2012 IEEE 16th International Enterprise Distributed Object Computing Conference Workshops

# Optimization of Business Processes in Service Oriented Architecture

Andrzej Ratkowski, Krzysztof Sacha and Andrzej Zalewski
Warsaw University of Technology
Warszawa, Poland
{a.ratkowski, k.sacha, a.zalewski}@ia.pw.edu.pl

***Abstract*—This paper describes a method for the implementation and optimization of business processes in a service oriented architecture (SOA). A process specification is created by business people, and expressed in Business Process Modeling Notation (BPMN). The specification is then translated into Business Process Execution Language (BPEL), and used by technical people as a reference process, which is subject to a series of transformations that change the internal process structure in order to improve the quality of the process execution. The behavior of each transformed process is verified automatically against the behavior of the reference process. The verification mechanism is based on a mapping from BPEL to Language of Temporal Ordering Specification (LOTOS), followed by a comparison of the trace set that is generated using a program dependence graph of the reference process and the trace set of the transformed one. When the design goals have been reached, the transformed BPEL process can be executed on a target SOA environment using a BPEL engine.**

***Keywords*-Business Process, BPEL, Service Oriented Architecture, Program Dependence Graph, LOTOS**

## I. Introduction And Related Work

A business process is a set of partially ordered activities, which produce a specific product or service that adds value for a customer. The specification of a business process can be expressed graphically using an appropriate notation, such as the Business Process Modeling Notation [1] or UML activity diagrams [2]. The structure of the specified business process and the arrangement of activities reflect business decisions made by business people. The implementation of a business process on a computer system is expected to exhibit the defined behavior at a satisfactory level of quality. Reaching such a level of quality may require restructuring of the initial process, according to a series of technical decisions, which have to be made by technical people.

This paper describes a transformational method for the implementation and optimization of business processes in Service Oriented Architecture [3-5]. The starting point of the method is a process specification, defined by business people using BPMN. The specification is translated by an open source tool into a reference process expressed in Business Process Execution Language [6-8]. The reference process is subject to a series of transformations, which change the internal structure of the process and make it closer to the implementation. The transformations are selected manually by human designers (technical people) and performed automatically by a software tool. Each transformation changes the ordering of activities within the process in order to improve the quality of the process implementation, e.g. by benefiting from the parallel structure of services, but preserves the reference process behavior. When the design goals have been reached, the transformed BPEL process can be executed on a target SOA environment. The evaluation of the process quality can be guided by metrics, which are similar to metrics discussed in the literature [9-11].

A critical part of the method is the verification of behavior that the process exhibits before and after a transformation. One possible verification mechanism could be based on trace semantics of processes. To use this principle, we could define a trace of the reference process execution, and request that the trace of a process after a transformation was the same. Unfortunately, the identity of traces is not an appropriate criterion, because it eliminates such transformations, which reorder the process activities without changing the results of the process execution.

Therefore, we have chosen another approach, which is based on an original mapping from BPEL to Language of Temporal Ordering Specification [12,13]. In the first step, dataflow dependencies between the activities of the reference process are analyzed using program slicing techniques [14,15] and presented in the form of a program dependence graph [16,17]. All the insignificant ordering constraints are removed and the minimal dependence process is created and mapped into a LOTOS expression. The labeled transition system of this expression defines the set of all the traces that define the behavior of the reference process, which is considered acceptable from the application viewpoint.

In the second step, the reference process is subject to transformations selected by a designer, and the transformed process is mapped into a simple LOTOS expression. If all the traces generated by the labeled transition system of this expression are within the set of traces generated by the labeled transition system of the minimal dependence process, then the transformed process behavior is compliant with the reference process behavior.

The rest of this paper is organized as follows. The structure of a business process, BPMN notation and BPEL language are described in Section II. LOTOS language and a BPEL to LOTOS mapping are covered in Section III. Transformations of a BPEL process are shown in Section IV. The verification method is described in Section V. Quality metrics are exemplified in Section VI. Conclusions and plans for further research are given in Section VII.

978-0-7695-4786-2/12 $26.00 © 2012 IEEE
DOI 10.1109/EDOCW.2012.16

## II. THE STRUCTURE OF A BUSINESS PROCESS

A business process is a set of related activities, which produce a service or product for a customer. Part of these activities can be implemented on-site by locally executed functions, while others can be implemented externally, by services offered by a service-oriented environment. The services can be viewed from the process perspective as the main business data processing functions.

The specification of a business process can be defined textually, or with a flowchart, as a sequential or parallel set of activities with interleaving decision points. Business Process Modeling Notation is a graphical representation used frequently by business people for specifying business processes in their organizations. An example BPMN process is shown in Fig. 1. The process starts after receiving an invocation from a remote client (another process). Then, it invokes two services in parallel and when the invocations are finished, i.e. the results are received; it performs the first piece of a local computation. Next, it checks a condition and decides to perform one out of two other pieces of computation. Finally, the process replies to the client. This way, a business process in SOA environment can be implemented as a service, which is composed of services and which can be invoked by another service.

BPMN specification of a business process is created by a business analyst, who defines functioning of an organization. The specification can be automatically translated into a BPEL program, which can be used by technicians for semi-automatic implementation. The translation from BPMN to BPEL is subject to a research effort [7,8], which is not covered in this paper. An open-source tool can be found at http://code.google.com/p/bpmn2bpel/.

BPEL syntax is composed of a set of instructions, called activities, which are XML elements indicated in the document by explicit markup. The activities can be classified as simple, which define elementary pieces of computation, and structured, which comprise another activities. The set of BPEL activities is rich, however, we focus in this paper on a limited subset of activities for defining flow of control, service invocation and basic data handling.

The body of a BPEL process consists of simple activities, which are elementary pieces of computation, and structured elements, which comprise other simple or structured activities, nested in each other to an arbitrary depth. Simple activities are <assign>, which implements substitution, <invoke>, which invokes an external service, and <receive>, <reply> pair, which receives and replies to an invocation.

```
<sequence>
   <receive name="rcv" … > </receive>
   <flow>
      <invoke name="inv1" … > </invoke>
      <invoke name="inv2" … > </invoke>
   </flow>
   <assign name="ass1" > … </assign>
   <if name="alt">
      <condition> … </condition>
         <assign name="ass2" > … </assign>
      <else>
         <assign name="ass3" > … </assign>
   </if>
   <reply name="rpl" … > </reply>
</sequence>
```

Figure 2. BPEL program of the business process in Fig. 1

Structured activities are <sequence> element to describe sequential execution, <flow> element to describe parallel execution and <if> alternative branching. An example BPEL program, which implements the business process in Fig. 1, is shown in Fig. 2. We use roman letters for the language tags and attributes (terminal symbols), and italics for nonterminal symbols. Name attribute will be used to refer to particular activities of the program in the subsequent figures.

The first executable activity of the program is <receive>, which waits for a message that invokes the process execution and conveys a value of the input argument. The last activity of the process is <reply>, which responds to the invocation by sending a message that returns the result. The activities between <receive> and <reply> execute a business process, which invokes other services and transforms the input into the output. This is a typical construction of a BPEL process, which itself can be viewed as a service invoked and used by other services.

SOA services are assumed stateless [3]. This means that the behavior of a service, observed by another services, depends only on values passed to the service by means of messages. An impact of services on the real world will be discussed in Section V. On the other hand, the behavior of a service depends on the type, number and order of activities, which constitute the program body. The activities and the order of their execution can formally be described by a LOTOS expression, which captures a trace, or a set of traces, of the program execution. Services composed of the same set of activities and characterized by the same set of execution traces, return the same results for given arguments, regardless of their internal structure.

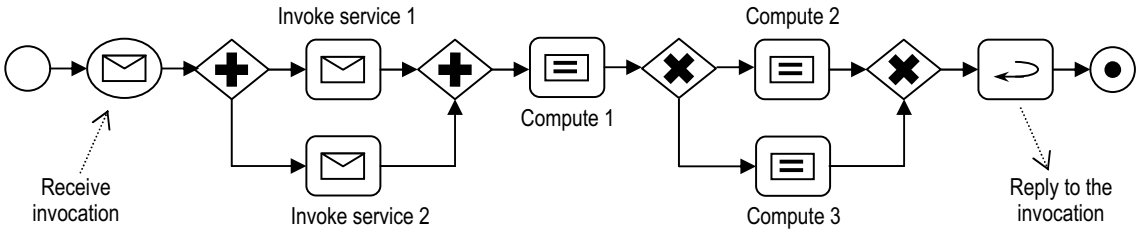


Figure 1. BPMN specification of a business process

## III. THE LANGUAGE LOTOS

Language of Temporal Ordering Specification (LOTOS) is one of the formal description techniques developed within ISO [12] for the specification of open distributed systems. The semantics of LOTOS is based on algebraic concepts and is defined by a labeled transition system (LTS), which can be built for each LOTOS expression.

A process, or a set of processes, is modeled in LOTOS as a behavior expression, composed of actions, operators and parenthesis. Actions correspond to activities, which constitute the process body. Operators describe the ordering of actions during the process execution. The list of operators, together with an informal explanation of their meaning is given in Table I. We use $\mu$ to denote an arbitrary action and $\delta$ to denote a special action of a successful termination of an expression or sub-expression.

LOTOS expression can be executed, generating a sequence of actions, which is called the execution trace. An expression which contains parallel elements can generate many traces, each of which describes an acceptable ordering of actions. Not all of the actions that are syntactic elements of an expression are directly visible within the execution trace. These actions are called observable actions and are denoted by alphanumeric identifiers, e.g. *g*1, *g*2 etc. Only observable actions are counted as members of an execution trace of the expression. Other actions cannot be identified when observing the trace. These actions are called unobservable actions. Unobservable actions are denoted by letter $i$ and are not counted as members of an execution trace.

Formally, unobservable actions are those that are listed within the **hide** clause of LOTOS. In this paper, we omit this clause to help keeping the expressions simple.

The operational semantics of LOTOS provides a means to derive the actions that an expression may perform from the structure of the expression itself. Formally, the semantics of an expression $B$ is a labeled transition system $< S, A, \rightarrow, I >$ where:

$S$ – is a set of states (LOTOS expressions),
$A$ – is a set of actions,
$\rightarrow$ – is a transition relation, $\rightarrow \subseteq S \times A \times S$,
$B$ – is the initial state (the given expression).

The transition relation is usually written as $B \xrightarrow{\mu} B'$. For example, the semantics of expression $(g; B1)$ can be described by a labeled transition:

$$g; B1 \xrightarrow{g} B1$$

This means that expression $(g; B1)$ is capable of performing action *g* and transforming into expression *B*1.

The semantics of a complex expression consists of a directed graph (a tree) of labeled transitions, which root is the expression itself, and which edges are the labeled transitions. Each path from the root node to a leaf node of the graph defines a sequence of actions, which is an execution trace of the expression.

LOTOS expression can serve as a tool for modeling the set of traces of execution of a BPEL process. To use the tool, we can model BPEL activities as observable actions in LOTOS, and describe the ordering of activities during the process execution by means of a LOTOS expression.

Simple activities of BPEL are mapped to observable actions of LOTOS, followed by **exit** symbol. For example:

<**assign** name="*ass*"> is mapped to *ass*; **exit**
<**invoke** name="*inv*"> is mapped to *inv*; **exit**

Structured activities of BPEL are translated into LOTOS expressions according to the following rules:

- <sequence> element is mapped into sequential composition (>>),
- <flow> element is mapped to parallel execution (|||),
- <if> element is mapped to alternative choice ([ ]).

Consider, for example, BPEL process in Fig. 2. If we map the process activities according to the above rules, then the resulting LOTOS expression looks as follows:

*rcv*;**exit** >> ( *inv*1;**exit** ||| *inv*2;**exit** ) >> *ass*1;**exit** >>
( *ass*2;**exit** [ ] *ass*3;**exit** ) >> *rpl*;**exit**

The trace set generated by the labeled transition system of this expression consists of four traces composed of the following observable actions:

*rcv*; *inv*1; *inv*2; *ass*1; *ass*2; *rpl*
*rcv*; *inv*1; *inv*2; *ass*1; *ass*3; *rpl*
*rcv*; *inv*2; *inv*1; *ass*1; *ass*2; *rpl*
*rcv*; *inv*2; *inv*1; *ass*1; *ass*3; *rpl*

The semantics of parallelism in LOTOS is interleaved. Parallel execution of the two <invoke> activities that are nested within <flow> element of the BPEL process is modeled by the possibility of executing the corresponding LOTOS actions *inv*1 and *inv*2 in an arbitrary order. The semantics of choice is exclusive. When one branch of <if> element begins execution, then the other branch disappears. Special action $\delta$ generated by **exit** is not counted in the execution traces because it is an unobservable action.

TABLE I. EXPRESSIONS IN BASIC LOTOS

| Syntax | Explanation |
|---|---|
| **stop** | inaction, lack of action |
| $\mu$; *B* | action $\mu$ precedes execution of expression *B* |
| *B*1 [ ] *B*2 | alternative choice of expressions *B*1 and *B*2 |
| *B*1 \|[ *g*1,…*gn* ]\| *B*2 | parallel execution of *B*1 and *B*2 synchronized at actions *g*1,…,*gn* |
| *B*1 \|\|\| *B*2 | parallel execution with no synchronization between *B*1 and *B*2 |
| **exit** | successful termination; generates a special action $\delta$ |
| *B*1 >> *B*2 | sequential composition: successful execution of *B*1 enables *B*2 |
| *B*1 [> *B*2 | disabling: successful execution of *B*1 disables execution of *B*2 |
| **hide** *g*1,…,*gn* **in** *B* | hiding: actions *g*1,…,*gn* are transformed into unobservable ones |

## IV. TRANSFORMATIONS

The body of a BPEL process consists of activities. Simple activities invoke external services and perform local data processing operations. Structured activities comprise other activities and decide on the order, in which the activities are executed. The results as well as the quality, e.g. efficiency of execution, of the process depend on the set of activities, which constitute the process body, and on the process structure. The goal of a transformation is to change the process structure in a way, which improves the process quality without changing the results of the process execution.

A transformation applies to an element of a BPEL process. It can consist in moving an activity from one place to another in a BPEL program or in replacing one structured activity, e.g. <sequence>, by another, e.g. <flow>, composed of the same elements. If the results of the process execution are not changed by the transformation, then this transformation is considered safe for the user. It can easily be seen that a sequence of safe transformations is also safe.

Several transformations can be defined. The basic ones are the following: Displacement, parallelization of the process operations, serialization, aggregation of processes into a single entity and a split of a single process. The first three of these transformations are described in detail below.

**Transformation 1: Displacement.**

Consider an arbitrary BPEL process, composed of activities nested in each other in compliance with the rules of BPEL syntax. Transformation 1 moves a selected activity, either simple or structured, from its original location in the program, into another place in the program structure.

**Transformation 2: Parallelization.**

Consider <sequence> element of a BPEL program, which contains $n$ arbitrary activities $A_1$ through $A_n$. Transformation 2 parallelizes the execution of activities by replacing <sequence> element with <flow> element, composed of the same activities.

**Transformation 3: Serialization.**

Consider <flow> element of a BPEL program, which contains $n$ arbitrary activities $A_1$ through $A_n$. Transformation 3 serializes the execution of activities by replacing <flow> element with <sequence> element, composed of the same activities.

Transformations 1 through 3 can be composed in any order, resulting in a complex transformation of a process structure. Transformations 4 and 5 play an auxiliary role and facilitate such a superposition. These transformations do not change the process behavior, because they do not change the order of execution of commands.

**Transformation 4: Sequential structuring.**

Transformation 4 adds a pair of tags <sequence> </sequence> around a single activity $A$.

**Transformation 5: Parallel structuring.**

Transformation 5 adds a pair of tags <flow> </flow> around a single activity $A$.

To illustrate optimization of a BPEL process and the use of transformations, assume that *ass*1 activity in Fig. 2 uses only a data value, which has been set by *inv*1 activity, and that the structured <if> activity uses data values set by *inv*2 only. It is easy to see that according to these assumptions, the activities *ass1* and <if> are independent. Our goal is to speed-up execution of the process in Fig. 2.

One way to reach the goal is by parallelizing the execution of *ass*1 and <if> activities. To do this, we can apply transformation 5 to add a pair of <flow> </flow> tags around *ass*1, and then apply transformation 1 to move <if> activity into the scope of the previously added <flow> tag. The structure of the transformed process is shown in Fig. 3a. The proof of safeness of this transformation sequence is given in Section V.

Alternatively, we can restructure the process in such a way that two branches are executed in parallel: One composed of *inv*1 and *ass*1 activities, and the other one

(a)

```
<sequence>
  <receive name="rcv">
  <flow>
    <invoke name="inv1">
    <invoke name="inv2">
  </flow>
  <flow>
    <assign name="ass1">
    <if name="alt">
      <condition>
        <assign name="ass2">
      <else>
        <assign name="ass3">
    </if>
  </flow>
  <reply name="rpl">
</sequence>
```

(b)

```
<sequence>
  <receive name ="rcv">
  <flow>
    <sequence>
       <invoke name="inv1">
       <assign name="ass1">
     </sequence>
     <sequence>
       <invoke name="inv2">
       <if name="alt">
         <condition>
           <assign name'"ass2">
         <else>
           <assign name="ass3">
        </if>
      </sequence>
   </flow>
   <reply name="rpl">
</sequence>
```

Figure 3. The process after transformations 5 (a), and after transformations 4 and 1 (b)

composed of *inv*2 and <if>. To do this, we can apply transformation 4 twice, in order to add two pairs of <sequence> </sequence> tags: Around *inv*1 and around *inv*2. Then, we can use transformation 1 twice, to move *ass*1 activity into the scope of the first added <sequence>, and to move <if> activity into the scope of the second added <sequence>. The structure of the transformed process is shown in Fig 3b. The proof of safeness of this transformation sequence is given in Section V.

## V. VERIFICATION

The reference process defines a behavior, which is acceptable from the application viewpoint. In the transformation phase, the structure of the process and the flow of execution are changed, in order to improve the performance characteristics. However, the externally observable behavior of the process must remain unchanged. The problem is how to define this behavior and how to verify that it has not been changed.

The observable behavior of a process in a SOA environment consists of values of the variables that are visible to the outside world, i.e. variables, which are passed as arguments when external services are being invoked, and variables, which values are returned at the end of the process execution. This is sufficient, because services are stateless [3] and return the same results if invoked with the same values of arguments. An impact of services on the real world is taken into account, as described in the next subsection.

The verification follows a two-phase approach, illustrated in Fig. 4 where B2L acronym stands for: BPEL-to-LOTOS mapping. In the first phase, data dependencies between the activities of the reference process are analyzed using the Program Dependence Graph (PDG) and all the unnecessary sequencing constraints on these activities are removed. The resulting graph reflects all the dataflow dependencies between the activities of the reference process and is free from the initial process structuring. If we preserve the dataflow dependencies during the process transformation, then the values computed by all the activities remain unchanged. In particular, the values that are passed between the processes by means of the inter-process communication activities: <invoke> in one process and <receive> <reply> pair in the other one, remain also unchanged. Program dependence graph is then transformed into a LOTOS

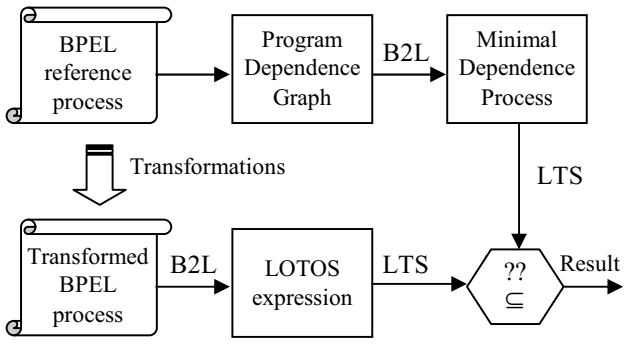


Figure 4. Verification of a process behavior

expression, which is called a Minimal Dependence Process (MDP). The labeled transition system of the minimal dependence process defines a set of traces that define the behavior of all processes, which comply with dataflow dependencies defined within the reference process. The first phase is performed only once for a given reference process.

The second phase is performed repetitively during the transformational implementation cycle. A transformed BPEL process is mapped into a LOTOS expression, as described in Section III. The set of traces generated by the labeled transition system of this expression is compared with the set of traces generated by the labeled transition system of the minimal dependence process. If the trace set generated by the expression is within the trace set of MDP, then the behavior of the transformed BPEL process is safe in that it preserves the behavior of the reference process.

### A. Program Dependence Graph

To capture the behavior of a process, we use a technique called program slicing [14,15], which allows finding all such activities in a program, which influence the value of a selected variable in a specific point of the program. For example, the value of a variable used as an argument by a service invocation activity or by the final <reply> activity of the process. The conceptual tool for the analysis is program dependence graph [16,17], which nodes are elements of a BPEL program and edges are control and dataflow dependencies between these elements.

The algorithm for constructing PDG of a BPEL program consists of the following steps (Fig. 5):

1. Define nodes of the graph, which are activities at all layers of nesting.
2. Define control edges (solid lines in Fig. 5b), which follow the nested structure of the program, e.g. an edge from <sequence> to <flow> means that <flow> activity is nested within the <sequence> element.
3. Define dataflow edges (dashed lines in Fig 5b), which show data dependencies between activities revealed using program slicing techniques. For example, an edge from *rcv* to *inv*1 means that the output value of *rcv* activity is used as the input argument of *inv*1.

A dataflow edge between two nodes in a program dependence graph implies that the result of the activity at the end of the edge depends on the result of the activity at the beginning of this edge. Therefore, the arrangement of activities during the program execution, reflected by the succession of activities in an execution trace, must comply with the direction of dataflow edges. Any change to this arrangement may lead to a change in the program behavior.

Structured nodes <sequence> and <flow>, as well as control edges connected to these nodes reflect the structure and the flow of control within the reference process. Both of the two can be changed during the process transformation. Therefore, structured nodes <sequence> and <flow> are removed from the program dependence graph. Control edges, which output a removed node, are redirected to the direct predecessor node, if one exists, or are removed as well. On the other hand, structured node <if> and the control

(a)

```
<sequence>
  <receive name="rcv">
  <flow>
    <invoke name="inv1">
    <invoke name="inv2">
  </flow>
  <assign name="ass1">
  <if name="alt">
    <condition>
      <assign name="ass1">
    <else>
      <assign name="ass2">
  </if>
  <reply name="rpl">
</sequence>
```

(b)

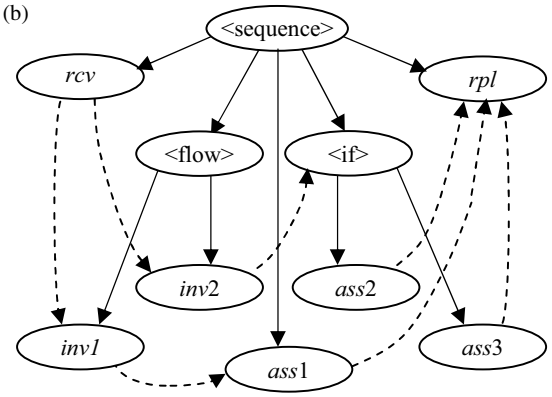


Figure 5. A nested BPEL process: Process body (a) and program dependence graph (b)

edges, which output this node, are not removed from the graph, because they reflect the logic of conditional branching. The reduced program dependence graph of the reference process in Fig. 5 is shown in Fig. 6.

The services invoked by a process can have an impact on the real world. If this is the case, a specific ordering of these services can be required. A designer can reflect this requirement adding appropriate edges to the reduced program dependence graph.

Let $G_P = (N_P, E_P)$ be a reduced program dependence graph of a BPEL process $P$. It can be proved from the above algorithm that graph $G_P$ is acyclic. We say that node $n_i$ precedes node $n_j$, denoted $n_i < n_j$, if there exists a path from $n_i$ to $n_j$ in the program dependence graph. Precedence relation is a strict partial order in $N_P$.

### *B. Minimal Dependence Process*

An execution of a BPEL program can be modeled as a process of traversing through the program dependence graph, starting at the initial node and moving along the directed arcs. The process stops when the last node of the graph is reached. Because the ordering of nodes is only partial, then the succession of visited nodes and arcs may vary. For example, the first node in Fig. 6 is *rcv*. After visiting this node, data can be passed along the arc to *inv*1 or along the arc to *inv*2. If the former is true, then in the next step node *inv*1 can be visited or data can be passed along the arc to *inv*2. However, node *inv*2 could not be visited in that step.

Nodes and arcs of a program dependence graph can be mapped to LOTOS actions in such a way that a visit to a node is mapped to an observable action, while moving along an arc is mapped to an unobservable action. A sequence of execution steps is mapped to a sequence of LOTOS actions. An example mapping of nodes and arcs is shown in Fig. 7.

A visit to a node enables visiting all the succeeding nodes. However, the way of reaching this node (described by an expression $B1$) has no influence on the other part of execution after visiting the node (described by an expression $B2$), and vice versa. This means that actions performed before the visit (within $B1$) and actions performed after the visit (within $B2$) are independent. However, finishing the visit and passing data along the output arcs of the visited node make a synchronization point between the two. This informal description can be expressed formally in LOTOS using the operator of parallel execution of $B1$ and $B2$ synchronized at action assigned to the output arc.

Minimal dependence process is a LOTOS expression that defines the set of traces, which are compliant with dataflow dependencies described by the program dependence graph. This way, minimal dependence process defines the semantics of a BPEL reference process. The algorithm for building MDP searches through the reduced program dependence graph, starting at the initial node. LOTOS expression is constructed iteratively, by appending a new sub-expression to the existing part of MDP in each visited node.

For example, the first action in the graph in Fig 7 is *rcv*, followed by one of the actions *a* or *b*. Hence, the appropriate LOTOS expression begins with:

*rcv*; (*a*|||*b*) …

Passing data along one of the output arcs enables traversing through the other parts of the graph. Action *a* enables *inv2*, while action *b* enables *inv1*; *ass1*. Both of the enabled groups of actions are independent and can be executed in parallel. Hence, the next part of the LOTOS expression is:

( (*rcv*; (*a*||*b*) ) |[*a*]| *a*; *inv2*;…) |[*b*]| *b*; *inv1*;*ass1*;…

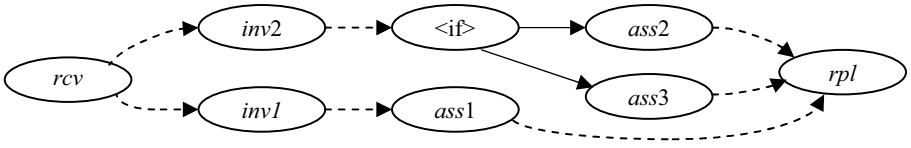


Figure 6. The reduced program dependence graph of the process in Fig. 5

Formally, the algorithm for constructing MDP of a BPEL program described by a reduced program dependence graph consists of the following steps:

1. Assign an observable LOTOS action to each node of the reduced program dependence graph, except of <if> nodes. The action is identified by the name attribute of the node (nodes in PDG are BPEL activities).
2. Find paths in the reduced program dependence graph, such that the first node of a path has one output edge, the last node has one input edge and each other node has one input and one output edge. Substitute each path with a single node, and assign to this node LOTOS expression composed of actions, which were assigned to the removed nodes, separated by semicolons.
3. Assign an unobservable LOTOS action to each edge of the graph. The actions should be distinct, except of the edges, which output the alternative nodes of an <if> activity and input the same node. These actions should be equal.
4. Initiate graph search from the initial node. Create LOTOS expression, denoted MDP', composed of:
   - the expression assigned to the initial node,
   - semicolon and parallel composition of actions assigned to the output edges.
5. Search through the nodes of the reduced program dependence graph in a sequence complying with the precedence relation ($n_i$ is visited before $n_j$, if $n_i < n_j$). For each node, place parentheses around the MDP' and append the following expressions:
   - parallel composition synchronized on actions assigned to the input edges,
   - a sequence of actions assigned to the input edges, separated by semicolons,
   - semicolon and LOTOS expression assigned to the node (empty for <if> node),
   - semicolon, and parallel composition of actions assigned to the output dataflow edges or an alternative selection of actions assigned to the output control edges (the case of <if> activity).
6. When the algorithm stops, after visiting the last node, MDP' becomes the minimal dependence process MDP.

For example, consider the reduced program dependence graph in Fig.6. The steps of assigning LOTOS expressions to nodes (step 1), removing paths (step 2) and assigning unobservable actions to edges (step 3) change the graph as shown in Fig. 7.

The minimal dependence process, derived from the graph in Fig. 7, takes the form of the following LOTOS expression:

$$(((((rcv;(a|||b))\ |[a]|\ a;inv2;(y[\,]n))\ |[b]|\ b;inv1;ass1;c)\ |[y]|\ y;ass2;d)\ |[n]|\ n;ass3;d)\ |[c,d]|\ c;d;rpl$$

The labeled transition system of this expression generates a set of 12 traces, each of which is a sequence of observable actions:

{ *rcv*; *inv*1; *ass*1; *inv*2; *ass*2; *rpl* ,
*rcv*; *inv*1; *ass*1; *inv*2; *ass*3; *rpl* ,
*rcv*; *inv*1; *inv*2; *ass*1; *ass*2; *rpl* ,

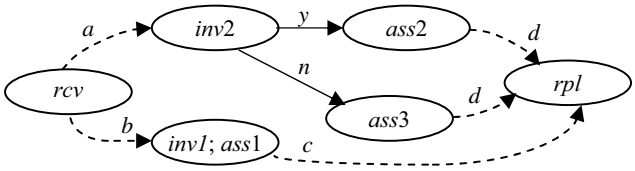


Figure 7. Construction of MDP: The reduced program dependence graph (Fig. 6) after step 3

*rcv*; *inv*1; *inv*2; *ass*1; *ass*3; *rpl* ,
*rcv*; *inv*1; *inv*2; *ass*2; *ass*1; *rpl* ,
*rcv*; *inv*1; *inv*2; *ass*3; *ass*1; *rpl* ,
*rcv*; *inv*2; *ass*2; *inv*1; *ass*1; *rpl* ,
*rcv*; *inv*2; *ass*3; *inv*1; *ass*1; *rpl* ,
*rcv*; *inv*2; *inv*1; *ass*2; *ass*1; *rpl* ,
*rcv*; *inv*2; *inv*1; *ass*3; *ass*1; *rpl* ,
*rcv*; *inv*2; *inv*1; *ass*1; *ass*2; *rpl* ,
*rcv*; *inv*2; *inv*1; *ass*1; *ass*3; *rpl* }

The trace set of the reference process in Fig. 2 consists of 4 traces:

{ rcv; inv1; inv2; ass1; ass2; rpl ,
rcv; inv1; inv2; ass1; ass3; rpl ,
rcv; inv2; inv1; ass1; ass2; rpl ,
rcv; inv2; inv1; ass1; ass3; rpl }

The trace set of the first transformed process in Fig. 3a consists of 8 traces:

{ rcv; inv1; inv2; ass1; ass2; rpl ,
rcv; inv1; inv2; ass1; ass3; rpl ,
rcv; inv1; inv2; ass2; ass1; rpl ,
rcv; inv1; inv2; ass3; ass1; rpl ,
rcv; inv2; inv1; ass1; ass2; rpl ,
rcv; inv2; inv1; ass1; ass3; rpl ,
rcv; inv2; inv1; ass2; ass1; rpl ,
rcv; inv2; inv1; ass3; ass1; rpl }

The trace set of the second transformed process in Fig. 3b consists of 12 traces, and is identical to the trace set of MDP. Obviously, the trace set of MDP includes the trace sets of the reference process as well as of the transformed processes. This proves that both transformations are safe.

## VI. Quality Metrics

Quality metrics to measure parallel programs have been studied for many years. A traditional tool for measuring performance of a parallel application is Program Activity Graph, which describes parallel flow of control within the application [9]. We do not use this graph; nevertheless, our Length of thread metric can be viewed as an approximation of Critical path metric described in [9]. Similarly, our Number of threads metric is similar to Available concurrency defined in [10]. The value of each metric can be calculated using a program dependence graph.

**Size of a process** is measured as the number of simple activities in the BPEL program. More precisely, the value of this metric equals the number of leaf nodes in the program

dependence graph of this process. Leaf nodes are simple activities, which perform the processing of data. Therefore, the value of the process size metric could be considered a measure of the amount of work, which can be provided by the process. For example, the size of all the processes shown in Fig. 2, 3a and 3b is 7.

**Complexity of the process** is measured as the total number of nodes in PDG. For example, the size of program in Fig. 2 is 10, the size of program in Fig. 3a is 11, and the size of program in Fig. 3b is 12. The number of nodes in PDG, compared to the size of the process, describes the amount of excess in the graph, which can be considered a measure of complexity of the program structure.

**Number of threads** is measured as the number of items within <*flow*> elements of a BPEL program, at all levels of nesting. The algorithm of computation assigns weights to nodes of the program dependence graph of the process, starting from the leaves up to the root, according to the following rules:

- the weight of a simple BPEL activity is 1,
- the weight of a <*flow*> element is the sum of weights assigned to the descending nodes (i.e. nodes directly nested within the <*flow*> element),
- the weight of a <*sequence*> or <if> element is the maximum of weights assigned to the descending nodes (i.e. nodes directly nested within the <*sequence*> element).

The metric value equals the weight assigned to the top <*sequence*> node of PDG. For example, the number of threads of all the processes in Fig. 2, 3a and 3b is 2.

**Length of thread** is measured as the number of sequentially executed activities within a BPEL program. The algorithm of computation assigns weights to nodes of the program dependence graph of the process, starting from the leaves up to the root, according to the following rules:

- the weight of a simple BPEL activity is 1,
- the weight of a <*flow*> or <if> element is the maximum of weights assigned to the descending nodes (i.e. nodes directly nested within the <*flow*> element),
- the weight of a <*sequence*> element is the sum of weights assigned to the descending nodes (i.e. nodes directly nested within the <*sequence*> element).

The metric value equals the weight assigned to the top <*sequence*> node of PDG. For example, the length of thread of programs in Fig. 2 and Fig. 3a is 1, while the length of thread of program in Fig. 3b is 2.

To summarize the examples, one can note that all the three processes has the same size, the complexity of the transformed processes is higher than the complexity of the reference process, the number of threads of all the processes is the same, but the last process has the threads longer. This means that the last process is probably the optimum one. Unfortunately, the complexity of this process is also the highest.

## VII. CONCLUSIONS

The behavior of a business process is a business decision. The way of implementation of a business process is a technical decision. The transformational method for implementation and optimization of business processes in SOA, described in this paper, promotes separation of concerns and allows making business decisions by business people and technical decisions by technical people. The former relates to the specification phase of a reference process, which reflects the flow of business logic used in an organization. The latter relates to the implementation phase, in which the reference process is transformed in order to improve efficiency and benefit from the parallel structure of services in a SOA environment. The quality of processes can be measured by metrics exemplified in Section VI.

Transformations described in Section IV can change the process structure and, in some circumstances, can preserve the observable behavior of the reference process. To guarantee that the process behavior has not been changed during the transformations, we developed a verification technique, which relies on relaxing the ordering of activities by means of a program dependence graph and modeling the resulting minimal dependence process by means of a LOTOS expression. A comparison of trace sets generated by the relaxed reference process and the transformed process allows deciding whether the process behavior has changed. If the answer is 'no' (no changes), then the transformed process can be accepted. Otherwise, the comparison of traces can show the process fragment, in which the change occurred. If this is the case, the decision whether to allow or to deny such a change can be made by a human designer.

## ACKNOWLEDGMENT

This research was supported in part by the Ministry of Science and Higher Education under the grant number 5321/B/T02/2010/39.

## REFERENCES

[1] OMG, Business Process Modeling Notation (BPMN), http://www.omg.org/spec/BPMN/1.2

[2] OMG, Unified Modeling Language (OMG UML): Superstructure, version V2.1.2, http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF (2007)

[3] Erl T.: Service-oriented Architecture: Concepts, Technology, and Design. Prentice Hall, Englewood Cliffs (2005)

[4] MacKenzie C. M., Laskey K., McCabe F., Brown P. F, Metz R.: Reference model for service oriented architecture 1.0. Technical report, OASIS (2006)

[5] Ratkowski A., Zalewski A.: Transformational Design of Business Processes for SOA. In: Huzar Z., Koci R., Meyer B., Walter B., Zendulka J. (Eds.): CEE-SET 2008. LNCS vol. 4980, pp. 76-90. Springer, Heidelberg (2011)

[6] Jordan D., Evdemon J.: Web Services Business Process Execution Language Version 2.0. OASIS Standard (2007)

[7] White S.: Using BPMN to Model a BPEL Process, BPTrends 3, www.bptrends.com (2005)

[8] Ouyang, C., van der Aalst,W.M.P., Dumas, M., ter Hofstede, A.H.M.: Translating BPMN to BPEL, BPM Center Report BPM-06-02, BPMcenter.org (2006)

[9] Hollingsworth J. K., Miller B. P.: Parallel program performance metrics: A comparison and validation, Proc. ACM/IEEE Conference on Supercomputing, pp. 4 - 13, IEEE Computer Society Press (1992)

[10] Van Amesfoort A.S., Varbanescu A.L., Sips H.J.: Proc. 15th Workshop on Compilers for Parallel Computing, pp 1-13 (2010)

[11] Ratkowski A, Sacha K.: Business Process Design In Service Oriented Architecture. In: A. Grzech, L. Borzemski, J. Świątek, Z. Wilimowska (Eds.): Information Systems Architecture and Technology, pp. 15-24. Wroclaw University of Technology, Wroclaw (2011)

[12] ISO 8807: Information Processing Systems: Open Systems Interconnection: LOTOS: A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Organization for Standards (1989)

[13] Salaun G., Ferrara A., Chirichiello A.: Negotiation among web services using LOTOS/CADP. In: L. Zhang (Ed.): Proc. European Conference on Web Services. LNCS vol. 3250, pp. 198-212. Springer, Heidelberg (2004)

[14] Weiser M.: Program slicing. IEEE Trans. Software Eng., 10 (4), pp. 352-357 (1984)

[15] Binkley D., Gallagher K.B.: Program slicing, Advances in Computers, 43, pp. 1-50 (1996)

[16] Ottenstein K.J., Ottenstein L.M.: The program dependence graph in a software development environment. Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM, pp. 177-184 (1984)

[17] Mao C.: Slicing web service-based software. IEEE International Conference on Service-Oriented Computing and Applications, IEEE, pp. 1-8 (2009)