RZ 3679 (# 99689) 11/27/2006
Computer Science 33 pages

Research Report

A Static Compliance Checking Framework for Business Process Models

Y. Liu

IBM China Research Lab
Beijing, China
aliceliu@cn.ibm.com

S. Müller

IBM Research
Zurich Research Laboratory
8803 Rüschlikon, Switzerland
sml@zurich.ibm.com

K. Xu

Department of Automation
Tsinghua University
Beijing, China
xk02@mails.tsinghua.edu.cn

A Static Compliance Checking Framework for Business Process Models

Y. Liu[a], S. Müller[b], K. Xu[a,c]
(aliceliu@cn.ibm.com, sml@zurich.ibm.com, xk02@mails.tsinghua.edu.cn)
[a]IBM China Research Lab, Beijing, China
[b]IBM Zurich Research Lab, Switzerland
[c]Department of Automation, Tsinghua University, Beijing, China

Abstract. 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 compliance of process models against compliance rules are becoming important. In this paper, we present a method for improving the reliability and minimizing the risk of failure of business process management systems from a compliance perspective. The proposed method allows for the 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 non-compliant processes. Furthermore, it reduces the cost associated with inspecting business process models for compliance. Finally, compliance checking may guarantee compliance of new process models before their execution and thereby increases the reliability of business operations in general.

Keywords: Model Transformation, Compliance, Compliance Rule Modeling, Business Process Modeling, Model Checking

1. 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 so-called business process management systems (BPMS) in order to efficiently support, execute, and monitor business processes.

The above development has been paralleled by a growing amount of regulatory requirements imposed on businesses. Prominent examples in the U.S. are the Gramm-Leach-Bliley Act [3], the Sarbanes Oxley Act (SOX) [4], and the USA Patriot Act [5]. While these U.S. regulations have a very broad coverage, also 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 all around the globe. Demonstrating compliance with legal requirements and international standards generally requires that affected companies document their business processes. While 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, or 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 nowadays recommend the use of respected standards such as COBIT [9] and ITIL [10] for the implementation of an enterprise's IT systems. These standards consist of well-defined abstract process definitions that can tailored according to a company's individual needs.

Because of the increasing amount of relevant 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 often necessitating changes to the structure of particular work flows 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.

While business processes 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 cannot be assessed using these frameworks. For large enterprises with thousands of business processes deployed on the BPMS and stored in specific repositories, however, the assessment 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 non-compliant business processes before their execution and indicates the nature of the problem in case of non-compliance.

1.1 Potential Benefits from Automatic Verification of Business Process Models

To ensure compliance, the impact of each new regulatory requirement on the existing business process models needs to be identified. While BPM does not help here, our approach indicates which processes are compliant and which are not, hence providing a valuable tool for ensuring that

new requirements are incorporated into the companies' 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 the efficiency during deployment, and lowers the risk of implementing and activating non-compliant 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 guarantees compliance of new models before their execution and thereby increases the reliability of business operations in general.

1.2 Case Study

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

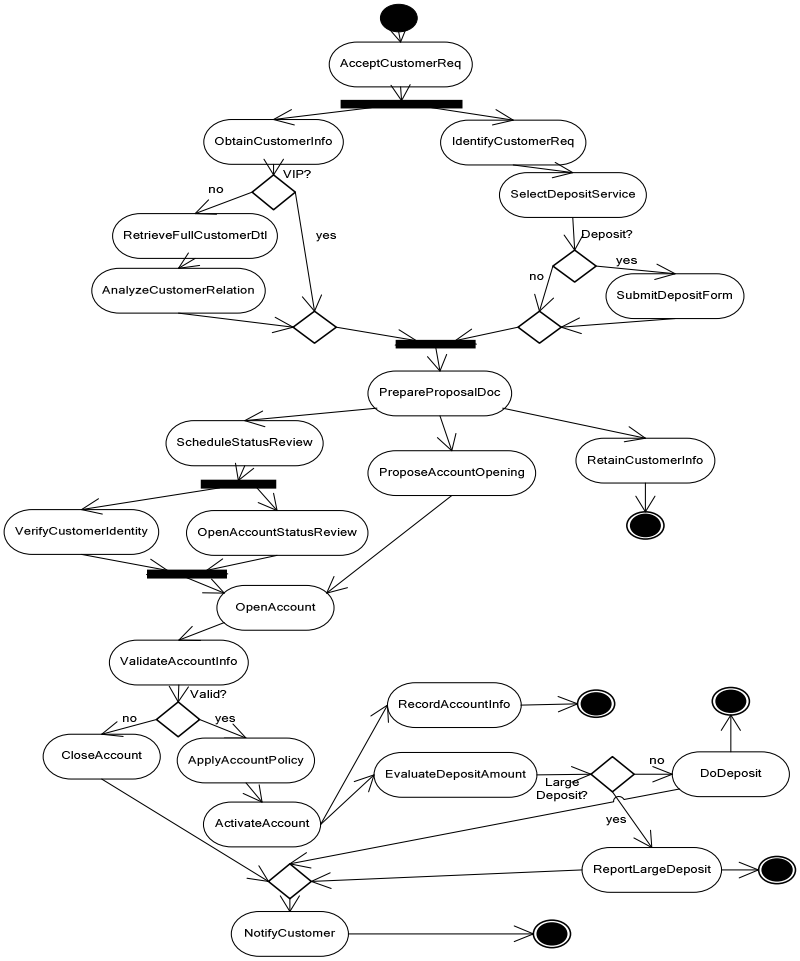


Figure 1. Account Opening Process

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, *Article 11* and *Article 13*, read as follows:

Article 11: *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. […]*

In other words, for financial institutions, Article 11 implies the following: Whenever a financial institution opens a deposit account or provides settlement service for an individual customer, it must verify the ID of the customer and record both the ID and the name of the customer.

Article 13: *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.

Article 13 implies: Financial institutions must report to the People's Bank of China and/or the State Administration of Foreign Exchange any large-value transactions that are detected in the process of providing financial services to customers. 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 that this is a parameter that can be flexibly adjusted.

Given a business process in Figure 1 and a set of compliance rules, in this paper, we demonstrate 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*.

1.3 Overview of the Compliance Checking Method

Overall, our compliance checking method includes six major steps as shown in Figure 2.

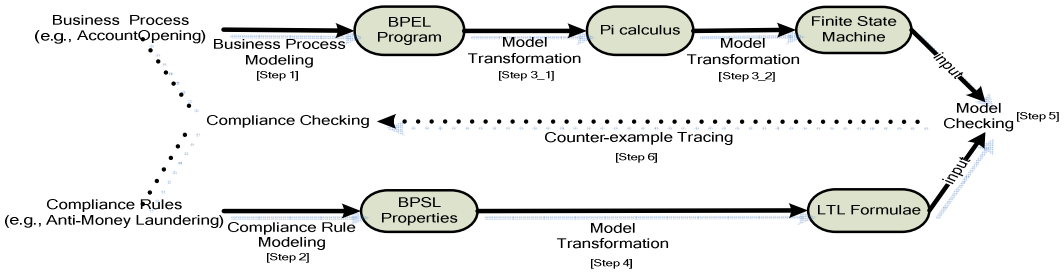


Figure 2. Compliance Checking Method

In step 1, we model our business processes using the Business Process Execution Language (BPEL) [14]. In the second step, we use the visual Business Property Specification Language (BPSL) [15] to specify relevant compliance rules. In section 2, we provide both the BPEL process model and the compliance rules formalized in BPSL for the *SimpleBank* case study introduced above. We transform the BPEL process model into a representation using Pi calculus [16] in step 3_1. Then, the Pi calculus is transformed into a Finite State Machine in step 3_2. In step 4, the BPSL compliance rules are transformed into LTL (Linear Temporal Logic) [17]. The model transformations of steps 3_1, 3_2, and step 4 are described in section 3. Having thus formalized both business processes and compliance rules, we use model checking technology [18] to statically verify whether the business processes comply with the imposed regulation in step 5. In step 6, counter-examples are fed back to the business process layer to demonstrate how the compliance rules have been violated. Details about the model checking, counter-example tracing, and specific optimization approaches for compliance checking are presented in sections 4 and 5.

2. Business Process Modeling and Compliance Rule Modeling

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.

2.1 Business Process Modeling using BPEL

In the previous section, we have introduced a conceptual account opening process for *SimpleBank*. Assuming that *SimpleBank* wants to take advantage of a BPMS to manage this process, the process should be specified using an executable business process modeling language. BPEL is such a language. It is an XML-based de-facto standard for business application integration and business-to-business processing with a specific focus on web services. It synthesizes essential aspects of WSFL (e.g., support for graph-oriented processes) [19] and XLANG (e.g., structural constructs for processes) [20] into one cohesive language to support implementing business processes in a natural manner. While 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 WSDL (Web Service Definition Language) 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 the normal behavior with BPEL activities, including *BasicActivity, StructuredActivity,* and *ScopeActivity*. The BPEL program corresponding to the introduced account opening process introduced earlier is given in Listing 1. It only shows the behavior definition section, whose essence should be intuitive to comprehend without further explanations. The precise semantics of BPEL activities and variables are explained in section 3.

As writing BPEL code manually is relatively cumbersome, process designers often model business processes visually 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 for exporting graphical process models as BPEL programs. We do not focus on the transformation from UML to BPEL here and refer to Mantell [22] for more detailed information. (The visual process modeling language provided by WBI Modeler is partially compatible to the UML activity diagram notation.)

Part 1

```
<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>
```

Part 2

```
      <bpws:invoke inputVariable="variable_pac" name="ProposeAccountOpenning"
                   outputVariable="variable_pac" partnerLink="PAC" ……/>
      <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>
```

Listing 1. BPEL Program of Account Opening Process

2.2 Compliance Rule Modeling using BPSL

Temporal constraints in compliance rules can be formally specified with temporal logic formulae, such as LTL (Linear Temporal Logic) and CTL (Computation Tree Logic). However, for people without a background in formal logics, temporal logics are rather difficult to understand and use. The purpose of the BPSL is to provide a more intuitive formalism to 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: First of all, obscure logical operators are replaced with an intuitive visual notation. Second, recurring logical patterns from a business or regulatory domain are defined as dedicated operators. Third, domain-specific templates (e.g., the property patterns [23]) can be predefined and re-used in BPSL to help increase the efficiency of property specification. Fourth, BPSL has a

direct semantic interpretation in both LTL and CTL. In this paper, we focus more on LTL because we agree on the argument by Vardi et al. [24] that the branching-time formalism of CTL is unintuitive for business analysts and it does not support compositional reasoning as opposed to LTL. The complete syntax, semantics, and notation of BPSL have been described by Xu [15]. To ease understanding, in Figure 3 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 section 3.

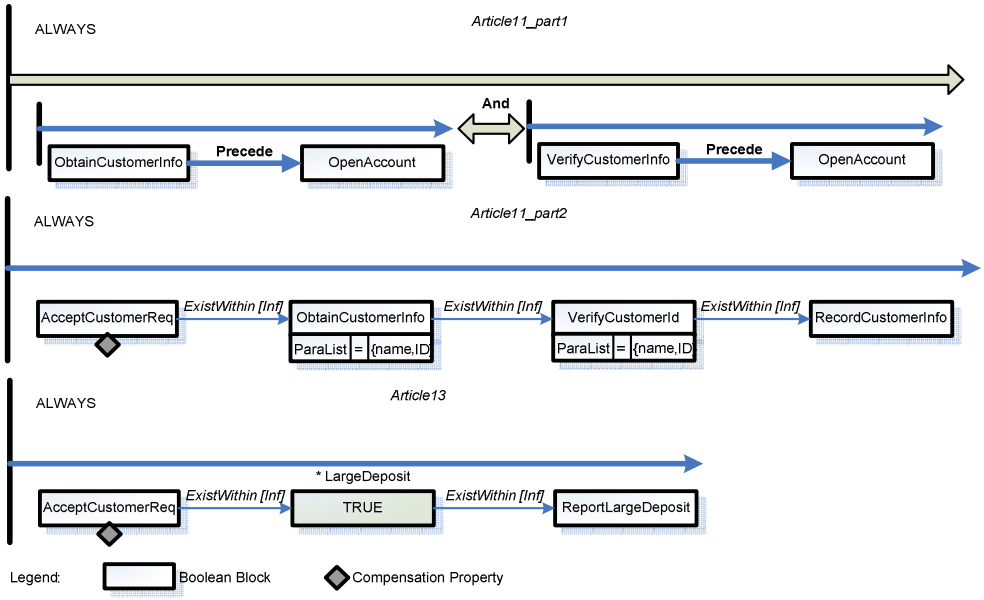


Figure 3. BPSL Specification of Article 11 and Article 13

In Figure 3, 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*, *VerifyCustomerInfo*, etc) 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 be 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 one which consists of two other simple temporal sequences that are combined by a logical *And* relation).

The BPSL properties in Figure 3 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 information must be later verified with the provided customer name and ID. Finally, the customer information must be recorded

(*Article11_part2*). Whenever a customer request for opening deposit account is received, it must be checked whether it is a large value deposit. If it is, it must be later reported.

In previous work, Giblin et al. have developed REALM (Regulations Expressed as Logical Models) [12], a meta-model 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 specification language in our static compliance checking work.

3. 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 Finite State Machine (FSM) (see steps 3_1 and 3_2 in Figure 2). Then, we show how BPSL properties are transformed into LTL formulae (see Step 4 in Figure 2).

3.1 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 overview introduction to Pi calculus.

3.1.1 Pi Calculus

Pi calculus [16] is a model of concurrent communicating processes which allows for the modeling of complex communication patterns. We have chosen Pi calculus as the formal method to formalize BPEL programs. A detailed discussion justifying this choice can be found in section 6.

The syntax of Milner's polyadic Pi calculus is as follows.

$$P ::= \sum_{i=1}^{n} \pi_i.P_i \mid new\ x\ P \mid P|Q \mid !P \mid \phi P \mid A(y_1,\cdots,y_n) \mid 0$$
$$\pi_i ::= \overline{x} < y > \mid x(y) \mid \tau$$
$$\phi ::= [x = y] \mid \phi \wedge \phi \mid \neg\phi$$

The simplest entities of Pi calculus are *names* (denoted with lowercases) and *processes* (denoted with uppercases). Processes can evolve by performing *actions*. Syntactically, $\overline{x} < y >$ denotes an *output* action which 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* which expresses un-observable behavior. A *sum*

$\pi_1.P_1 + \pi_2.P_2 + \cdots + \pi_n.P_n$ denotes a non-deterministic choice of process execution. In the *restriction* $new\ x\ P$, the scope of name x is bound to P. In the *composition* $P \mid Q$, the processes P and Q can proceed independently and can interact via shared names. The *replication* $!P$ can be thought of as an infinite composition $P \mid P \mid P \mid \cdots$ of processes. Finally, ϕP represents a process that is guarded by a Boolean expression ϕ evaluated by name *matching*.

3.1.2 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 only present the important basic activities and structured activities below.

BPEL contains program variables to which values can be assigned. We adopt the approach proposed by Jacobs et al. [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 ($=_{def}$ is used as a definition symbol):

$$Variable(x) =_{def} \mathrm{Re}g(x)$$
$$\mathrm{Re}\,g(x) =_{def} put(y).\mathrm{Re}\,g(y) + \overline{get} < x > .\mathrm{Re}\,g(x)$$

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

BPEL's basic activities (*Receive*, *Reply*, *Invoke*, *Assign*, *Empty*, and *Termination*) define how message communication, service invocation and variable assignment are done in a process model. We present the formalization of these basic activities in the following. '*Link name*', '*Partner name*' and '*Operation name*' are three elements in the activities of BPEL and they often appeared 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*', '*put*' in Pi calculus. In addition, two special names '*start*' and '*done*' are used to indicate common internal communications in a BPEL process.

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

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 have discussed how to solve these problems in another paper [25]. Because they are not very critical to understand our compliance checking method, we will not cover them in this paper. We take $Receive(start, \ell, put, done)$ as an example to explain the semantics of the Pi calculus formalization. The process of *Receive* contains some free names (e.g. *put*, *done*, etc), which are defined as the communication channels of this process. The two 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 of $\overline{done}$ 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 below.

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

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 non-deterministic 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 usage 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*:

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

$P_i\ (i = 1, \cdots, n)$ 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:

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

Since 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.

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

3.2 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 FSMs 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 leaves us the opportunity to apply other verification techniques like structural verification including (e.g., 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 will be 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 [27] and Pistore [28] have already shown the approach and feasibility of how to correctly transform finitary Pi calculus processes into the corresponding ordinary automaton 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 [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 (cf. Figure 4).

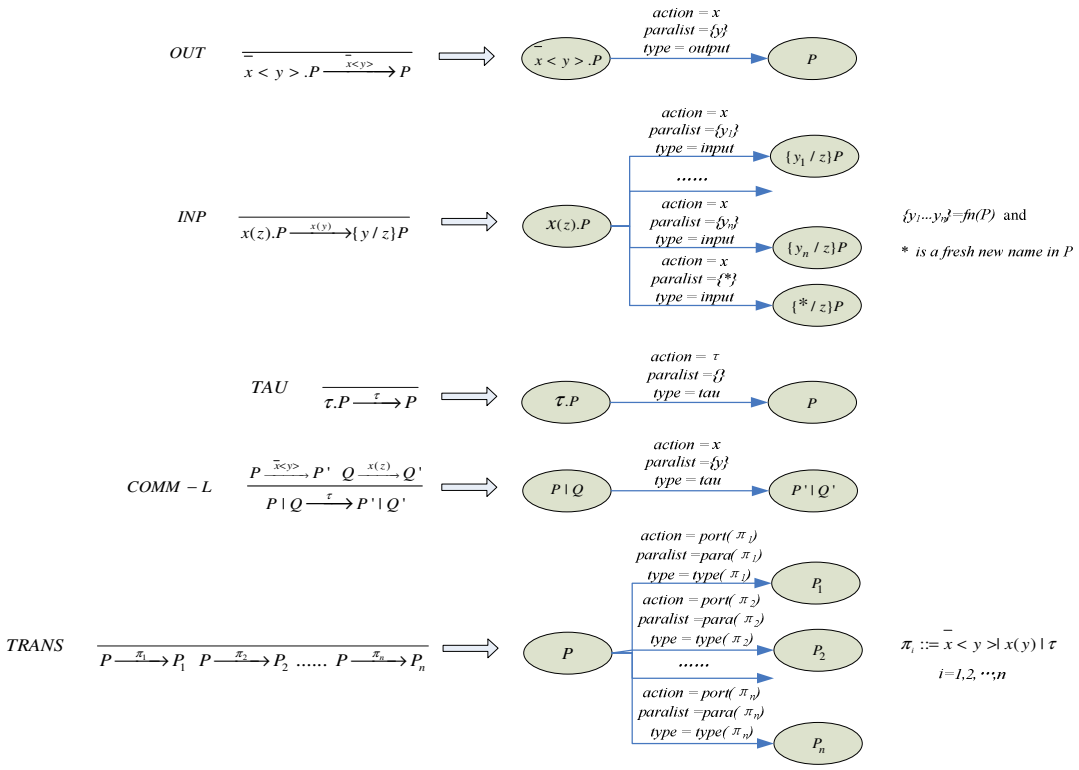


Figure 4. Transformation Rules from Pi Calculus to FSMs

For each transition fired 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 the following two advantages:

(1) While transforming from Pi process to FSM, two properties can be directly checked, deadlock and redundant activities of BPEL process. A deadlock exists if there is a Pi process that cannot generate its state space any more before it transits to an empty process ("0"). If there is a Pi process that can never communicate with any other Pi processes, it means there is a redundant activity in the corresponding BPEL process.
(2) Using Pi calculus and FSM as the formals 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.

3.3 BPSL to LTL

Though BPSL supports both the semantic mapping to LTL and CTL, in this section we mainly

explain the mapping from BPSL specifications to LTL formulae based on the three compliance rules in section 2.2 as examples. Before diving into the mapping from BPSL to LTL, an overview on LTL is given as preliminary information.

3.3.1 Linear Temporal Logic (LTL)

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

Linear Temporal Logic uses formulae of the form **A** *f*, where: 1) **A** is the universal path quantifier, which means that *f* has to be true on all possible paths in the future; and 2) *f* is a path formula (which is true along a specific path) in which the only state sub-formulae permitted are atomic propositions. More precisely:

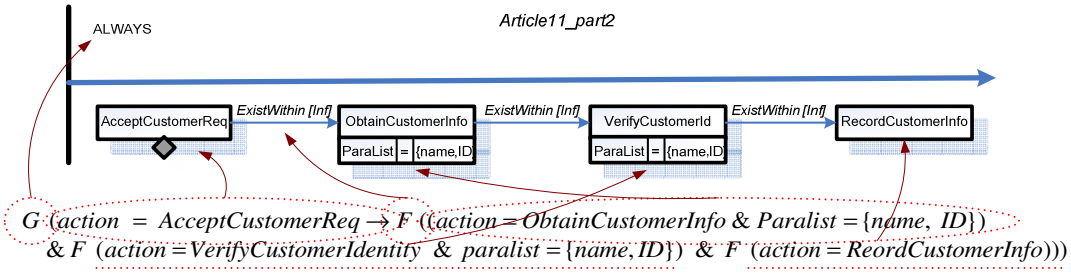
- If $p \in AP$ (*AP* is a non-empty set of atomic propositions), then *p* is a path formula.
- If *f* and *g* are path formulae, then $\neg f$, $f \vee g$, $f \wedge g$, **X** *f*, **F** *f*, **G** *f*, *and f* **U** *g* are path formulae.

The four basic operators of X, F, G and U are explained informally below.

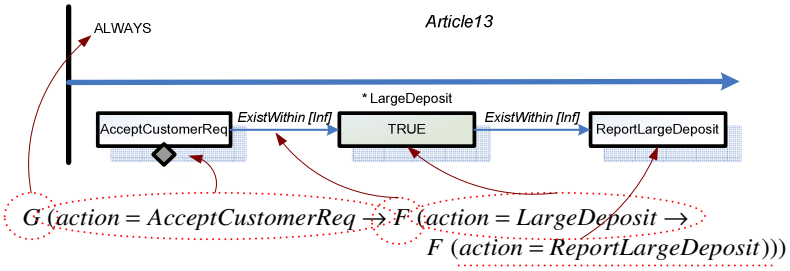
- X ("next time"): a formula (*f*) is true in the second state of the path;
- F ("eventually"): a formula (*f*) will be true at some future state on the path;
- G ("always"): a formula (*f*) is true at every state on the path;
- U ("until"): there is a state on the path where the second formula (*g*) is true, and at every proceeding state on the path the first formula (*f*) is true;

Take formula of "*G* (AcceptCustomerReq $\rightarrow$ *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.

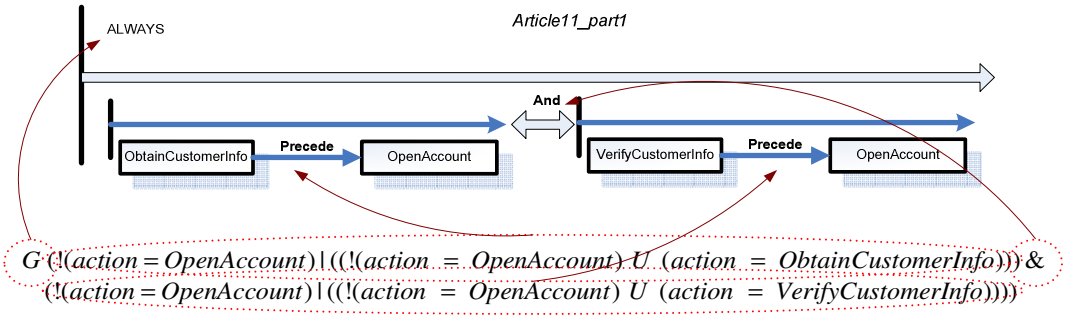
3.3.2 Mapping from BPSL to LTL



(a) Mapping from BPSL to LTL for Article11_part2



(b) Mapping from BPSL to LTL for Article13



(c) Mapping from BPSL to LTL for Article11_part1

Figure 5. Mapping BPSL to LTL

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 where 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*' while 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'. A Compensation Property (*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 short cut to the Compensation Property being 'True'. It specifies that *Article11_part2* will only be evaluated when a customer request is accepted (because it is always ok 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 never hold at all. Different Boolean Blocks are associated with the Temporal Operators in BPSL. For example, in Figure 5(a) *ExistWithin [inf]* specifies that after a customer request is accepted (*AcceptCustomerReq*), the customer information must be obtained including the customer name and ID. Here *inf* is the scope parameter for the operator *ExistWithin* with the value of *infinity*. 17 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 *X X X*, *AllWithin [inf]* for *G*, etc), 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 *Article11_part2* is:

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

The mapping of this formula to the BPSL notations is also illustrated in Figure 5(a).

As to *Article13*, two new elements need to be introduced. The rectangle TRUE indicates a Boolean Block which 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:

$$G\ (action = AcceptCustomerReq \rightarrow F\ (action = LargeDeposit \rightarrow F\ (action = ReportLargeDeposit)))$$

The mapping of this formula to the BPSL notations is also illustrated in Figure 5(b).

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

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

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 also illustrated in Figure 5(c). The above mapping follows exactly the semantics of BPSL [15] and thus ensures its correctness.

4. Compliance Checking Framework and Case Study Results

Having the above introduced FSM and LTL, model checking technology can be used to verify whether the business process complies with the compliance rules. We firstly briefly explain the basic concepts of model checking. We then introduce our compliance checking framework. Finally, we present a case study result to illustrate the practical feasibility of this framework.

4.1 Model Checking

Model checking [18] is an automatic technique for verifying finite state systems, which has been successfully applied to the verification of hardware designs, communication protocols, etc. The main idea of model checking is to search the state space of a system model and to verify whether 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 from users. In fact, the procedure of model checking needs little user intervention, 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, our 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 a Finite State Machine (FSM) expressing the operational semantics of Pi calculus. BPEL and Pi calculus have already been introduced in the previous sections.
2. *Specifying the properties that the formal system model is expected to satisfy.* Often, Linear Time Logic (LTL) is used to capture such specifications. As writing correct statements in temporal logic is relatively difficult, for our compliance checking method, we have developed the Business Property Specification Language (BPSL), a visual property specification language, to formally express such properties. BPSL properties are then automatically translated into LTL formulae.
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 BDD-based symbolic model checking algorithm implemented in NuSMV2 [30]. The validity and effectiveness of the algorithm has already been proved from both academic and industrial sides. In addition, OPAL also provides a *counter-example* tracing capability for business process models and offers its own optimization approach for compliance checking.

4.2 Compliance Checking Framework

Figure 6 provides an overview of the Open AnaLyzer (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 a 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]) via the *Process Model-to-Pi Transformer* and the *Model Checker Adapter* respectively. The component *Process Model-to-Pi Transformer* is responsible for generating 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 *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.

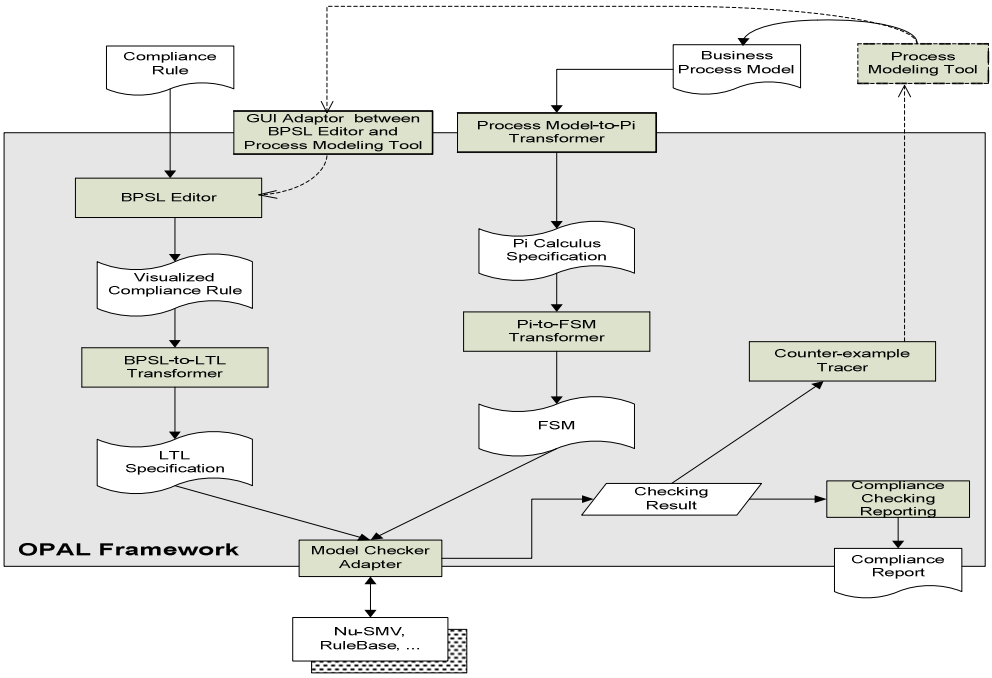


Figure 6. Compliance Checking Framework

As introduced previously, we use Pi calculus as the formal 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* have already been introduced in the previous sections. The *Model Checker Adapter* is used to integrate existing model checking engines such as NuSMV2 and Rule Base with OPAL. The framework provides an additional capability to trace counter-examples in the business process model, called *Counter-example Tracer*. Further, compliance checking results can be generated as compliance reports using the *Compliance Checking Reporting* module.

Since this framework is intended to be used by business people for business process model checking, two points must be addressed. The first one 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. *Counter-example Tracer* can help solve this problem. The other aspect is how to guarantee 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 features of OPAL are discussed further below.

4.3 Running Example with Results

Let's now recall the case study about *SimpleBank*'s account opening process that was described in section 1. It has been used as a running example throughout the entire paper to illustrate our compliance checking method. We have applied OPAL to check the compliance of the account opening process in Listing 1 against the compliance rules defined in Figure 3. As explained in section 3, the account opening process was automatically transformed to Pi calculus, and further into a FSM with the help of OPAL. Likewise, the regulatory requirements of Article 11 and Article 13 were formalized as compliance rules using OPAL's BPSL modeler, which were then automatically transformed into LTL.

OPAL has been developed as a plug-in for the Eclipse platform, which allows for the integration with different business process modelers like WBI Modeler and other Eclipse-based BPEL editors [32]. We tested our case study using OPAL on a Windows platform with an Intel P4 processor, 3.0MHz, and 2.5GB 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 section 3. The time consumed for transforming the Pi calculus formalization into its corresponding FSM was 49.959 seconds. The final FSM contained 11832 (i.e., 2^13.5304) reachable states out of 535840 (i.e., 2^19.0314) states after OPAL applied its *sequentialization of interleaving actions* optimization to compact the FSM, which is introduced further below. The checking of the three compliance rules in Figure 3 consumed 121.0 seconds of CPU time. This also included an additional 21.0 seconds for generating the needed counter-examples for the violated compliance rules. The peak memory cost 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 counter-example 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 where the customer information was already recorded in the banking system before it has been 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 introduced in the introduction. Thus, OPAL users realize that the current account opening process is non-compliant before deployment, which helps increase the deployment efficiency and lowers the risk of installing non-compliant 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.

5. Advanced Features of the Compliance Checking Framework

Two important features of the framework, counter-example tracing and performance optimization, are addressed in this separate section.

5.1 Counter-example tracing

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

Since the previous counter-example in our running example is too large (a state trace of 54 states), we now present a simpler example to explain how OPAL's counter-example tracing mechanism works. The BPEL program of the account opening process in Listing 1 is simplified to the BPEL program *SimpleAC* given in Listing 2:

```
<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>
```

Listing 2. BPEL Program of SimpleAC

This simple BPEL process can be transformed to the following Pi process using the BPEL-to-Pi transformation rules introduced in section 3 (For simplification, some channel names have been shortened, for example, 'AcceptCustomerReq' is abbreviated as 'ACR'):

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

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

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

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

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

$$Partner\ AcceptCustomerReq = \overline{ACR} < req >$$

$$Partner\ ObtainCustomerInfo = OCI < w > .\overline{OCI} < Info >$$

$$Partner\ VerifyCustomerInfo = VCI < w > .\overline{VCI} < Info >$$

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

In the above formalism, *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 in section 3. In the same section, we have also introduced how to transform a Pi process to a FSM. Accordingly, the above Pi process can be transformed into the FSM shown in Figure 7 with the help of OPAL.

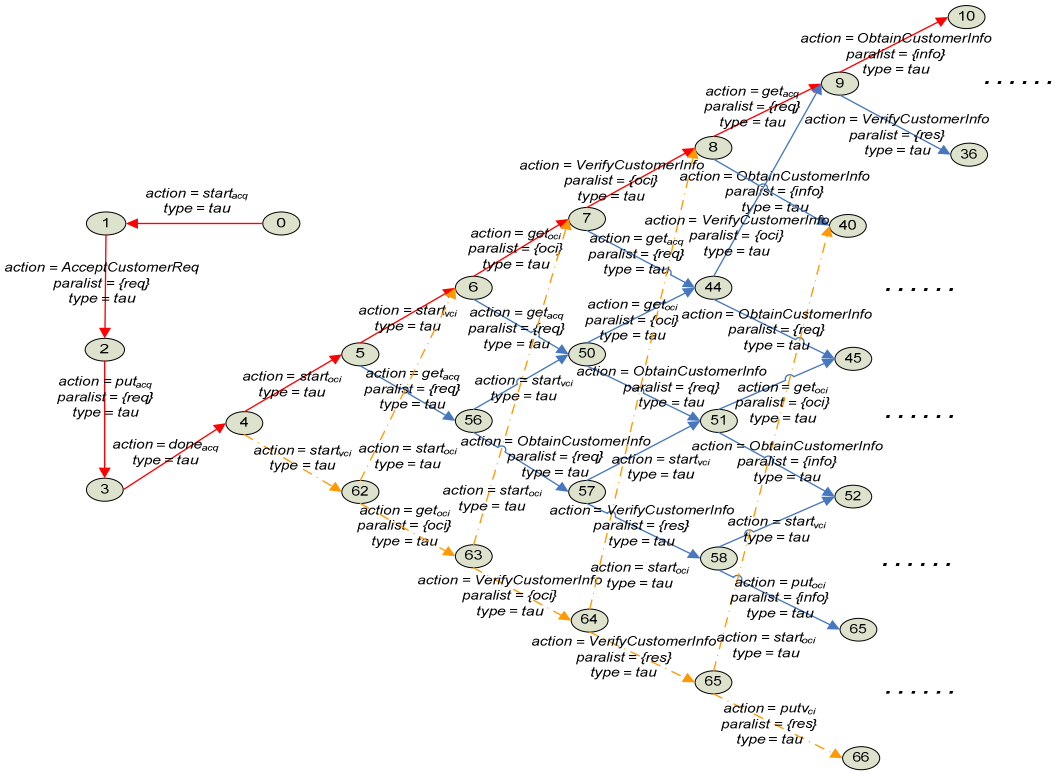


Figure 7. FSM Example

The above diagram only shows part of the FSM of the entire *SimpleAC Process* model, which contains totally 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.
$G(!(action = VerifyCustomerInfo)) \mid (!(action = VerifyCustomerInfo) \cup (action = ObtainCustomerInfo))$
OPAL visualizes the transformed FSM and the counter example generated by NuSMV2. The counter-example is indicated by the state transitions in red lines in the above diagram, showing a possible execution path in *SimpleAC Process*, where the customer information may be verified before it is obtained in the first place.

Since the counter-example in the above diagram is given at a state machine level and contains many redundant action information (e.g., $start_{acq}$, *ack, etc*), it is hard for business people to comprehend what the counter-example stands for from a business perspective. Therefore, OPAL provides a counter-example mapping from the FSM to the BPEL process model. Such mapping is implemented by preserving only the actions that have a directly connection with the corresponding BPEL activities (e.g., *AcceptCustomerReq, ObtainCustomerInfo, VerifyCustomerInfo*) and remove any other redundant information in the counter-example. Then the mapped counter example in BPEL is shown as given below:
Invoke AcceptCustomerReq -> Invoke VerifyCustomerInfo -> Invoke ObtainCustomerInfo

5.2 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 mainly on the business process level with its own optimizations. Specifically, 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 a 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 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.

5.2.1 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, which will 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 a 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, i.e., $\overline{start}_{oci} | \overline{start}_{vci}$ in the formalism of *Sequence*, since these two activities are executed independently and in an arbitrary order. However, it is easy to note that $\overline{start}_{oci}$ and $\overline{start}_{vci}$ only play the role of triggering the execution of two Pi processes but do not affect the execution order of the two activities *ObtainCustomerInfo* and *VerifyCustomerInfo* at all. 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 (i.e., the retrieval and assignment of variables via *Get* and *Put*, the invocation of *PartnerLinks* ℓ, etc.) 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 a 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 a FSM. Typical sequentializations are implemented in OPAL including the formalism of the *Fork* nodes, *Join* nodes, multiple inputs/outputs for an activity in the UML activity diagrams and compatible models, the *Flow* structure, multiple incoming/outgoing links for an activity in BPEL models, etc. Note that 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. The states (ID 62-65) and their transitions in brown color and dashed lines in figure 7 show the part of the FSM that can be reduced due to the 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.

5.2.2 Business bug patterns guided state searching

Despite great improvements regarding the performance of model checking, the exploitation of domain knowledge is crucial to further improve the efficiency of compliance checking. Since model checking is more useful to probe hidden bugs in a system than to prove its correctness, we have developed *business bug patterns*, i.e., a set of anti-patterns 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], let's 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) ⋁NoResponse(A, B)

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

/*After 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} |->{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 *NoResponse(A, B)* respectively. The symbol '⋁' indicates that the *SequentialBug* holds when either *SimultaneousStart* or *NoResponse* is satisfied. Here, the *SequentialBug* does not necessarily check whether *A* is possibly executed after *B* since 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*, *Exit*, etc.) 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 8 shows an example of such mapping for the *Receive* activity.

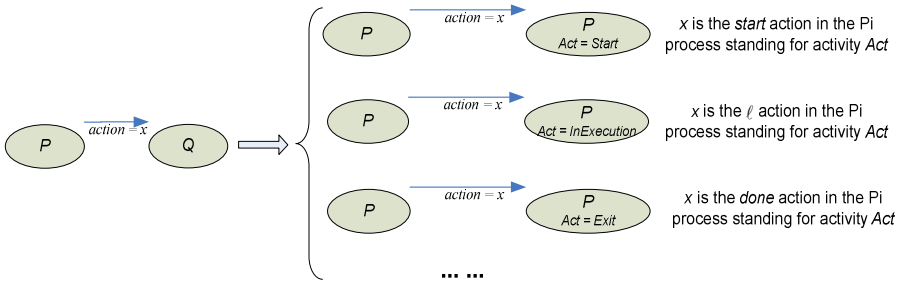


Figure 8. Mapping Example from Pi calculus to FSM

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

In order 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. The *interesting states* should be the states that can lead to the detection of a targeted business bug pattern within the least number of transitions.

More specifically, we define:

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

S(m)={s(act1),s(act2),…}: A state in *M(m)* which 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 need 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)= Σi *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 experiment results on the compliance checking of the account opening process in Figure 1, which has a total state space of 8361 (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 P4 processor, 3.0MHz, and 2.5GB RAM.

Table 1: Experiment Result on Checking Account Opening Process

Target Bug	OA		*GBH*	Result
	From Pi to FSM	Model Checking		
B1	57.218 s	118.200 s	2.031 s	Found
B2		112.700 s	41.703 s	Not Found
B3		177.600 s	26.438 s	Found
B4		129.300 s	60.469 s	Found
B5		221.900 s	72.266 s	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) && PallelSplitBug(PrepareProposalDoc, {VerifyCustomerIdentity, ProposeAccountOpening}) && SequentialBug(ProposeAccountOpening, ActivateAccount) && SynchronizingMergeBug({ActivateAccount, VerifyCustomerIdentity }, DoDeposit)*

(OA: Original Approach; GBH: *GuidedBugHunting*)

(*Mlv* = 1 and *Gate* = 10)

The experiment 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 activity status and always following 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 for checking the compliance of complex large-scale business processes. However, the approach is not perfect: its merits come at some cost too:

1. It can only be used to falsify a business process because it does not guarantee the full traversal of state space of the business process. Since 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 which are too large for classical model checking to run to completion.
2. It is not suitable for the application in small scale business processes. On the one hand, it will be 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.

6. 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 have been a lot of works with respect to the modeling of business processes and developing verification techniques and tools for them. A recent survey is done by Breugel et al [37]. In this section, we identify three aspects pertaining to the compliance checking of business process models. We will report on the literature in each area and clarify the difference of our work.

6.1 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 like BPEL [38]. "*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 Petri-net and Pi calculus. However, this has also lead to a long debate on what is the most suitable formal foundation for business process models. Smith et al support the Pi calculus and argued that "workflow is just a Pi calculus" [40]. The design of web service composition languages like XLANG and BPEL is also claimed to be based on Pi calculus. Van der Aalst, on the other hand, appealed that more solid works should be done in order to prove the effectiveness of Pi calculus in modeling business process [38]. As a matter of fact, there have been some previous works on formalizing various business process models with Pi calculus including UML statechart diagram [41,42], UML(2.0) activity diagram [43,44], workflow patterns [45], etc. Previous work has also shown that Pi calculus is a suitable formal composition language for software composition and web service composition [46,47,48]. In our work, we have formalized BPEL process models with Pi calculus, instead of using Petri-nets or Automata (and their extensions). We rely on Pi calculus for the following reasons:

(1) Automata and Petri-nets are often used to model *closed systems*, whose behavior is completely determined and controlled by the state of the system. However, the 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 the interaction with the behavior of the environment [29,49]. For example, FSM model is under complete control over its transitions, while 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.

In the case of BPEL, although it 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 / mapping of the target services for invocation, it becomes critical to consider the communications with the environment information like the available service candidates in the service portfolio, service selections (which can be best modeled by the mobility feature of Pi calculus), etc.

(2) 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 sub-components. This operator does not exist in Petri-nets. 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 Petri-net, which will involve additional operations and computations for the composition.

(3) 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 like 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.

6.2 Formal Verification of BPEL Process Models

Based on the formal semantics, there have been some previous works on the formal verification of BPEL process models. Fu et al. [50] first translates 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 [51]. Besides, Fu et al. [50] studied the so-called synchronizability and realizability analysis for the composition of web services. Kovács et al. [52] also exploit the model checker of SPIN, although the intermediate model between BPEL and Promela is a model of dataflow network. Foster et al. [53], on the other hand, take the BPEL process and translates it into the form of Finite State Process (FSP) calculus and then compile it into a Labeled Transition System (LTS). The formal verification is then performed by the existing Labeled Transition System Analyzer (LTSA) tool suite. Ouyang et al. [54] and Lohmann et al. [55] both provide a semantic transformation from BPEL to Petri-net. However, while Ouyang et al. [54] focuses on the analysis of specific process properties like reachability analysis, competing message-consuming activities and garbage collection of queued messages, Lohmann et al. [55] focuses on the controllability of the process, i.e., 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] proposed a pattern-based mapping approach to model a general business process. Two typical properties in a process model (i.e., reachability and

termination) are formulated with the temporal logic of CTL, which can be later verified 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 have already been addressed in the previous sub-section.
(2) The completeness of the approach is different. We not only focused on the verification of our formalized BPEL models against specific structural errors. More importantly, our work involves a more detailed proposal of subjects including counter-example guiding, performance enhancement and visualization of temporal logics, which are critical issues to make the formal verification of business processes really practical and usable.

6.3 Specifying Regulatory Rules with Temporal Logics

Specifying user-desired properties with logical formulae is an important step in the formal verification of business process models. The intuitiveness and convenience in the property specification thus becomes 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 like a predefined set of business entity types (e.g., *Artifact*, *Resource*, *Principle*, etc.) and relations (e.g. *Do*, *Input*, *Output*, etc.) 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 formula intuitively. Related visualization works can be found for commonly used temporal logics including CTL [58], LTL [59], Interval temporal logics [60], etc. 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 above, our Open Process AnaLyzer (OPAL) toolkit contains an editor for the Business Property Specification Language (BPSL) to visually specify various regulatory rules. BPSL is different from the above works in the following aspects.
(1) It is a visual specification language which supports both the temporal logics of LTL and CTL. It is also compatible with the IEEE standard of the Property Specification Language [35].
(2) It enables the intuitive and convenient specification of regulatory rules by customizing predefined Property Templates in BPSL. The source of these templates comes from the existing works on Business Property Specification Patterns [23], Business Bug Patterns [34], etc.

7. 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 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 automatable transformations into low-level formalisms yields easier, more intuitive, and less error-prone process modeling, thus reducing the risk of implementation errors and non-compliant operations. If non-compliant business process models are discovered, counter-examples 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 non-compliant processes. Because of these capabilities, also business people can potentially 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 a 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 address the application of the framework in BPEL processes, our current implementation of the compliance checking framework (i.e., the OPAL toolkit) has also been applied in the verification of WebSphere Business Integrator process models. Since performance is always a critical problem in the area of model checking, we have also proposed the *sequentialization of interleaving actions* method to reduce the overall state space. The *business bug patterns guided state searching* approach 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.

As to future work, we intend to extend the existing compliance checking method to also support the verification of resource and data constraints that are related to the business process models. Additionally, we will 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.

Acknowledgements

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 on improving this paper.

References

1. F. Leymann and D. Roller, *Production Workflow: Concepts and Techniques*, Prentice Hall. (2000)
2. M. Havey, *Essential Business Process Modeling*, O'Reilly. (2005)
3. Gramm-Leach-Bliley Act of 1999 (GLBA), PL 106-102, 113 Stat. 1338. (1999)
4. Sarbanes-Oxley Act of 2002, PL 107-204, 116 Stat 745. (2002)
5. USA Patriot Act of 2001, PL 107-56, HR 3162 RDS. (2001)
6. The Revised Basel Capital Framework (Basel II), June 26. (2004); see *http://www.federalreserve.gov/boarddocs/press/bcreg/2004/20040626/attachment.pdf*
7. The Money Laundering Regulations. (2003); see *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. (2003); see *http://www.pbc.gov.cn/english//detail.asp?col=6800&ID=22*
9. Control Objectives for Information and Related Technology (COBIT), *IT Governance Institute*, Version 4.0. (2005); see *http://www.itgi.org*
10. IT Infrastructure Library (ITIL), *Office of Government Commerce (OGC).* (2006); see *http://www.itil.co.uk*
11. C. Abrams, J. von Känel, S. Müller, B. Pfitzmann, and S. Ruschka-Taylor, "Optimized Enterprise Risk Management", IBM Research Report RZ 3657, IBM Research Division, August 2006. To appear in: *Special Issue on Compliance Management, IBM Systems Journal, 46(2).* (2007)
12. C. Giblin, A. Y. Liu, S. Müller, B. Pfitzmann, and X. Zhou, "Regulations Expressed As Logical Models (REALM)", *Proc. 18th Annual Conference on Legal Knowledge and Information Systems,* IOS, pp. 37–48. (2005)
13. Rules for Anti-money Laundering by Financial Institutions. (2003); see *http://www.pbc.gov.cn/english//detail.asp?col=6800&ID=31*.
14. Business Process Execution Language for Web Services. (2003); see *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", *Proc. Graph Transformation and Visual Modelling Techniques*, Electronic Notes in Theoretical Computer Science, pp. 205-214. (2006)
16. R. Milner, "The polyadic Pi-calculus: A tutorial", *Technical Report*, CS-LFCS-91-180, University of Edinburgh. (1991)
17. A. Pnueli, "The temporal logic of programs", *Proc. 18th IEEE Symposium on Foundation of Computer Science*, pp. 46–57. (1977)
18. E. M. Clarke, Jr. Orna Grumberg, and D. A. Peled, *Model Checking*. MIT Press. (2000)
19. *Web Services Flow Language (WSFL 1.0).* (2001); see: http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf
20. *XLANG: Web Services for Business Process Design.* (2001); see: http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm
21. IBM WebSphere Business Integration Modeler. (2006); see *http://www-306.ibm.com/software/integration/wbimodeler/library/*

22. K. Mantell, "From UML to BPEL", *IBM DeveloperWorks*. (2005); see *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", *Proc. 2nd Workshop on Formal Methods in Software Practice*, pp. 7-15. (1998)
24. M. Y. Vardi, "Branching vs. Linear Time: Final Showdown", *Lecture Notes in Computer Science*, 2031, pp. 1-22. (2001)
25. K. Xu, Y. Liu, and G. G. Pu, "Formalization, Verification and Restructuring of BPEL Models with Pi calculus and Model Checking", *IBM Technical Report*, RC23962(C0605-012). (2006)
26. B. Jacobs and F. Piessens, "A Pi-Calculus Semantics of Java: The Full Definition", *Technical Report CW 355,* Department of Computer Science, K.U.Leuven. (2003); see *http://www.cs.kuleuven.ac.be/~frank/publications.htm.*
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(4), pp. 440–473. (2003)
28. M. Pistore, *History dependent automata*. Ph.D. Thesis, Dipartimento di Informatica, University of Pisa, TD-5/99. (1999)
29. D. Sangiorgi and D. Walker, *The Pi calculus: A Theory of Mobile Processes*, Cambridge University Press. (2001)
30. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: an OpenSource tool for symbolic model checking", *Lecture Notes in Computer Science*, 2404, pp. 359-364. (2002)
31. I. Beer, S. Ben-David, C. Eisner, D. Geist, L. Gluhovsky, T. Heyman, A. Landver, P. Paanah, Y. Rodeh, G. Ronin, and Y. Wolfsthal, "RuleBase: Model checking at IBM", *Proc. International Conference on Computer Aided Verification*, pp. 480-483. (1997)
32. *WebSphere Studio Application Developer Integration Edition (WSAD-IE).* (2006): See 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(3), pp. 5-51. (2003)
34. K. Xu, Y. Liu, and C. Wu, "Guided Reasoning of Complex E-Business Process with Business Bug Patterns", *Proc. International Conference on E-Business Engineering*, In Press. (2006)
35. D. Geist, "The PSL/Sugar Specification Language: a Language for All Seasons", *Proc. Correct Hardware Design and Verification Methods.* pp. 21-24. (2003)
36. C. H. Yang and D. L. Dill, "Validation with Guided Search of the State Space", *Proc. Design Automation Conference,* pp. 559-604. (1998)
37. F, V. Breugel and M. Koshkina, "Models and Verification of BPEL", *Technical Report*, York University. (2006); see 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(5), pp. 1-11. (2005)
39. OASIS WSBPEL TC, Issue 42, *Need for Formalization.* (2003)
40. H. Smith, "Business Process Management—The Third Wave: Business Process Modeling Language (BPML) and its Pi calculus Foundations", *Information and Software Technology*, 45,

pp. 1065-1069. (2003)
41. S.W.L. Vitus and P. Julian, "Formalization of UML statechart diagrams in the Pi calculus", *Proc. the 13th Australian Software Engineering Conference*, pp.213-223. (2001)
42. S.W.L. Vitus and P. Julian, "Analyzing Equivalences of UML Statechart Diagrams by Structural Congruence and Open Bisimulations", *Proc. IEEE Symposium on Human Centric Computing Languages and Environments*, pp. 137-144. (2003)
43. D. Yang and S.S. Zhang, "Using Pi-calculus to Formalize UML Activity Diagram for Business Process Modeling", *Proc. the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pp. 47-54. (2003)
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*, In Press. (2006)
45. F. Puhlmann and M. Weske. "Using the Pi calculus for Formalizing Workflow Patterns", *Lecture Notes in Computer Science*, 3649, pp. 153-168. (2005)
46. N. Oscar and D.M. Theo, "Requirements for a Composition Language", *Lecture Notes in Computer Science*, 924, pp. 147-161. (1995)
47. M. Lumpe, F. Achermann, and N. Oscar, "A Formal Language for Composition", *Foundations of Component-based Systems,* Cambridge University press, pp. 69-90. (2000)
48. C. Pahl, "A Formal Composition and Interaction Model for a Web Component Platform", *Electronic Notes in Theoretical Computer Science*, 66(4), pp. 1-15, (2002)
49. R. Alur, T. A. Henzinger, and O. Kupferman, "Alternating-Time Temporal Logic", *Lecture Notes in Computer Science*, 1536, pp. 23-60. (1997)
50. X. Fu, T. Bultan, and J. Su, "WSAT: A Tool for Formal Analysis of Web Services", *Lecture Notes in Computer Science*, 3114, pp. 510-514. (2004)
51. G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley, Boston, Massachusetts. (2003)
52. M. Kovács and L. Gőnczy, "Simulation and Formal Analysis of Workflow Models", *Proc. Graph Transformation and Visual Modelling Techniques*, Electronic Notes in Theoretical Computer Science, pp. 215-224. (2006)
53. H. Foster, S. Uchitel, J. Magee, and J. Kramer, "LTSA-WS: A Tool for Model-based Verification of Web Service Compositions and choreography", *Proc. 28th International Conference on Software Engineering*, pp. 771-774. (2006)
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", *Lecture Notes in Computer Science*, 3826, pp. 484-489. (2005)
55. N. Lohmann, P. Massuthe, C. Stahl, and D Weinberg, "Analyzing Interacting BPEL Processes", *Lecture Notes in Computer Science*, 4102, pp. 17-32. (2006)
56. J. Koehler, G. Tirenni, and S. Kumaran, "From Business Process Model to Consistent Implementation: A Case for Formal Verification Methods", *Proc. 6th IEEE International Enterprise Distributed Object Computing Conference*, pp. 96-106. (2002)
57. H.M.W. Verbeek, B.F. van Dongen, J. Mendling, and W.M.P. van der Aalst, "Interoperability in the ProM Framework", *Proc. CAiSE'06 Workshops and Doctoral Consortium*, pp. 619-630. (2006)
58. B. Del, A. L. Rella, and E. Vicario, "Visual Specification of Branching Time Temporal

Logic", *Proc. 11th IEEE Symp. on Visual Languages*, pp. 61-68. (1995)
59. M. Brambilla, A. Deutsch, L. Y. Sui, and V. Vianu. "The Role of Visual Tools in a Web Application Design and Verification Framework: A Visual Notation for LTL Formulae", *Lecture Notes in Computer Science*, 3579, pp. 557-568. (2005)
60. A. C. Rao, A. Cau, and H. Zedan, "Visualization of Interval Temporal Logic", *Proc. 5th Joint Conference on Information Sciences*, pp. 687-690. (2000)
61. W.M.P. van der Aalst and M. Pesic, "DecSerFlow: Towards a Truly Declarative Service Flow Language", *Proc. 3rd International Workshop on Web Services and Formal Methods*, Invited Talks, pp. 1-23. (2006)

Biographies:

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 her Ph.D. degree in applied mathematics from Peking University in 2003. She subsequently joined the IBM Research Division in Beijing, China, where she began working on business process integration related topics. From 2003 to 2005, Dr. Liu focused her attention to business process management, including process model verification and model-driven solution engineering management. Currently, her research interests include formal methods, business process management, and service building technologies.

Samuel Müller, IBM Research Division, IBM Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland (sml@zurich.ibm.com). Samuel Müller obtained a M.S. in Computer Science from the University of Zurich in 2004 and a M.A. in Economics from the University of Zurich in 2006. 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 towards his doctorate as an external Ph.D. student at the Swiss Federal Institute of Technology (ETH) Zurich, where he is a member of the Information Security group. His thesis advisors are Prof. Dr. David Basin and Prof. Dr. Birgit Pfitzmann and his research interests include modal logics, formal methods and modeling, risk & compliance management, game theory and economics.

Ke Xu, Automation Department, Tsinghua University, Beijing, 100084, Peoples' Republic of China (xk02@mails.tsinghua.edu.cn). Ke Xu received his B.Sc. Degree (July 2002) from the Automation Department of Shanghai JiaoTong University, P.R. China. He is currently a Ph.D. candidate at the National CIMS Research and Engineering Center in Tsinghua University, P.R. China. His main research interests include process algebra, model checking, and their applications in grid computing and business integration. He serves as a member of academic committee in the Automation Department of Tsinghua University. He is also an IBM Ph.D. Fellow in year 2006-2007.