly explain the basic concepts of model checking and then introduce our compliance-checking framework.

Finally, we present a case study result to illustrate the practical feasibility of this framework.

Model checking

Model checking[18] is an automatic technique for verifying finite state systems. It has been successfully applied to many systems, including hardware designs and communication protocols. The main idea of model checking is to search the state space of a system model and to verify that it satisfies some user-defined properties (e.g., temporal constraints, such as liveness or safety properties). The advantage of model checking over traditional simulation and testing is that it can exhaustively search the whole state space of a system and can prove the system is indeed error-free. The advantage of model checking over deductive verification is that it requires less expertise and experience in logical reasoning on the part of users. In fact, the procedure of model checking needs little user intervention; it can be performed automatically and results in a final yes or no answer.

Model checking generally includes three steps:

1. *Transforming the target system that is to be checked into a formal system model*—In our case, the target is the business process model expressed in BPEL, and the formal model we chose is pi-calculus.[16,29] The total behavior of the business process can thus be obtained in an FSM expressing the operational semantics of pi-calculus.

2. *Specifying the properties that the formal system model is expected to satisfy*—LTL is often used to capture such specifications. As writing correct statements in temporal logic is relatively difficult, for our compliance-checking method, we developed BPSL, a visual property specification language, to formally express such properties. BPSL properties are then automatically translated into LTL formulas.

3. *Performing the verification of the formal system model against desired properties with model-checking algorithms*—The generated result indicates whether the system satisfies the properties. Our compliance-checking framework, Open Process AnaLyzer (OPAL), integrates the binary decision diagram (BDD)-based symbolic model-checking algorithm implemented in NuSMV2.[30] The validity and effectiveness of the algorithm has already been proven from both the academic and industrial sides. In addition, OPAL also provides a counterexample tracing capability for business process models and offers its own optimization approach for compliance checking.

Compliance-checking framework

Figure 7 is an overview of the OPAL toolkit, our implementation of the compliance-checking framework. While OPAL supports compliance checking of different business process models against compliance rules, the OPAL framework is independent of a specific business-process modeling approach and model-checking method. OPAL offers an open framework to integrate different business-process modeling tools (e.g., WBI Modeler) and model-checking engines (e.g., NuSMV2[30] and Rule Base[31]) by means of the process-model-to-pi transformer and the model-checker adapter, respectively. The component-process-model-to-pi transformer generates the pi processes for different business process models. Compliance rules are specified in the BPSL editor. Because the elements in a compliance rule are closely related to the business process model, there is a graphical user interface (GUI) adapter between the BPSL editor and the process modeling tool. For example, some activity elements can be directly dragged into the BPSL editor from the process modeling tool through the respective adapter.

As introduced earlier, we use pi-calculus as the method to formalize business process models, and LTL is used to specify temporal properties. The details of the BPSL-to-LTL transformer, the process model-to-pi transformer, and pi-to-FSM transformer were introduced earlier. The model-checker adapter is used to integrate existing model-checking engines, such as NuSMV2 and Rule Base, with OPAL. The framework uses the counterexample tracer to trace counterexamples in the business process model. Further, compliance checking results can be gener-

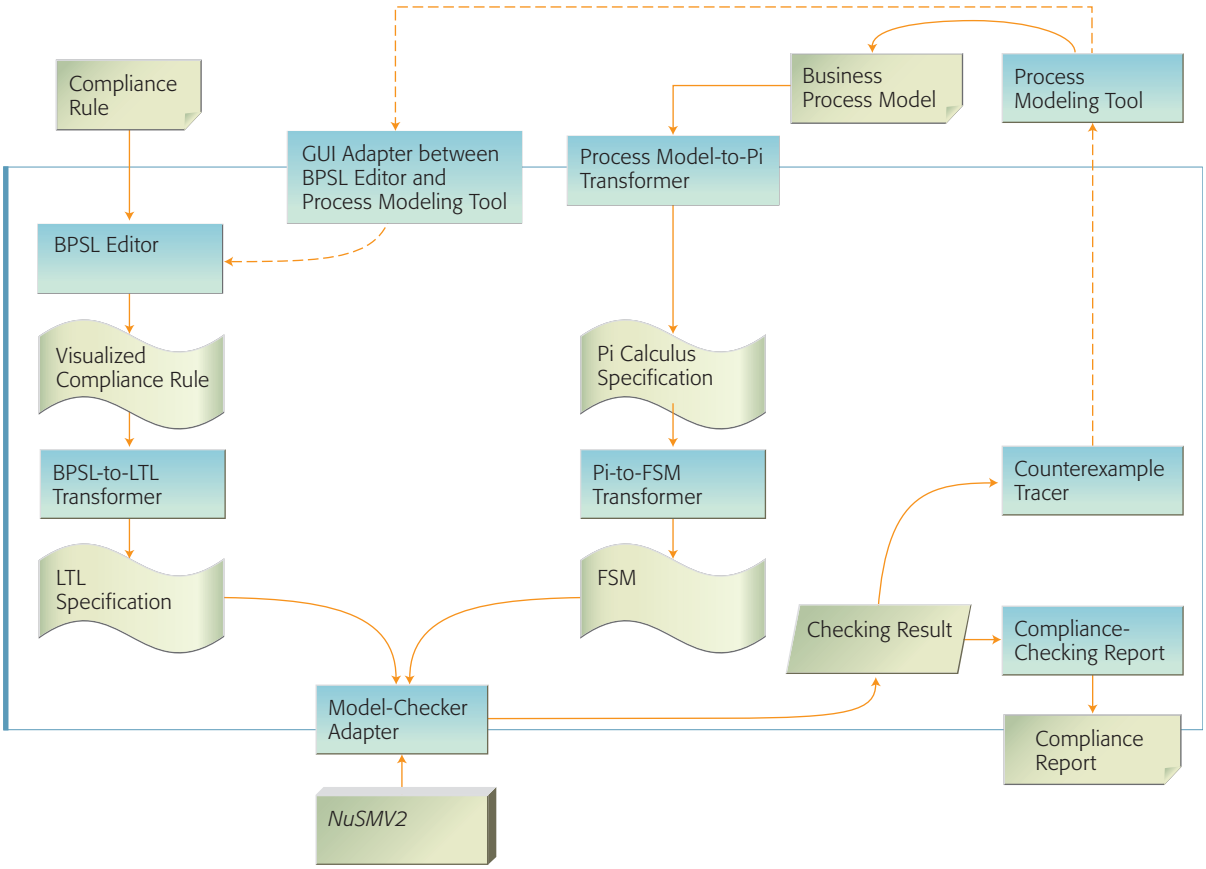

Figure 7
Compliance-checking framework

ated as compliance reports using the compliance-checking reporting module.

Because this framework is intended for use by business people for business-process model checking, two aspects must be addressed. The first aspect is the understandability of the checking results. For example, if a business process model does not comply with a compliance rule, business people require help to position error points in the process model. The counterexample tracer can help solve this problem. The other aspect is how to ensure an acceptable performance of the compliance-checking method. If a business process model is very complex, optimization helps improve the performance of compliance checking. These important OPAL features are discussed later.

Running example with results

Recall the SimpleBank account-opening process that was described in the introduction. It has been used as a running example throughout the paper to illustrate our compliance-checking method. We have applied OPAL to check the compliance of the account-opening process in Figure 3 against the compliance rules defined in Figure 4. As explained in the section "Model transformations," the account-opening process was automatically transformed to pi-calculus and further into an FSM with the help of OPAL. Likewise, the regulatory requirements of Article 11 and Article 13 were formalized as compliance rules using the OPAL BPSL modeler, and were then automatically transformed into LTL.

OPAL was developed as a plug-in for the Eclipse** platform, which allows for the integration with different business process modelers, such as WBI Modeler and other Eclipse-based BPEL editors.[32] We tested our case study using OPAL on a Microsoft Windows** platform with an Intel** Pentium** 4 processor, 3.0 GHz, and 2.5 GB random access memory (RAM), and obtained the following results:

OPAL took 0.056 seconds to transform the account-opening process expressed in BPEL into pi expressions, as introduced in the section "Model transformations." The time consumed for transforming the pi-calculus formalization into its corresponding FSM was 49.959 seconds. The final FSM contained 11,832 (i.e., 2^13.5304) reachable states out of 535,840 (i.e., 2^19.0314) states after OPAL applied its opimization by means of sequentialization of interleaving actions to compact the FSM (introduced later in the section "Sequentialization of interleaving actions"). The checking of the three compliance rules in Figure 4 consumed 121.0 seconds of CPU time. This also included an additional 21.0 seconds to generate the needed counterexamples for the violated compliance rules. The peak memory use was 71.960 MB.

As the final compliance-checking results showed, the account-opening process complies with the two compliance rules *Article11_part1* and *Article13*. However, it does not comply with the compliance rule *Article11_part2*. The counterexample for *Article11_part2* contains the state trace of 54 states, which shows there is a possible execution path in the account-opening business process in Figure 1 in which the customer information was already recorded in the banking system before it was verified for correctness.

As we have explained so far, OPAL is capable of automatically checking the compliance of our account-opening process model against the three compliance rules in the example. Thus, OPAL users realize that the current account-opening process is noncompliant before deployment, which helps increase deployment efficiency and lowers the risk of installing noncompliant processes.

Our experience with OPAL has shown that our current implementation can handle a state space with 10^6 reachable states out of 10^8 total states within 15 minutes.

ADVANCED FEATURES

Two important features of the framework, counterexample tracing and performance optimization, are addressed in this section.

Counterexample tracing

If a regulation rule is not satisfied by a business process model, normally a counterexample would be generated only in the corresponding FSM of the process model because the model-checking algorithm is executed based on FSM. However, for business people, a counterexample based on an FSM is meaningless. Hence, we must provide a mechanism to trace the counterexample back to the business process model. We have developed such a mechanism for OPAL.

As the previous counterexample in our running example is too large (a state trace of 54 states), we now present a simpler example to explain how the OPAL counterexample tracing mechanism works. The BPEL program of the account-opening process in Figure 3 is simplified to the BPEL program *SimpleAC*, given in ***Figure 8A***.

This simple BPEL process can be transformed to the pi process shown in ***Figure 8B*** by using the BPEL-to-pi transformation rules introduced earlier. (For simplification, some channel names have been shortened, for example, *AcceptCustomerReq* is abbreviated as *ACR*.)

In the formalism shown in Figure 8B, *allnames* represents all free names in *AccountOpeningProcess*, thus making *AccountOpeningProcess* a closed system with all its names restricted to itself. The formalization of $Variable_{acq}$, $Variable_{oci}$, and $Variable_{vci}$ is done as explained earlier in the section "Model transformations," where we also introduced how to transform a pi process to an FSM. Accordingly, the pi process can be transformed into the FSM shown in ***Figure 9*** with the help of OPAL.

Figure 9 shows only part of the FSM of the entire *SimpleAC Process* model, which in total contains 69 states as the transformation result of OPAL. Using OPAL, we may now check the following property, and we are thus informed that the result is wrong.

$$G(!(action = VerifyCustomerInfo))|$$
$$(!(action = VerifyCustomerInfo) \cup$$
$$(action = ObtainCustomerInfo))$$

OPAL visualizes the transformed FSM and the counterexample generated by NuSMV2. The counterexample is indicated by the state transitions in orange lines in Figure 9, showing a possible execution path in *SimpleAC Process* where the customer information may be verified before it is obtained in the first place.

A

```
<bpws:partnerLinks>
    <bpws:partnerLink name="ACQ" ...../>
    <bpws:partnerLink name="OCI" ...../>
    <bpws:partnerLink name="VCI" ...../>
  </bpws:partnerLinks>
  <bpws:variables>
    <bpws:variable name="variable_acq" ...../>
    <bpws:variable name="variable_vci" ...../>
    <bpws:variable name="variable_oci" ...../>
  </bpws:variables>
  <bpws:sequence name="HiddenSequence">
    <bpws:invoke inputVariable="variable_acq" name="AcceptCustomerReq"
                 outputVariable="variable_acq" partnerLink="ACQ"/>
    <bpws:flow name="ParallelActivities">
      <bpws:invoke inputVariable="variable_oci" name="ObtainCustomerInfo"
                   outputVariable="variable_oci" partnerLink="OCI"/>
      <bpws:invoke inputVariable="variable_vci" name="VerifyCustomerInfo"
                   outputVariable="variable_vci" partnerLink="VCI"/>
    </bpws:flow>
  </bpws:sequence>
```

B

$$Receive\ AcceptCustomerReq = Start_{acq}.ACR\ (v).\overline{put}_{acq}\langle v\rangle.\overline{done}_{acq}$$

$$Invoke\ ObtainCustomerInfo = Start_{oci}.Get_{acq}\ (v).\overline{OCI}\langle v\rangle.OCI(w).\overline{put}_{oci}\langle w\rangle.\overline{done}_{oci}$$

$$Invoke\ VerifyCustomerInfo = Start_{vci}.Get_{oci}(v).\overline{VCI}\langle v\rangle.VCI(w).\overline{put}_{vci}\langle w\rangle.\overline{done}_{vci}$$

$$Flow(Invoke\ ObtainCustomerInfo, Invoke\ VerifyCustomerInfo) = new\ ack((Invoke\ ObtainCustomerInfo \mid done_{oci}.\overline{ack}) \mid (Invoke\ VerifyCustomerInfo \mid done_{vci}.\overline{ack}) \mid ack.ack.\overline{done}_{flow}$$

$$Sequence(Receive\ AcceptCustomerReq, Flow(Invoke\ ObtainCustomerInfo, Invoke\ VerifyCustomerInfo)) = (Receive\ AcceptCustomerReq \mid Done_{acq}(\overline{Start}_{oci} \mid \overline{Start}_{vci}) \mid Flow(Invoke\ ObtainCustomerInfo, Invoke\ VerifyCustomerInfo)$$

$$Partner\ AcceptCustomerReq = \overline{ACR}\langle req\rangle$$

$$Partner\ ObtainCustomerInfo = OCI\langle w\rangle.\overline{OCI}\langle Info\rangle$$

$$Partner\ VerifyCustomerInfo = VCI\langle w\rangle.\overline{VCI}\langle Info\rangle$$

$$AccountOpeningProcess = new\ \overline{all\ names}(\overline{Start}_{acq} \mid Sequence(Receive\ AcceptCustomerReq, Flow(Invoke\ ObtainCustomerInfo, Invoke\ VerifyCustomerInfo)) \mid Partner\ AcceptCustomerReq \mid Partner\ ObtainCustomerInfo \mid Partner\ VerifyCustomerInfo \mid Variable_{acq}(acq) \mid Variable_{oci}(oci) \mid Variable_{vci}(vci))$$

Figure 8
BPEL simple account-opening program (A) in XML and (B) transformed to the pi process

Because the counterexample in the preceding diagram is given at a state machine level and contains information about many redundant actions, such as $start_{ac}$ and *ack*, it is hard for business people to comprehend what the counterexample stands for from a business perspective. Therefore, OPAL provides a counterexample mapping from the FSM to the BPEL process model. Such mapping is implemented by preserving only the actions that have a direct connection with the corresponding BPEL activities (e.g., *AcceptCustomerReq*, *ObtainCustomerInfo*, *VerifyCustomerInfo*) and remove any other redundant information in the counterexample. The mapped counterexample in BPEL is the following:

Invoke AcceptCustomerReq
→ *Invoke VerifyCustomerInfo*
→ *Invoke ObtainCustomerInfo*

Performance optimization

The basic idea of model checking is to exhaustively search the state space of formal system models to discover potential violations of specific logical constraints that a user specifies. To make model checking more applicable to realistic large-scale models, performance tuning and improvement of model checking is a critical research area. OPAL reuses optimized, state-of-the-art model-checking algorithms and focuses primarily on the business process level with its own optimizations. Specifi-

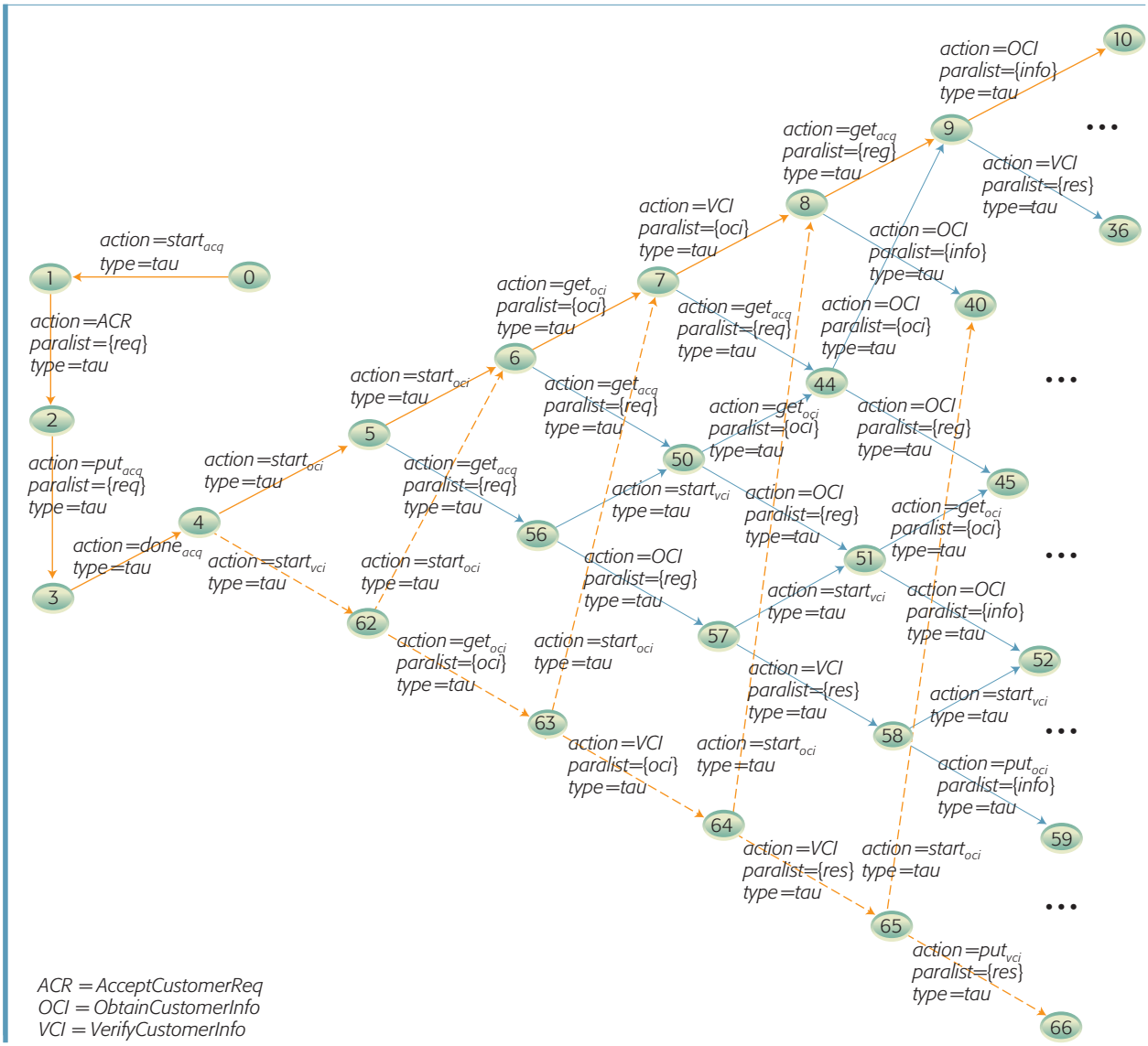

Figure 9
FSM example

cally, it improves the performance of compliance checking by concentrating on the following two aspects:

1. *State space reduction of the business process when transforming it to an FSM*—States that do not influence the checking result in the FSM are identified and removed so that the state space of the business process can be decomposed or radically reduced. In OPAL, sequentialization of interleaving actions, explained in the next section, is one of the methods implemented to remove redundant states from the state space.
2. *Controlled state-space searching using business bug patterns*—Compliance checking can be rendered more efficient by a guided search for bugs in the business process.

Sequentialization of interleaving actions

As we know, concurrency is a major cause of state-space explosion. Accordingly, unnecessary concurrencies in business process models may be eliminated to avoid redundant states. Inspired by the idea of *partial order reduction*,[18] OPAL tries to remove unnecessary concurrencies in the pi-calculus specification that do not affect the semantics of the

corresponding business process. We call this approach *sequentialization of interleaving actions*. When a business process is formalized in pi-calculus and transformed to an FSM, the state space can thus be made as compact as possible.

To explain the rationale of our approach, we take the formalization of *SimpleAC* as an example. In our *SimpleAC* example, there are two parallel activities: *ObtainCustomerInfo* and *VerifyCustomerInfo*. It seems reasonable to model the triggering of the two activities in a concurrent form, that is, $\overline{start}_{oci}|\overline{start}_{vci}$ in the formalism of *Sequence*, as these two activities are executed independently and in an arbitrary order. However, it is easy to note that $\overline{start}_{oci}$ and $\overline{start}_{vci}$ play only the role of triggering the execution of two pi processes, but do not affect the execution order of the two activities *ObtainCustomerInfo* and *VerifyCustomerInfo*. Consequently, even if $\overline{start}_{oci}|\overline{start}_{vci}$ are sequentialized as $\overline{start}_{oci}.\overline{start}_{vci}$ the internal behavior of the two activities (such as the retrieval and assignment of variables by means of *Get* and *Put* and the invocation of *PartnerLinks* ℓ) is still interpreted in an interleaving form. To be intuitive, the execution of $\overline{start}_{oci}$ and $\overline{start}_{vci}$ in either order can result in the same global state in the FSM of *SimpleAC*. Therefore, we can safely replace this concurrency with a sequence and reduce the redundant states caused by the concurrency. The same situation also holds for the formalism of *Flow*, where $done_{oci}$ and $done_{vci}$ are modeled as $done_{oci}|done_{vci}$.

Thus OPAL avoids unnecessary concurrencies in the formalization of a business process model when transforming it into an FSM. Typical sequentializations are implemented in OPAL, including the formalism of the *Fork* nodes, *Join* nodes, multiple inputs and outputs for an activity in the UML activity diagrams and compatible models, the *Flow* structure, and multiple incoming and outgoing links for an activity in BPEL models. The sequentialization in the pi-calculus processes does not mean that the corresponding activities in the BPEL process are sequentialized.

As an example, the state-transition diagram of *SimpleAC* can be simplified. In ***Figure 10***, the states numbered from 62 to 65 and their transitions indicated by orange dashed lines show the part of the FSM that can be reduced by optimization. The optimized FSM is reduced to 43 states. Internal experiments have demonstrated the practical value of state-space reduction using sequentialization of interleaving actions, especially for complex process models with many unnecessary parallelisms.

Guided state searching for business bug patterns

Despite great improvements in the performance of model checking, the exploitation of domain knowledge is crucial to further improve the efficiency of compliance checking. Because model checking is more useful to probe hidden bugs in a system than to prove its correctness, we have developed business bug patterns, that is, a set of antipatterns corresponding to the well-known workflow patterns[33] to represent common behavioral violations in a business process. A guided search mechanism is then implemented to more efficiently search for these business bugs in a business process model.

To explain the main idea of business bug patterns,[34] we take the simple sequential pattern between two activities, A and B, as an example. To falsify the semantics that activity A and B are executed in a sequential order, a business bug pattern *SequentialBug(A, B)* is shown below:

$$SequentialBug(A,B) = SimultaneousExecution(A,B) \vee NoResponse(A,B)$$

$$SimultaneousExecution(A,B) = \{[*]; !A.Exit \,\&\, B.Start\}$$

*/*After a certain number of steps, a state is reached in the process where B is started while the execution of A does not yet take place.*

$$NoResponse(A,B) = \{[*]; A.InExecution; A.Exit\} \mapsto \{B.InExecution[=0]\}$$

*/*If A is finished in the process, no B will be executed afterwards.*

The semantics of the *SequentialBug* pattern are formally captured with the IEEE Property Specification Language (PSL).[35] Contrary to *Sequential(A, B)*, the sequential bug pattern tries to find that either both *A* and *B* start their execution simultaneously or that *B* is never executed after *A* is done. The above two aspects can be defined with two more atomic bug patterns, *SimultaneousStart(A, B)* and *NoRe-*

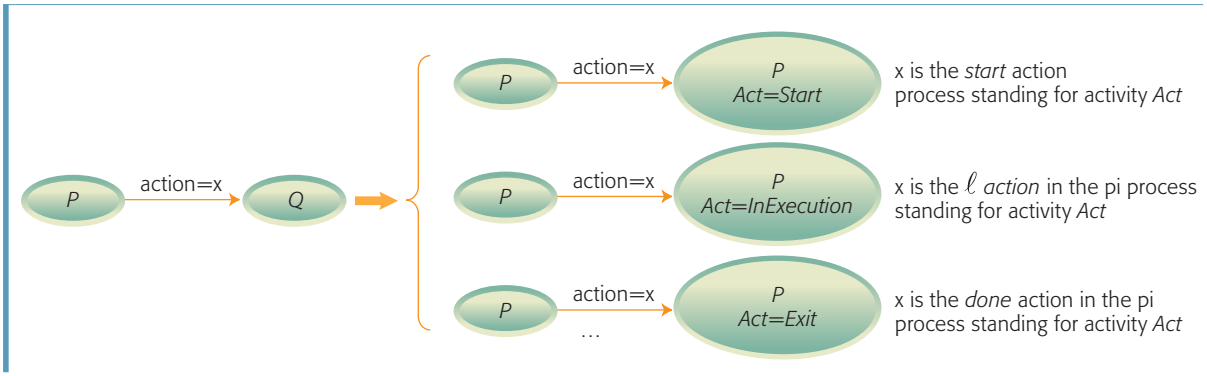

Figure 10
Mapping example from pi-calculus to FSM

sponse(A, B), respectively. The symbol ∨ indicates that the *SequentialBug* pattern holds when either *SimultaneousStart* or *NoResponse* is satisfied. Here, the *SequentialBug* pattern does not necessarily check whether A is possibly executed after B, as this is acceptable (e.g., when B loops back to A).

In the above definition, the form of *A.Exit* indicates that the execution of activity A is terminated. The execution status (e.g., *Start, InExecution*, and *Exit*) can be encoded in the FSM model of the business process according to the actions in the pi-calculus process that has been enacted. Figure 10 shows an example of such a mapping for the *Receive* activity.

A full reference of all the business bug patterns can be found in our previous work.[34]

To more efficiently probe the potential existence of such bugs in a business process model, our idea is to always follow a subset of interesting states while traversing the state space. Interesting states are those that can lead to the detection of a targeted business bug pattern within the least number of transitions. More specifically, we define the following:

M(m): the complete state space (universe) of a business process *m*, with its initial state in which all of the activities are *NotStarted*;

$S(m) = \{s(act1), s(act2), \ldots\}$: A state in *M(m)* that is encoded as the states of all activities in *m*, where $act_i \in m$ & $s(act_i) \in \{act_i.NotStarted, act_i.Start, act_i.InExecution, act_i.Cancel, act_i.Failed, act_i.Exit\}$;

The distance between two activity states *D(s(act)1, s(act)2)* is thus defined as the least number of transitions from one state $s(act)_1$ to another state $s(act)_2$. For example, if $D(act.Start, act.Exit) = 2$, it means that at least two steps are needed from state *Start* to state *Exist* for action *act*. Therefore, the distance between two states in the process is defined as the weighted average of *D*:

$$D\_S(S(m)1, S(m)2) = \sum i\, D(s(act)1i, s(act)2i)/|S(m)|$$

The interesting states for a given commitment state *CS* in state set *SS* are thus:

$$S(m)\_CS = \{S(m) | S(m) \in SS, \forall S'(m) \in SS\, D\_S (S(m), CS) < D\_S(S'(m), CS)\}$$

A more detailed reference of our guided searching mechanism and its algorithms can be found in our previous work.[34] Our experiments have shown that the guided reasoning of business bug patterns can help improve the performance of compliance checking in OPAL. ***Table 1*** shows a set of experimental results on the compliance checking of the account-opening process in Figure 1, which has a total state space of 8,361 (2^13.0295) reachable states. The guided business bug-searching algorithm has been implemented in OPAL. The test environment was again a Windows platform with an Intel Pentium 4 processor, 3.0 GHz, and 2.5 GB RAM.

The results show that our guided business-bug searching approach can improve the performance of finding potential violations in a business process compared to the original approach. Intuitively speaking, the reason for the improvement is that our approach takes advantage of the pre-identified

Table 1 Experimental results on checking-account-opening process

Target Bug	Original Approach (seconds)		Guided Bug Hunting (seconds)	Result
	From pi to FSM	Model checking		
Bl	57.218	118.200	2.031	Found
B2		112.700	41.703	Not found
B3		177.600	26.438	Found
B4		129.300	60.469	Found
B5		221.900	72.266	Found
B1	*SequentialBug(VerifyCustomerIdentity, RecordCustomerInfo)*			
B2	*MilestoneBug(ProposeAccountOpening.Exit, ActivateAccount.PreStart, ValidateAccountInfo)*			
B3	*InterleavedParallelRoutingBug(OpenAccount, DoDeposit, RecordAccountInfo)*			
B4	*ExclusiveChoiceBug(AcceptCustomerReq, {VerifyCustomerIdentity, ValidateAccountInfo})*			
B5	*SequentialBug(AcceptCustomerReq, PrepareProposalDoc) && ParallelSplitBug(PrepareProposalDoc, {VerifyCustomerIdentity, ProposeAccountOpening}) && SequentialBug(ProposeAccountOpening, ActivateAccount) && SynchronizingMergeBug({ActivateAccount, VerifyCustomerIdentity}, DoDeposit)*			

Mlv (the maximum interesting level for the next states to be traversed) = 1;
Gate (the maximum number of times allowed for computing preimages for the current states) = 10

activity status and always follows the shortest path that may lead to the detection of a targeted bug. Uninteresting state traces are neglected to narrow down the state space that needs to be traversed. Therefore, the guided business-bug searching approach is useful to check the compliance of complex large-scale business processes. However, the approach is not perfect; its merits come at some cost too:

- It can be used only to falsify a business process because it does not ensure the full traversal of the state space of the business process. Because model checking is more useful for finding system bugs than to prove them correct,[36] the approach is still valuable for checking compliance of real industrial business designs that are too large for classical model checking to run to completion.
- It is not suitable for application in small-scale business processes. On the one hand, it is totally affordable to have a thorough and precise checking of simple business processes with model checking. On the other hand, the computation of the interesting states in the approach is not negligible for the compliance checking of a simple business process; thereby, giving away the performance advantage of the approach.

RELATED WORK

With the rapid growth of complexity in existing business applications and their supporting IT infrastructures, ensuring highly secure and reliable business process development is becoming a critical task. In the past few years, there has been a lot of work modeling business processes and developing verification techniques and tools for them. A recent survey was done by Breugel and Koshkina.[37] In this section, we identify three aspects pertaining to the compliance checking of business process models. We report on the literature in each area and clarify how our work is different.

Pi-calculus as the formal foundation for business process models

Many researchers generally agree that formal models should be used as a basis for complex

business-process modeling languages, such as BPEL.[38] According to Sid Askary, "It will allow us to not only reason about the current specification and related issues, but also uncover issues that would otherwise go unnoticed."[39] In this context, tremendous focus has been concentrated on Petrinet and pi-calculus. However, this has also led to a long debate on what is the most suitable formal foundation for business process models. Smith supports pi-calculus and argues that "... workflow is just a pi-calculus."[40] The design of Web Services composition languages, such as XLANG and BPEL, is also claimed to be based on pi-calculus. Van der Aalst, on the other hand, has appealed that more solid work should be done to prove the effectiveness of pi-calculus in modeling business processes.[38] In fact, there has been some previous work on formalizing various business process models with pi-calculus, including UML statechart diagrams,[41,42] UML (2.0) activity diagrams,[43,44] and workflow patterns.[45] Previous work has also shown that pi-calculus is a suitable formal composition language for software composition and Web service composition.[46–48] In our work, we have formalized BPEL process models with pi-calculus instead of using Petrinets or automata (and their extensions).

We rely on pi-calculus for the following reasons. First, automata and Petrinets are often used to model closed systems, whose behavior is completely determined and controlled by the state of the system. However, pi-calculus, aside from its mobility feature, is designed to model open communicating systems whose behavior is determined by the state of the system and interaction with the behavior of the environment.[29,49] For example, FSM model is under complete control of its transitions, whereas in pi-calculus, all observable actions are under the joint control of the process and its environment. Therefore, one may regard FSM as the processes in pi-calculus with only internal actions.

We favor pi-calculus over BPEL because, although BPEL can be regarded as a fully controllable orchestration of various services, there are also cases when the behavior of a BPEL process (e.g., a service invocation according to the WSDL specification) needs an interactive feedback from the environment of the process (e.g., a dynamically changing service portfolio). For example, when BPEL is used as an abstract service-composition language with the automatic discovery and mapping of the target services for invocation, it becomes critical to consider the communication with the environmental information, such as the available service candidates in the service portfolio and service selections (which can be best modeled by the mobility feature of pi-calculus).

Another advantage of pi-calculus is its mobility and compositionality. Here compositionality means that there is a natural composition operator in pi-calculus to model a system from its subcomponents. This operator does not exist in Petrinets. Therefore, for composing various Web services by BPEL to form a process, it is more natural and beneficial to use a compositional language like pi-calculus instead of FSM or Petrinet, which involve additional operations and computations for the composition.

Pi-calculus is theoretically sound and supports bisimulation analysis and model checking. It enjoys increasing acceptance and tool support in the industry. It has also been used as the formal foundation for business-process modeling languages such as BPEL and XLANG. However, as pointed out by van der Aalst,[38] more work needs to be done to provide formal models, verification approaches, and automatic tools for business processes based on pi-calculus. Our work can be regarded as a response to this appeal.

Formal verification of BPEL process models

Based on the formal semantics, there has been previous work on the formal verification of BPEL process models. Fu et al.[50] first translate BPEL processes into (guarded) automata, and LTL model checking can thus be performed with SPIN[51] with an additional transformation from (guarded) automata to Promela (the input language used by the free model checker of SPIN, developed by Bell Laboratories).[51] Besides, Fu et al.[50] studied the so-called synchronizability and realizability analysis for the composition of Web services. Kovács and Gönczy[52] also exploit the model checker of SPIN, although the intermediate model between BPEL and Promela is a model of a dataflow network. Foster et al.,[53] on the other hand, take the BPEL process and translate it into the form of finite state process calculus and then compile it into a labeled transition system. The formal verification is then performed by the existing labeled-transition-system analyzer tool suite. Ouyang et al.[54] and Lohmann et al.[55] both provide a

semantic transformation from BPEL to Petrinet. However, whereas Ouyang et al.[54] focus on the analysis of specific process properties, such as reachability analysis, competing message-consuming activities, and garbage collection of queued messages, Lohmann et al.[55] focus on the controllability of the process; that is, whether a strategy can be constructed to impose the weak termination property on the corresponding workflow net. Finally, instead of dealing with the BPEL model, Koehler et al.[56] propose a pattern-based mapping approach to model a general business process. Two typical properties in a process model, reachability and termination, are formulated with the temporal logic of CTL, which can be verified later by existing model checkers.

Our compliance-checking approach differs from these works in two ways:

1. The theoretical foundation is different. We have used a pi-calculus-based approach instead of an automata- or Petri-net-based approach. The benefits of our selection were addressed in the previous subsection.
2. The completeness of the approach is different. We focused on the verification of our formalized BPEL models against specific structural errors. More important, our work involves a more detailed proposal of subjects—including counterexample guiding, performance enhancement, and visualization of temporal logics—which are critical to making the formal verification of business processes practical and usable.

Specifying regulatory rules with temporal logics

Specifying user-desired properties with logical formulas is an important step in the formal verification of business process models. The intuitiveness and convenience in the property specification thus becomes a key issue in making the formal verification approach more applicable to business analysts, who may not be logical experts. The LTL model checker plug-in in ProM[57] exploits a textual form of LTL formula directly. The work in Giblin et al.[12] extends a timed propositional temporal logic and is devoted to the specification of regulatory rules in a textual form. REALM[12] provides several easy-to-use features, such as a predefined set of business entity types (e.g., *Artifact, Resource*, and *Principle*) and relations (e.g., *Do, Input*, and *Output*) whose syntax conforms to a UML profile. Unfortunately, there is still no tool support for the verification of REALM specifications.

On the other hand, visual extension to existing logical languages is an important research direction to help business analysts understand and specify logical formulas intuitively. Related visualization works can be found for commonly used temporal logics, including CTL,[58] LTL,[59] and interval temporal logics.[60] Especially in DecSerFlow,[61] a graphical representation of the so-called Declarative Service Flow Language is proposed, which can be mapped onto LTL and enables the LTL verification of Web-service flow models.

As explained earlier, our OPAL toolkit contains an editor for BPSL to visually specify various regulatory rules. BPSL is different from the preceding works in the following aspects:

- It is a visual specification language which supports the temporal logics of both LTL and CTL. It is also compatible with the IEEE standard, Property Specification Language.[35]
- It enables the intuitive and convenient specification of regulatory rules by customizing predefined property templates in BPSL. The source of these templates comes from existing work on patterns such as business property specification patterns[23] and business bug patterns.[34]

CONCLUSION

We have introduced OPAL, a compliance-checking framework, and related tools, including a static method to check business process models against compliance rules. Compliance-checking tools enable one to quickly assess the compliance of business process models in batch mode. The use of high-level specification languages, such as BPEL (as opposed to pi-calculus or FSMs directly) and BPSL (as opposed to LTL specifications) and the definition of transformations that can be automated into low-level formalisms yields easier, more intuitive, and less error-prone process modeling, thus reducing the risk of implementation errors and noncompliant operations. If noncompliant business process models are discovered, counterexamples can be generated on the level of the business process model. This capability provides a better understanding of the nature of the problem and enables a quicker reaction to address and rectify the noncompliant processes.

These capabilities make it possible for business people to use the compliance-checking tool.

Our compliance-checking method builds on classical model-checking technology. After a business process model has been formalized with pi-calculus, it can be transformed into an FSM representation. The intermediate models of pi-calculus and FSM enable our compliance-checking framework to be scalable to both the future emergence of new business-process modeling techniques and the reuse of more powerful model-checking tools. As a matter of fact, although this paper mainly addresses the application of the framework in BPEL processes, our current implementation of the compliance-checking framework (the OPAL toolkit) has also been applied in the verification of WBI process models. As performance is always a critical problem in the area of model checking, we have also proposed the method of sequentialization of interleaving actions to reduce the overall state space. The approach of guided state searching for business bug patterns can further help improve the efficiency of compliance checking. As conducted experiments illustrated, these two optimization approaches can greatly help improve the performance of compliance checking.

We plan to extend the existing compliance-checking method to support the verification of resource and data constraints that are related to business process models. Additionally, we plan to focus on performance optimization. Finally, we intend to apply our compliance-checking method to more real cases to further validate the capabilities and usability of our compliance-checking framework.

ACKNOWLEDGMENTS

The authors in the author list contributed equally to this paper. The ordering of the author list follows the principle of alphabetical ordering according to the first character of family name. We would like to express our sincere thanks to Jun Zhu for his great help in improving this paper.

CITED REFERENCES

1. F. Leymann and D. Roller, *Production Workflow: Concepts and Techniques*, Prentice Hall PTR, Upper Saddle River, NJ (2000).
2. M. Havey, *Essential Business Process Modeling*, O'Reilly & Associates, Sebastopol, CA (2005).
3. *Gramm-Leach-Bliley Act of 1999* (GLBA), Public Law 106-102 (113 Statute 1338), United States Senate Committee on Banking, Housing, and Urban Affairs (1999).
4. *Sarbanes-Oxley Act of 2002*, Public Law 107-204 (116 Statute 745), United States Senate and House of Representatives in Congress (2002).
5. *USA Patriot Act of 2001*, Public Law 107-56, HR 3162 RDS, United States Senate and House of Representatives in Congress (2001).
6. *International Convergence of Capital Measurement and Capital Standards* (Basel II), Basel Committee on Banking Supervision (2004), http://www.federalreserve.gov/boarddocs/press/bcreg/2004/20040626/attachment.pdf.
7. *The Money Laundering Regulations*, Statutory Instrument 2003 No. 3075, Act of Parliament, http://www.opsi.gov.uk/si/si2003/20033075.htm.
8. *Law of the People's Republic of China on the People's Bank of China*, the 8th National People's Congress (2003), http://www.pbc.gov.cn/english//detail.asp?col=6800&ID=22.
9. *Control Objectives for Information and Related Technology* (COBIT), Version 4.0, IT Governance Institute (2005), http://www.itgi.org.
10. IT Infrastructure Library (ITIL), Office of Government Commerce (2006), http://www.itil.co.uk.
11. C. Abrams, J. von Känel, S. Müller, B. Pfitzmann, and S. Ruschka-Taylor, "Optimized Enterprise Risk Management," *IBM Systems Journal* **46**, No. 2, 219–234 (2007, this issue).
12. C. Giblin, A. Y. Liu, S. Müller, B. Pfitzmann, and X. Zhou, "Regulations Expressed as Logical Models (REALM)," *Proceedings of the 18th Annual Conference on Legal Knowledge and Information Systems*, Brussels, Belgium (2005), pp. 37–48.
13. *Rules for Anti-Money Laundering by Financial Institutions*, The People's Bank of China (2003), http://www.pbc.gov.cn/english//detail.asp?col=6800&ID=31.
14. T. Andrews, F. Cubera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, et al., *Business Process Execution Language for Web Services*, Version 1.1, BEA Systems, International Business Machines Corporation, Microsoft Corporation, SAP AG, Siebel Systems (2003), ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.
15. K. Xu, Y. Liu, and C. Wu, "BPSL Modeler–Visual Notation Language for Intuitive Business Property Reasoning," *Proceedings of the 5th International Workshop on Graph Transformation and Visual Modeling Techniques*, Vienna, Austria (2006), pp. 205–214.
16. R. Milner, *The Polyadic Pi-Calculus: A Tutorial*, Technical Report CS-LFCS-91-180, School of Informatics, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland EH8 9YL (1991).
17. A. Pnueli, "The Temporal Logic of Programs," *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, Providence, RI (1977), pp. 46–57.
18. E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*, MIT Press, Cambridge, MA (2000).

19. F. Leymann, *Web Services Flow Language (WSFL 1.0)*, IBM Corporation (2001), http://xml.coverpages.org/WSFL-Guide-200110.pdf.

20. S. Thatte, *XLANG: Web Services for Business Process Design*, Microsoft Corporation (2001).

21. WebSphere Business Integration Modeler, IBM Corporation (2006), http://www-306.ibm.com/software/integration/wbimodeler/library/.

22. K. Mantell, *From UML to BPEL*, IBM Corporation (2005), http://www-128.ibm.com/developerworks/webservices/library/ws-uml2bpel/.

23. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property Specification Patterns for Finite-State Verification," *Proceedings of the 2nd Workshop on Formal Methods in Software Practice*, Clearwater Beach, FL (1998), pp. 7–15.

24. M. Y. Vardi, "Branching vs. Linear Time: Final Showdown," *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Genova, Italy (2001), pp. 1–22.

25. K. Xu, Y. Liu, and G. G. Pu, *Formalization, Verification and Restructuring of BPEL Models with Pi Calculus and Model Checking*, Research Report, RC-23962, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (2006).

26. B. Jacobs and F. Piessens, "A Pi-Calculus Semantics of Java: The Full Definition," Technical Report CW 355, Department of Computer Science, Katholieke Universiteit Leuven, Leuven B-3001, Belgium (2003).

27. G.-L. Ferrari, S. Gnesi, U. Montanari, and M. Pistore, "A Model-Checking Verification Environment for Mobile Processes," *ACM Transactions on Software Engineering and Methodology* **12**, No. 4, 440–473 (2003).

28. M. Pistore, *History Dependent Automata*, Ph.D. thesis, University of Pisa, Pisa, Italy (1999).

29. D. Sangiorgi and D. Walker, *The Pi Calculus: A Theory of Mobile Processes*, Cambridge University Press, New York, NY (2001).

30. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," *Proceedings of the 14th International Conference on Computer-Aided Verification*, Copenhagen, Denmark (2002), pp. 359–364.

31. I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver, et al., "RuleBase: Model Checking at IBM," *Proceedings of the International Conference on Computer Aided Verification*, Haifa, Israel (1997), pp. 480–483.

32. WebSphere Studio Application Developer Integration Edition, IBM Corporation (2006), http://www-306.ibm.com/software/integration/wsadie/support/.

33. W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow Patterns," *Distributed and Parallel Databases* **14**, No. l, pp. 5–51 (2003).

34. K. Xu, Y. Liu, and C. Wu, "Guided Reasoning of Complex E-Business Process with Business Bug Patterns," *Proceedings of the IEEE International Conference on e-Business Engineering*, Shanghai, China (2006), pp. 195–202.

35. D. Geist, "The PSL/Sugar Specification Language: A Language for All Seasons," *Proceedings of The Correct Hardware Design and Verification Methods Conference*, L'Aquila, Italy (2003), pp. 21–24.

36. C. H. Yang and D. L. Dill, "Validation with Guided Search of the State Space," *Proceedings of the 35th Annual Conference on Design Automation*, San Francisco, CA (1998), pp. 559–604.

37. F. van Breugel and M. Koshkina, *Models and Verification of BPEL*, Technical Report, York University, Toronto, M3J 1P3, Canada (2006), http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf.

38. W. M. P. van der Aalst, "Pi Calculus Versus Petri Nets: Let Us Eat 'Humble Pie' Rather Than Further Inflate the 'Pi Hype'," *BPTrends* **3**, No. 5, 1–11 (2005).

39. Issue 42: Need for Formalism, OASIS Web Services Business Process Execution Language (WSBPEL) Technical Committee, http://www.oasis-open.org/archives/wsbpel/200307/msg00177.html.

40. H. Smith, "Business Process Management—The Third Wave: Business Process Modeling Language (BPML) and Its Pi-Calculus Foundations," *Information and Software Technology* **45**, No. 15, 1065–1069 (2003).

41. V. S. W. Lam and J. Padget, "Formalization of UML Statechart Diagrams in the Pi-Calculus," *Proceedings of the 13th Australian Software Engineering Conference*, Canberra, Australia (2001), pp. 213–223.

42. V. S. W. Lam and J. Padget, "Analyzing Equivalences of UML Statechart Diagrams by Structural Congruence and Open Bisimulations," *Proceedings of the IEEE Symposia on Human Centric Computing Languages and Environments*, Auckland, New Zealand (2003), pp. 137–144.

43. Y. Dong and Z. Shensheng, "Using /spl Pi/-Calculus to Formalize UML Activity Diagram for Business Process Modeling," *Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, Huntsville, AL (2003), pp. 47–54.

44. K. Xu, Y. Liu, J. Zhu, and C. Wu, "Pi Calculus Based Bi-transformation of State-Driven Model and Flow-Driven Model," *International Journal of Business Process Integration and Management* **2** (In Press 2006).

45. F. Puhlmann and M. Weske, "Using the Pi-Calculus for Formalizing Workflow Patterns," *Proceedings of the International Conference on Business Process Management*, Nancy, France (2005), pp. 153–168.

46. O. Nierstrasz and T. D. Meijler, "Requirements for a Composition Language," *Proceedings of the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems*, Bologna, Italy (1994), pp. 147–161.

47. M. Lumpe, F. Achermann, and O. Nierstrasz, "A Formal Language for Composition," in *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Editors, Cambridge University Press, New York (2000), pp. 69–90.

48. C. Pahl, "A Formal Composition and Interaction Model for a Web Component Platform," *Electronic Notes in Theoretical Computer Science* **66**, No. 4, 1–15 (2002).

49. R. Alur, T. A. Henzinger, and O. Kupferman, "Alternating-Time Temporal Logic," *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, Miami Beach, FL (1998), pp. 100–109.

50. X. Fu, T. Bultan, and J. Su, "WSAT: A Tool for Formal Analysis of Web Services," *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, Boston, MA (2004), pp. 510–514.

51. G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley Professional, Boston, MA (2003).
52. M. Kovács and L. Gönczy, "Simulation and Formal Analysis of Workflow Models," *Proceedings of the 5th International Workshop on Graph Transformation and Visual Modeling Techniques*, Vienna, Austria (2006), pp. 215–224.
53. H. Foster, S. Uchitel, J. Magee, and J. Kramer, "LTSA-WS: A Tool for Model-Based Verification of Web Service Compositions and Choreography," *Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China (2006), pp. 771–774.
54. C. Ouyang, H. M. W. Verbeek, W. M. P. van der Aalst, S. Breutel, M. Dumas, and A. H. M. ter Hofstede, "WofBPEL: A Tool for Automated Analysis of BPEL Processes," *Proceedings of the 3rd International Conference on Service Oriented Computing*, Amsterdam, The Netherlands (2005), pp. 484–489.
55. N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg, "Analyzing Interacting BPEL Processes," *Proceedings of the 4th International Conference on Business Process Management*, Vienna, Austria (2006), pp. 17–32.
56. J. Koehler, G. Tirenni, and S. Kumaran, "From Business Process Model to Consistent Implementation: A Case for Formal Verification Methods," *Proceedings of the 6th IEEE International Enterprise Distributed Object Computing Conference*, Lausanne, Switzerland (2002), pp. 96–106.
57. H. M. W. Verbeek, B. F. van Dongen, J. Mendling, and W. M. P. van der Aalst, "Interoperability in the ProM Framework," *Proceedings of the CAiSE '06 Workshops and Doctoral Consortium*, Luxembourg, Luxembourg (2006), pp. 619–630.
58. A. Del Bimbo, L. Rella, and E. Vicario, "Visual Specification of Branching Time Temporal Logic," *Proceedings of the 11th IEEE International Symposium on Visual Languages*, Darmstadt, Germany (1995), pp. 61–68.
59. M. Brambilla, A. Deutsch, L. Sui, and V. Vianu, "The Role of Visual Tools in a Web Application Design and Verification Framework: A Visual Notation for LTL Formulae," *Proceedings of the 5th International Conference on Web Engineering*, Sydney, Australia (2005), pp. 557–568.
60. A. C. Rao, A. Cau, and H. Zedan, "Visualization of Interval Temporal Logic," *Proceedings of the 5th Joint Conference on Information Sciences, Kaohsiung*, Taiwan, ROC (2000), pp. 687–690.
61. W. M. P. van der Aalst and M. Pesic, "DecSerFlow: Towards a Truly Declarative Service Flow Language," *Proceedings of the 3rd International Workshop on Web Services and Formal Methods*, Invited Talks, Vienna, Austria (2006), pp. 1–23.

Accepted for publication October 24, 2006.
Published online May 2, 2007.

Ying Liu
IBM Research Division, IBM China Research Laboratory, Diamond Building, ZGC Software Park No. 19, Dong Beiwang Road, ShangDi, Beijing, 100094, Peoples' Republic of China (aliceliu@cn.ibm.com). Dr. Liu is a research staff member in the Service Ecosystem Department at the IBM China Research Laboratory. She received a Ph.D. degree in applied mathematics from Peking University. She subsequently joined the IBM Research Division in Beijing, China, where she began working on business process integration. From 2003 to 2005 she focused on BPM, including process model verification and model-driven solution engineering management. Dr. Liu's current research interests include formal methods, BPM, and service building technologies.

Samuel Müller
IBM Research Division, IBM Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (sml@zurich.ibm.com). Mr. Müller obtained an M.S. degree in computer science and an M.A. degree in economics, both from the University of Zurich. He joined IBM Research in Zurich in 2004, where he is currently doing research in the area of risk and compliance. In parallel, he is working toward his doctorate degree as an external Ph.D. student at the Swiss Federal Institute of Technology Zurich, where he is a member of the Information Security group. Mr. Müller's research interests include modal logics, formal methods and modeling, risk and compliance management, game theory, and economics.

Ke Xu
Automation Department, Tsinghua University, Beijing, 100084, Peoples' Republic of China (xk02@mails.tsinghua.edu.cn). Mr. Xu received a B.S. degree from the Automation Department of Shanghai JiaoTong University. He is currently a Ph.D. candidate at the National Engineering Research Center for Computer Integrated Manufacturing Systems in Tsinghua University. His main research interests include process algebra and model checking and their application in grid computing and business integration. He serves as a member of the academic committee in the Automation Department of Tsinghua University. Mr. Xu is also an IBM Ph.D. Fellow for 2006–2007. ■