Transformational Implementation of Business Processes in SOA

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

Abstract—**The paper develops a method for transformational implementation and optimization of business processes in a service oriented architecture. The method promotes separation of concerns and allows making business decisions by business people and technical decisions by technical people. To achieve this goal, a description of a business process designed by business people is automatically translated into a program in Business Process Execution Language, which is then subject to a series of transformations developed by technical people. Each transformation changes the process structure in order to improve the quality characteristics. Two approaches to the verification of the process correctness are discussed. The first one applies a correct-by-construction approach to transformations. The other one relies on automatic verification of the transformed process behavior against the behavior of the original reference process. The verification mechanism is based on a mapping from Business Process Execution Language to Language of Temporal Ordering Specification, 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; service oriented architecture; BPEL; LOTOS; transformational implementation.

I. INTRODUCTION

This paper is an extension of the ICSEA paper [1] on transformational implementation of business processes in a service oriented architecture. A business process is a set of logically related activities performed to achieve a defined business outcome [2]. The structure of a business process and the ordering of activities reflect business decisions made by business people and, when defined, can be visualized using an appropriate notation, e.g., Business Process Model and Notation [3] or the notation of ARIS [4]. The implementation of a business process on a computer system is expected to exhibit the defined behavior at a satisfactory level of quality. Reaching the required level of quality may need decisions, made by technical people and aimed at restructuring of the initial process in order to benefit from the characteristics offered by an execution environment. The structure of the implementation can be described using another notation, e.g., Business Process Execution Language [5] or UML activity diagrams [6].

This paper describes a transformational method for the implementation and optimization of business processes in a service oriented architecture (SOA). The method begins with an initial definition of a business process, written by business people using Business Process Modeling Notation (BPMN). The business process is automatically translated into a program in Business Process Executable Language (BPEL), called a reference process. The program is subject to a series of transformations, each of which preserves the behavior of the reference process, but changes the order of activities, as means to improve the quality of the process implementation, e.g., by benefiting from the parallel structure of services. Transformations applied to the reference process are selected manually by human designers (technical people) and performed automatically, by a software tool. When the design goals have been reached, the iteration stops and the result is a transformed BPEL process, which can be executed on a target SOA environment.

Such an approach promotes separation of concerns and allows making business decisions by business people and technical decisions by technical people.

A critical part of the method is providing assurance on the correctness of the transformational implementation of a business process. Two approaches to the verification of the process correctness are discussed in this paper. The first one applies a correct-by-construction approach that consists in defining a set of safe transformations, which do not change the process behavior. If all transformations are safe, then the transformed program will also be correct, i.e., semantically equivalent to the original reference process.

The other approach relies on automatic verification of the transformed process behavior against the behavior of the original 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.

The rest of this paper is organized as follows. Related work is briefly surveyed in Section II. The semantics of a BPEL process and its behavior are defined in Section III. A set of safe transformations are introduced in Section IV. An illustrative case study is provided in Section V. A method for the verification of correctness, based on LOTOS language and a BPEL to LOTOS mapping is covered in Section VI. Quality metrics to assess transformation results are described in Section VII. Conclusions and plans for future research are given in Section VIII.

II. Related Work

Transformational implementation of software is not a new idea. The approach was developed many years ago within the context of monolithic systems, with the use of several executable specification techniques. The formal foundation was based on problem decomposition into a set of concurrent processes, use of functional languages [7] and formal modeling by means of Petri nets [8].

An approach for transformational implementation of business processes was developed in [9]. This four-phase approach is very general and not tied to any particular technology. Our method, which can be placed in the fourth phase (implementation), is much more specific and focused on the implementation of runnable processes described in BPMN and BPEL.

BPMN defines a model and a graphical notation for describing business processes, standardized by OMG [3]. The reference model of SOA [10,11] and the specification of BPEL [5] are standardized by OASIS. An informal mapping of BPMN to BPEL was defined in [3]. A comprehensive discussion of the translation between BPMN and BPEL, and of some conceptual discrepancies between the languages, can be found in [12,13]. An open-source tool is available for download at [14].

The techniques of building program dependence graph and program slicing, which we adopted for proving safeness of transformations, were developed in [15,16] and applied to BPEL programs in [17].

Several metrics to measure the quality of parallel programs have been proposed in the literature and studied for many years. A traditional metric for measuring performance of a parallel application is Program Activity Graph, which describes parallel flow of control within the application [18]. We do not use such a graph, nevertheless, our two metrics: Length of thread and Response time, can be viewed as an approximation of Critical path metric described in [18]. Similarly, our Number of threads metric is similar to Available concurrency defined in [19].

To the best of our knowledge, our work on the implementation of business processes in service oriented architecture is original. Preliminary results of our research were published in [1]. An extended version, including a revised algorithm for building program dependence graph and an original method for the verification of transformation correctness are introduced in this paper.

III. The Semantics of a Business Process

A business process is a collection of logically related activities, performed in a specific order to produce a service or product for a customer. The activities can be implemented on-site, by local data processing tasks, or 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.

A specification of a business process can be defined textually, e.g., using a natural language, or graphically, using BPMN. An example BPMN process, which executes a simplified processing of a bank transfer order is shown in Fig. 1. The process begins and waits for an external invocation from a remote client (another process). When the invocation is received, the process extracts the source and the target account numbers from the message, checks the availability of funds at source and splits into two alternative branches. If the funds are missing, the process prepares a negative acknowledgement message, replies to the invoker, and ends. Otherwise, the alternative branch is empty. Then, the process invokes the withdraw service at source account, invokes the deposit service at target account, packs the results returned by the two services into a single reply message, replies to the invoker and ends. This way, the process implements a service, which is composed of another services.

BPMN specification of a business process can be automatically translated into a BPEL program, which can be used for a semi-automatic implementation.

BPEL syntax is composed of a set of instructions, called activities, which are XML elements indicated in the document by explicit markup. The set of BPEL activities is rich. However, in this paper, we focus on a limited subset of activities for defining control flow, 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 are composed of 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. 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

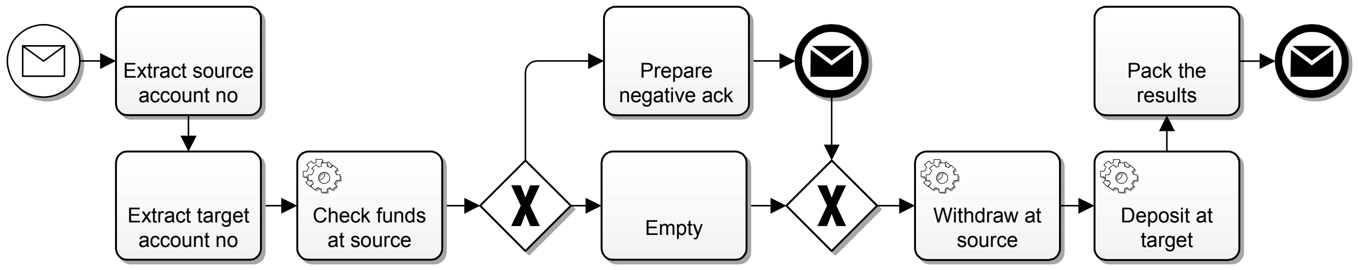

Figure 1. BPMN specification of a business process

```
<sequence>
   <receive name="rcv" variable="transfer"/>
   <assign name="src">
      <copy> <from variable="transfer" part="srcAccNo"/>
      <to variable="source" part="account"/> </copy>
      <copy> <from variable="transfer" part="srcAmount"/>
      <to variable="source" part="amount"/> </copy>
   </assign>
   <assign name="dst">
      <copy> <from variable="transfer" part="dstAccNo"/>
      <to variable="target" part="account"/> </copy>
      <copy> <from variable="transfer" part="dstAmount"/>
      <to variable="target" part="amount"/> </copy>
   </assign>
   <invoke name="verify" inputVariable="source"
      outputVariable="fundsAvailable"/>
   <if> <condition> $fundsAvailable.res </condition>
      <empty name="empty"/>
   <else> <sequence>
      <assign name="fail">
         <copy> <from> 'lack of funds' </from>
         <to variable="response" part="fault"/> </copy>
      </assign>
      <reply name="nak" variable="response"/>
      <exit name="exit"/>
   </sequence> </else> </if>
   <invoke name="withdraw" inputVariable="source"
         outputVariable="wResult"/>
   <invoke name="deposit" inputVariable="target"
         outputVariable="dResult"/>
   <assign name="success">
      <copy> <from variable="wResult" part="res"/>
      <to variable="result" part="withdraw"/> </copy>
      <copy> <from variable="dResult" part="res"/>
      <to variable="result" part="deposit"/> </copy>
   </assign>
   <reply name="ack" variable="result"/>
</sequence>
```

Figure 2. A skeleton of a BPEL program of a bank transfer (Fig. 1)

shown in Fig. 2. 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 and sends 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 can be viewed as a service invoked by other services.

SOA services are assumed stateless [20], which means that the result of a service execution depends only on values of data passed to the service at the invocation, and manifests to the outside world as values of data sent by the service in response to the invocation. Therefore, we assume that the observable behavior of a process in a SOA environment consists of data values, which the process passes as arguments when it invokes external services, and data values, which it sends in reply to the invoker.

A. Program Dependence Graph

To capture the influence of a process structure into the process behavior, we use a technique called program slicing [15,16], which allows finding all the instructions in a program, which influence the value of a variable in a specific point of the program. For example, finding the instructions that influence the value of a variable that is used as an argument by a service invocation activity or by a reply activity of the process.

The conceptual tool for the analysis is Program Dependence Graph (PDG), whose nodes are activities of a BPEL program, and edges reflect dependencies between the activities. An algorithm for constructing PDG of a BPEL program consists of the following steps:

1. Define nodes of the graph, which are activities at all layers of nesting.
2. Define control edges (solid lines in Fig. 3), which follow the nested structure of the program, e.g., an edge from <sequence> to <if> shows that <if> activity is nested within the <sequence> element. Output edges of <if> node are labeled "Yes" or "No", respectively.
3. Define dataflow edges (dashed lines in Fig. 3), which reflect dataflow dependencies between the activities, e.g., an edge from activity "rcv" to activity "src" shows that an output variable of "rcv" is used as input variable to "src".
4. Add dataflow edges from <receive> activity, which is nested within a <sequence> element, to each subsequent activity of this <sequence> such that no paths from <receive> to this activity exists (there are no such items in Fig. 3).
5. If an <exit> activity is nested within a <sequence>, then:
   a. remove all the activities, which are subsequent to <exit>, together with all the input and output edges,

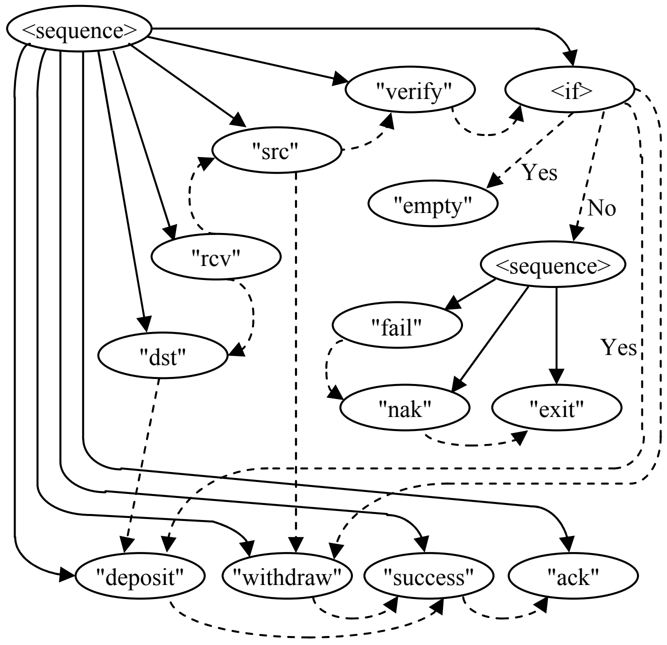

Figure 3. Program dependence graph of the bank transfer process

   b. for each antecedent activity with no path to <exit>, add a dataflow edge from this activity to <exit> ("nak" to "exit" edge in Fig. 3).
6. If an <if> element is nested within a <sequence> and there is an <exit> within "Yes" ("No") branch of <if>, then add "No" ("Yes") edges from <if> to subsequent activities with no path from <if> (<if> to "deposit" and <if> to withdraw edges in Fig. 3).
7. Convert "Yes" and "No" edges that output <if> activities into dataflow edges.

Dataflow edges within a program dependence graph reflect the dataflow dependencies between subsequent activities, which determine values of the program variables. The edges added in step 4 reflect the semantics of the process as a service, which starts after receiving an invocation message. The edges added in steps 5 and 6 reflect the semantics of <exit>, which stops the program and prevents execution of all the subsequent activities. Dataflow edges introduced in step 7 reflect the semantics of <if> statement, which outgoing branches may execute only after evaluating the condition. An example program dependence graph of the business process in Fig. 2 is shown in Fig. 3. It can be noted, that the flow of control within the original BPEL program complies with dataflow edges of its program dependence graph.

In the rest of this paper, we adopt a definition that a transformation preserves the process behavior, if it keeps the set of messages sent by the process as well as the data values carried by these messages unchanged. Such a definition neglects the timing aspects of the process execution. This is justified, given that it does not change the business requirements. There are many delays in a SOA system and the correctness of software must not relay on a specific order of activities, unless they are explicitly synchronized.

A transformation that preserves the process behavior is called safe.

Definition (Safeness of a transformation)

A transformation is safe, if the set of messages sent by the activities of a program remains unchanged and the flow of control within the transformed program complies with the direction of dataflow edges within the program dependence graph of the reference process. □

The set of activities executed within a program may vary, depending on decisions made when passing through decision points of <if> activities. To fulfill the above definition, the set of messages must remain unchanged, for each particular combination of these decisions.

A path composed of dataflow edges in a program dependence graph reflects the dataflow relationships between the activities, and implies that the result of the activity at the end of the path depends only on the preceding activities on this path. If the succession of activities executed within a program complies with the dataflow edges, then the values of variables computed by the program remain the same, regardless of the ordering of other activities of this program.

Safeness of a transformation guarantees that the transformation preserves the behavior of the transformed program as observed by other services in a SOA environment. Safeness is transitive and a sequence of safe transformations is also safe. Therefore, a process resulting from a series of safe transformations applied to a reference process preserves the behavior of the reference process.

IV. Safe Transformations

The body of a BPEL process consists of simple activities, e.g., <assign>, which define elementary pieces of computation, and structured elements, e.g., <flow>, which is composed of other simple or structured activities. The behavior of the process results from the order of execution of activities, which stem from the type of structured elements and the positioning of activities within these elements. A transformation applies to a structured element and consists in replacing one element, e.g., <flow>, by another element, e.g., <sequence>, or in relocation of activities within the structured element. If the behavior of the transformed element before and after the transformation is the same, then the behavior of the process stands also unchanged.

Several transformations have been defined. The basic ones: simple and alternative displacement, parallelization and serialization of the process operations, and process partitioning are described in detail below. All the transformations are safe, according to definition of safeness given in Section III. As pointed out in Section III, a safe transformation does not change the behavior of a process, which is composed of stateless services. A problem may arise, if the services invoked by a process have an impact on the real world. If this is the case, a specific ordering of these services may be required. In our approach, a designer can express the necessary ordering conditions adding supplementary edges to the program dependence graph.

Transformation 1: Simple displacement

Consider a <sequence> element, which contains n arbitrary activities executed in a strictly sequential order. Transformation 1 moves a selected activity A from its original position i, into position j within the sequence.

***Theorem* 1.** Transformation 1 is safe, if no paths between activity A and the activities placed on positions i+1, … j in the sequence existed in the program dependence graph of the transformed program.

Proof: Assume that $i < j$ (move forward). The transformation has no influence on activities placed on positions lower than i or higher than j. However, moving activity A from position i to j reverts the direction of the flow of control between A and the activities that are in-between. This could be dangerous if a dataflow from A to those activities existed. However, if no dataflow paths from A to the activities placed on positions i+1, … j existed in the program dependence graph, then no inconsistency between the control and data flow can exist.

If $j < i$ (move backward), the proof is analogous. The lack of dataflow path guarantees lack of inconsistency between the data and control flows within the program. □

Transformation 2: Pre-embracing

Consider a <sequence> element, which includes an <if> element preceded by an <assign> activity, among others. Branches of <if> element are <sequence> elements. Transformation 2 moves <assign> activity from its original position in the outer <sequence>, into the first position within one branch of <if> element.

***Theorem* 2.** Transformation 2 is safe, if neither a path from the moved <assign> to an activity placed in the other branch of <if>, nor a path from the moved <assign> to the activities positioned after <if> in the outer sequence, existed in the program dependence graph of the transformed program.

Proof: The transformation has no influence on activities placed prior to <if> element in the outer <*sequence*>. Moving <assign> activity to one branch of <if> removes the flow of control from <assign> to activities in the other branch of <if> and – possibly – to activities placed after <if>. But according to the assumption of this theorem, there is no data flow between these activities. Therefore, no inconsistency between the control and data flow can exist. □

Transformation 3: Post-embracing

Consider a <sequence> element, which includes an <if> activity followed by a number of another activities. Branches of <if> element are <sequence> elements, one of which contains <exit> activity. Transformation 3 moves the activities, which follow <if>, from its original position in the outer <sequence> into the end of the second <sequence> of <if> element.

***Theorem* 3.** Transformation 3 is safe.

Proof: Activities, which are placed after an <if> element in the reference process, are executed only after the execution of <if> is finished. The existence of <exit> in one branch of <if> prevents execution of these activities when this branch is selected. The activities are executed only in case the other branch is selected. Therefore, neither the flow of control nor the flow of data is changed in the program, when the activities are moved to the other branch of <if>, i.e., the branch without <exit> activity. □

Transformation 4: Parallelization

Consider a <sequence> element, which contains *n* arbitrary activities executed in a strictly sequential order. Transformation 4 parallelizes the execution of activities by replacing <sequence> element by <flow> element composed of the same activities, which – according to the semantics of <flow> – are executed in parallel.

***Theorem* 4.** Transformation 4 is safe, if for each pair of activities A_i, A_j neither a path from A_i to A_j nor a path from A_j to A_i existed in the program dependence graph of the transformed program.

Proof: The transformation changes the flow of control between the activities of the transformed element. The lack of dataflow paths between these activities means that no inconsistency between the control and data flow can exist. □

Transformation 5: Serialization

Consider a <flow> element, which contains *n* arbitrary activities executed in parallel. Transformation 5 serializes the execution of activities by replacing <flow> element by <sequence> element, composed of the same activities, which are now executed sequentially.

```
<invoke name="xxx"                                          (a)
   inputVariable="source"  outputVariable="target"
/>

<sequence>                                                  (b)
   <invoke name="xxx_i"  inputVariable="source"/>
   <receive name="xxx_r"  variable="target"/>
</sequence>
```

Figure 4. Synchronous (a) and asynchronous service invocation (b)

***Theorem* 5.** Transformation 5 is safe.

Proof: The proof is obvious. Parallel commands can be executed in any order, also sequentially.

Transformation 6: Asynchronization

Consider a two-way <invoke> activity, which sends a message to invoke an external service and then waits for a response (Fig. 4a). Transformation 6 replaces the two-way <invoke> activity with a sequence of a one-way <invoke> activity followed by a <receive> (Fig. 4b). This way, a synchronous invocation of a service is converted into an asynchronous one.

Transformation 6 can be proved safe, if we add a dataflow edge from <invoke> node to <receive> node in the program dependence graph of each program, which includes an asynchronous service invocation shown in Fig. 4b.

***Theorem* 6.** Transformation 6 is safe.

Proof: The transformation has no influence on activities executed prior to <invoke> activity. Dataflow edges from these activities to <invoke> remain unchanged. The transformation has no influence on activities executed after <invoke>, as well. Dataflow edges to these activities from <invoke> are redirected to begin at node <receive>. Hence, there is a one-to-one mapping between the sets of dataflow paths, which exist in program dependence graph of a program before and after the transformation. Therefore, no inconsistency between the control and data flow can exist.

Transformations 1 through 6 can be composed in any order, resulting in a complex transformation of the process structure. Transformations 7 and 8 play an auxiliary role and facilitate such a composition. These transformations are safe,

```
<sequence>          (a)           <flow>                (b)
   <sequence>                        <flow>
      <C1> </C1>                        <C1> </C1>
      ......                            ......
      <Ck> </Ck>                        <Ck> </Ck>
   </sequence>                       </flow>
   <sequence>                        <flow>
      <Ck+1> </Ck+1>                    <Ck+1> </Ck+1>
      ......                            ......
      <Cn> </Cn>                        <Cn> </Cn>
   </sequence>                       </flow>
</sequence>                       </flow>
```

Figure 5. Sequential (a) and parallel (b) partitioning of commands

because they do not change the order of execution of any activities within a BPEL program. □

Transformation 7: Sequential partitioning

Transformation 7 divides a single <sequence> element into a nested structure of <sequence> elements (Fig. 5a).

Transformation 8: Parallel partitioning

Transformation 8 divides a single <flow> element into a nested structure of <flow> elements (Fig. 5b).

V. Case Study

Consider a process of transferring funds between two different bank accounts, shown in Fig. 1, implemented by a BPEL process. A skeleton of the simplified BPEL program of this process is shown in Fig. 2.

The process body is a sequence of activities, which starts at <receive>. Then, it proceeds through a series of steps to process the received bank transfer order and to invoke services offered by the banking systems to verify availability of funds at source account, to withdraw funds and to deposit the funds at the destination account. Finally, it ends after replying positively, if the transfer has successfully been done, or negatively, if the required amount of funds was not available at source.

PDG of this program is shown in Fig. 3. The first two <*assign*> activities process the contents of the received message in order to extract the source and destination account numbers and the amount of money to transfer. Therefore, there are dataflow edges from "rcv" to "src" and to "dst" nodes in PDG. The next consecutive <invoke> activity uses the extracted source account number and the amount of money to invoke the verification service, and the response of the invocation is checked by <if> activity. Therefore, two dataflow edges from *src* to *verify* and from *verify* to <if> exist in the graph. Similarly, the <invoke> activities named "withdraw" and "deposit" use the account numbers calculated by "src" and "dst", respectively. Two dataflow edges from "withdraw" and "deposit" nodes to "success" node, and then an edge from "success" to "ack", reflect the path of preparing the acknowledgement message that is sent to the invoker when the transfer is finished.

The analysis of the program dependence graph in Fig. 3 reveals that no dataflow path between activity named "dst" and the next two activities "src" and "verify" exists in the graph. Therefore, these activities can be executed in parallel. Similarly, there is no dataflow path between two consecutive <invoke> activities "withdraw" and "deposit". These two activities can also be executed in parallel.

To perform these changes, we can partition the outer <sequence> element using transformation 6 three times, and then parallelize the program structure using transformation 4 twice. A skeleton of the resulting BPEL program is shown in Fig. 6. Only names of the activities are shown in Fig. 6. The variables used by the activities are omitted for brevity.

However, this is not the only way of transformation. Alternatively, the designer can displace "dst" forward, just before <if> activity, and then use transformation 2 in order to enter "dst" to the inside of <if> in place of <empty> activity. Next, transformation 3 can be used to embrace the last three

```
<sequence>
   <receive name="rcv">              - receive transfer order
   <flow>
      <assign name="dst">            - extract destination no
      <sequence>
         <assign name="src">         - extract source no
         <invoke name="verify">      - verify funds at source
      </sequence>
   </flow>
   <if>
      <condition> ... </condition>   - check availability
         <empty name="empty">        - do nothing if available
      <else> <sequence>
         <assign name="fail">        - set response
         <reply name="nak">          - reply negatively
         <exit name="exit">          - end of execution
      </sequence> </else>
   </if>
   <flow>
      <invoke name="withdraw">       - withdraw funds
      <invoke name="deposit">        - deposit funds
   </flow>
   <assign name="success">
   <reply name="ack">                - reply positively
</sequence>
```

Figure 6. A skeleton of the transformed bank transfer process – variant I

activities of the outer <sequence> element into the first branch of <if> element, consecutively following "dst". Then, the designer can move "dst" forward, adjacent to "deposit", partition the inner sequence of <if> using transformation 6, and parallelize the program structure using transformation 4. A skeleton of the resulting BPEL program is shown in Fig. 7. We removed "exit" activity from the final program, because it is obviously redundant at the end of the program.

```
<sequence>
   <receive name="rcv">                  - receive order
   <assign name="src">                   - extract source no
   <invoke name="verify">                - verify funds
   <if>
      <condition> ... </condition>       - check availability
      <sequence>
         <flow>
            <invoke name="withdraw">     - withdraw funds
            <sequence>
               <assign name="dst">       - extract dst. no
               <invoke name="deposit">   - deposit funds
            </sequence>
         </flow>
         <assign name="success">
         <reply name="ack">              - reply positively
      </sequence>
      <else> <sequence>
         <assign name="fail">            - set response
         <reply name="nak">              - reply negatively
      </sequence> </else>
   </if>
</sequence>
```

Figure 7. A skeleton of the transformed bank transfer process – variant II

The main advantage of the transformed process over the original one is higher level of parallelism, which can lead to better performance of the program execution. If we compare the two alternative designs, then intuition suggests that the structure of the second process is better than of the first one. In order to verify this impression, the reference process and the transformed processes can be compared to each other, with respect to a set of quality metrics. Depending on the results, the design phase can stop, or a selected candidate (a transformed process) can be substituted as the reference process for the next iteration of the design phase.

VI. Verification of Correctness

The correct-by-construction approach is appealing for the implementation designer because it can open the way towards automatic process optimization. However, the approach has also some practical limitations. It is possible that small changes to a process behavior can be acceptable within the application context. If this was the case, then a verification method is needed, capable not only of verifying the process behavior, but also showing the designer all the potential changes, if they exist. In this section, we introduce LOTOS language, which is used as a formal basis for such a verification method.

A. The language LOTOS

Language of Temporal Ordering Specification (LOTOS) is one of the formal description techniques developed within ISO [21] 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 μ to denote an arbitrary action and δ 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 that 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., $g1$, $g2$, 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 $\boldsymbol{i}$ 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

TABLE I. Expressions in basic LOTOS

Syntax	Explanation
stop	inaction, lack of action
$\mu; B$	action μ precedes execution of expression B
$B1$ [ ] $B2$	alternative choice of expressions $B1$ and $B2$
$B1 \mid[g1,\ldots gn]\mid B2$	parallel execution of $B1$ and $B2$ synchronized at actions $g1,\ldots,gn$
$B1 \mid\mid\mid B2$	parallel execution with no synchronization between $B1$ and $B2$
exit	successful termination; generates a special action $\boldsymbol{\delta}$
$B1 >> B2$	sequential composition: successful execution of $B1$ enables $B2$
$B1$ [> $B2$	disabling: successful execution of $B1$ disables execution of $B2$
hide $g1,\ldots,gn$ **in** B	hiding: actions $g1,\ldots,gn$ are transformed into unobservable ones

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 $B \overset{\mu}{\rightarrow} B'$. For example, the semantics of expression $(g; B1)$ can be described by a labeled transition:

$$g; B1 \overset{g}{\rightarrow} B1$$

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

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**

Exceptions are BPEL <empty>, which is mapped to **exit**, and <exit>, which is mapped to **stop**.

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 ([ ]).

The semantics of parallelism in LOTOS is interleaved. Parallel execution of activities that are nested within <flow> element of a BPEL process is modeled by the possibility of executing the corresponding LOTOS actions 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 δ generated by **exit** is not counted in the execution traces because it is an unobservable action.

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** >> *src*;**exit** >> *dst*;**exit** >> *verify*;**exit** >>
(**exit** [ ] *fail*;**exit** >> *nak*;**exit** >> **stop**) >>
withdraw;**exit** >> *deposit*;**exit** >> *success*;**exit** >> *ack*;**exit**

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

rcv; *src*; *dst*; *verify*; *withdraw*; *deposit*; *success*; *ack*
rcv; *src*; *dst*; *verify*; *fail*; *nak*

B. The Verification Method

The verification follows a two-phase approach illustrated in Fig. 8, where B2L acronym stands for: BPEL-to-LOTOS mapping. In the first phase, dataflow 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 reduced program dependence 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. The reduced program dependence graph is then transformed into a LOTOS 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.

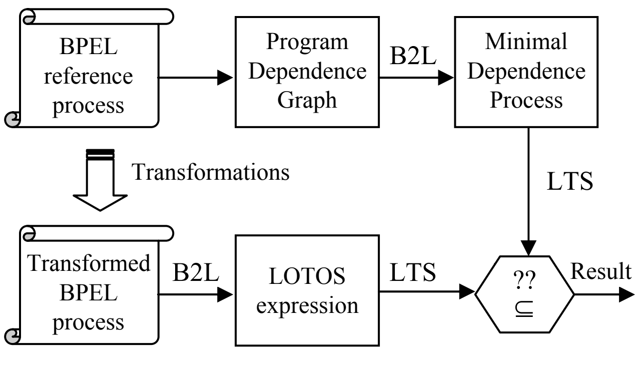

Figure 8. Verification of a process behavior

The second phase is performed repetitively during the transformational implementation cycle. A transformed BPEL process is mapped into a LOTOS expression, as described in the previous subsection. 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.

C. The Reduced Program Dependence Graph

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, <sequence> and <flow> nodes are removed from the program dependence graph. The reduced program dependence graph of the reference process in Figs. 2 and 3 is shown in Fig. 9. An algorithm for removing the nodes consists of the following steps:

1. Remove all <sequence> and <flow> nodes, which are not directly nested within an <if> element. Redirect control edges, which output each removed node, to the direct predecessor node if one exists, or remove otherwise.
2. If a <sequence> node is nested within an <if>, then:
   a. Remove <sequence> node together with the input "Yes" ("No") edge.
   b. Add dataflow edges labeled "Yes" ("No") from <if> node to each member of <sequence> such that no path from <if> to this node exists (<if> to "fail" edge in Fig. 9).

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, regardless of the dataflow relation between the invoking activities. A designer can reflect this requirement adding supplementary edges between the appropriate nodes of the reduced program dependence graph.

D. Minimal Dependence Process

Let $G_P = (N_P, E_P)$ be a reduced program dependence graph of a BPEL process P. It can be proved that graph G_P is acyclic. We say that node n_i precedes node n_j, denoted

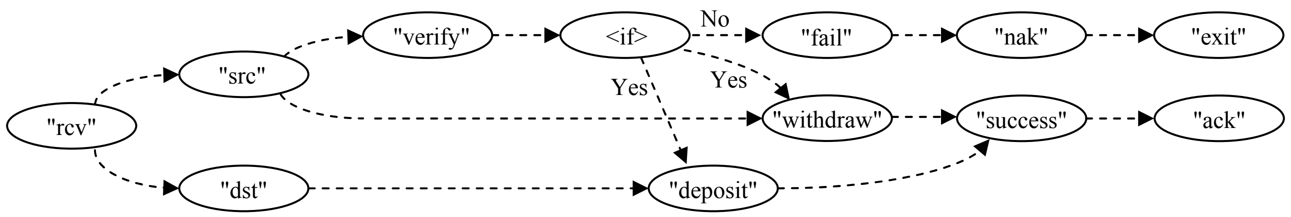

Figure 9. The reduced program dependence graph of the process in Figs. 2 and 3

$n_i < n_j$, if there exists a path from n_i to n_j in the reduced program dependence graph. Precedence relation is a strict partial order in N_P.

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 edges. 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 edges may vary. For example, the first node in Fig. 9 is *rcv*. After visiting this node, data can be passed along the edge to *src* or along the edge to *dst*. If the former is true, then in the next step either node *src* can be visited or data can be passed along the edge to *dst*. However, node *dst* could not be visited before the data were passed through its incoming edge.

Nodes and edges 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 edge 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 edges is shown in Fig. 10.

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 another 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 edges 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 edge.

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. 10 is *rcv*, followed by one of the actions *a* or *b*. Hence, the appropriate LOTOS expression begins with:

$rcv;\ (a|||b)\ \ldots$

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

$(\ (rcv;\ (a||b)\ )\ |[a]|\ a;\ src;\ldots)\ |[b]|\ b;\ dst;\ldots$

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, let us consider the reduced program dependence graph in Fig. 9. 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. 10.

The minimal dependence process derived from the graph in Fig. 10 takes the form of the following LOTOS expression:

$$(((((((rcv;(a|||b)) \,|[a]|\, a;src;(c|||d)) \,|[b]|\, b;dst;e) \\ |[c]|\, c;verify;(y1;y2[\,]n)) \,|[d,y1]|\, d;y1;withdraw;f\,) \\ |[e,y2]|\, e;y2;deposit;g) \,|[f,g]|\, f;g;success;ack) \\ |[n]|\, n;fail;nak;exit$$

The labeled transition system of this expression generates a set of 13 traces, each of which is a sequence of observable actions:

{ *rcv*; *dst*; *src*; *verify*; *fail*; *nak*; *exit* ,
rcv; *src*; *dst*; *verify*; *fail*; *nak*; *exit* ,
rcv; *src*; *verify*; *dst*; *fail*; *nak*; *exit* ,
rcv; *src*; *verify*; *fail*; *dst*; *nak*; *exit* ,
rcv; *src*; *verify*; *fail*; *nak*; *dst*; *exit* ,
rcv; *src*; *verify*; *fail*; *nak*; *exit*; *dst* ,
rcv; *dst*; *src*; *verify*; *withdraw*; *deposit*; *success*; *ack* ,
rcv; *dst*; *src*; *verify*; *deposit*; *withdraw*; *success*; *ack* ,
rcv; *src*; *dst*; *verify*; *withdraw*; *deposit*; *success*; *ack* ,
rcv; *src*; *dst*; *verify*; *deposit*; *withdraw*; *success*; *ack* ,
rcv; *src*; *verify*; *dst*; *withdraw*; *deposit*; *success*; *ack* ,
rcv; *src*; *verify*; *dst*; *deposit*; *withdraw*; *success*; *ack* ,
rcv; *src*; *verify*; *withdraw*; *dst*; *deposit*; *success*; *ack* }

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

{ *rcv*; *src*; *dst*; *verify*; *fail*; *nak*; *exit* ,
rcv; *src*; *dst*; *verify*; *withdraw*; *deposit*; *success*; *ack* }

The trace set of the first transformed process in Fig. 6 consists of 9 traces:

{ *rcv*; *dst*; *src*; *verify*; *fail*; *nak*; *exit* ,
rcv; *src*; *dst*; *verify*; *fail*; *nak*; *exit* ,
rcv; *src*; *verify*; *dst*; *fail*; *nak*; *exit* ,
rcv; *dst*; *src*; *verify*; *withdraw*; *deposit*; *success*; *ack* ,
rcv; *dst*; *src*; *verify*; *deposit*; *withdraw*; *success*; *ack* ,
rcv; *src*; *dst*; *verify*; *withdraw*; *deposit*; *success*; *ack* ,
rcv; *src*; *dst*; *verify*; *deposit*; *withdraw*; *success*; *ack* ,
rcv; *src*; *verify*; *dst*; *withdraw*; *deposit*; *success*; *ack* ,
rcv; *src*; *verify*; *dst*; *deposit*; *withdraw*; *success*; *ack* }

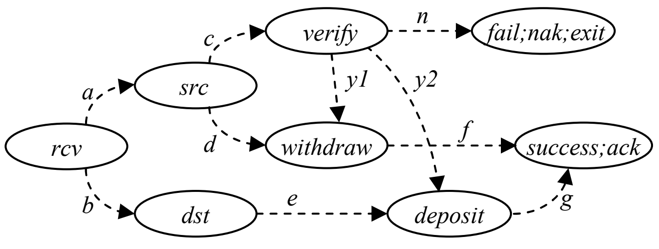

Figure 10. Construction of MDP: The reduced program dependence graph (Fig. 9) after step 3

The trace set of the second transformed process in Fig. 7 consists of 4 traces:

{ *rcv*; *src*; *verify*; *fail*; *nak* ,
rcv; *src*; *verify*; *dst*; *withdraw*; *deposit*; *success*; *ack* ,
rcv; *src*; *verify*; *dst*; *deposit*; *withdraw*; *success*; *ack* ,
rcv; *src*; *verify*; *withdraw*; *dst*; *deposit*; *success*; *ack* }

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.

VII. QUALITY METRICS

Many metrics to measure various characteristics of software have been proposed in literature [18,19]. In this research, we use simple metrics that characterize the size of a BPEL process, the complexity and the degree of parallel execution. The value of each metric can be calculated using a program dependence graph.

The size of a process is measured as the number of simple activities in a BPEL program. More precisely, the value of this metric equals the number of leaf nodes in the program dependence graph of a BPEL process. For example, the size of the processes shown in Figs. 2 and 6 is 12, while the size of the process in Fig. 7 equals 10.

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. However, smaller number of this metric may result from removing excessive, unstructured activities, like <empty> and <exit>. This is the case of program in Fig. 7.

The complexity of a process is measured as the total number of nodes in PDG. For example, The complexity of the process shown in Fig. 2 is 15, the complexity of the process in Fig. 6 is 18, and the complexity of the process in Fig. 7 is 16.

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 the process complexity.

The number of threads is measured as the number of items within <flow> elements of a BPEL program, at all levels of nesting. A problem with this metric is such that the number of executed items can vary, depending on values of conditions within <if> elements. Therefore, the metric is a vector of values, computed for all combinations of values of these conditions. 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> element is the maximum of weights assigned to the descending nodes (i.e.,

TABLE II. NUMBER OF THREADS METRIC

if - condition	Process in Fig. 2	Process in Fig. 6	Process in Fig. 7
YES	1	2	2
NO	1	2	1

nodes directly nested within the <sequence> element),

- the weight of an <if> element is the weight assigned to the activity in this branch of <if>, which is executed according to a given value of condition within the <if> element.

The number of executed items can be influenced also by the presence of <exit> activity, which ends the process execution. Therefore, the nodes directly nested within a <sequence> element are ordered in compliance with the order of execution. Nodes subsequent to a node, which is, or which comprises, <exit> activity, are not taken into account in the computation.

The metric value equals the weight assigned to the top <sequence> node of PDG. Values of the metric for the processes in Figs. 2, 6, and 7 are shown in Table I. Program dependence graph and calculation of the metric for the program in Fig. 7 is shown in Fig. 11 (grey numbers right to the nodes).

The length of thread is measured as the number of sequentially executed activities within a BPEL program. Because the number of executed items can vary, depending on values of conditions within <if> elements, the metric is a vector of values, computed for all combinations of values of these conditions. 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 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 weight of an <if> element is the weight assigned to the activity in this branch of <if>, which is executed according to a given value of condition within the <if> element.

Nodes directly nested within a <sequence> element are ordered in compliance with the order of execution. Nodes subsequent to a node, which is, or which comprises, <exit> activity, are not taken into account in the computation.

The metric value equals the weight assigned to the top <sequence> node of PDG. Values of the metric for the processes in Figs. 2, 6, and 7 are shown in Table II.

TABLE III. LENGTH OF THREAD METRIC

if - condition	Process in Fig. 2	Process in Fig. 6	Process in Fig. 7
YES	9	7	7
NO	7	6	5

TABLE IV. RESPONSE TIME METRIC

if - condition	Process in Fig. 2	Process in Fig. 6	Process in Fig. 7
YES	36	25	25
NO	16	15	14

The response time is measured as the sum of estimated execution times of activities, which are sequentially executed within a BPEL program. Because the number of executed items can vary, depending on values of conditions within <if> elements, the metric is a vector of values, computed for all combinations of values of these conditions The algorithm of computation is identical to the algorithm of computation of the length of thread metric, except of the first point, which now reads:

- the weight of a simple activity is the estimated execution time of this activity,

In the simplest case, the estimated execution time can just differentiate between local data manipulation activity and a service invocation. Values of the metric for the processes in Figs. 2, 6, and 7, calculated under an assumption that a local data manipulation time equals 1, while a service execution time equals 10, are shown in Table III. Program dependence graph and calculation of the metric for the program in Fig. 7 is shown in Fig. 11 (numbers left to the nodes).

Comparing the values of metrics calculated for the processes considered in the case study in Section V, one can note that both transformed processes are faster than the original reference process (smaller value of the response time

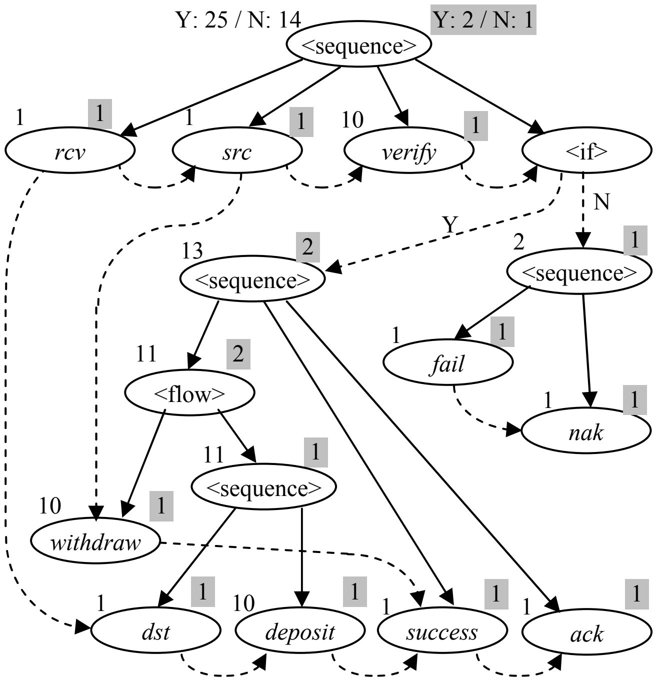

Figure 11. Program dependence graph of the program in Fig. 7 and calculation of metrics: Number of threads (grey numbers right to the nodes) and length of execution (left to the nodes)

metric). Speeding up the process execution is a benefit from parallel invocation of services in a SOA environment. Comparing the variants of the transformed bank transfer process (Fig. 6 and Fig. 7), one can note that the second variant is a bit faster and simpler (smaller values of the size metrics). This variant can be accepted by the customer or used as a new reference process in the next transformation cycle.

VIII. Conclusion and Future Work

Defining the behavior of a business process is a business decision. Defining the implementation of a business process on a computer system is a technical decision. The transformational method for implementing business processes in a service oriented architecture, described in this paper, promotes separation of concerns and allows making business decisions by business people and technical decisions by technical people.

The transformations described in this paper are correct by construction in that they do not change the behavior of a transformed process. However, the transformations change the process structure in order to improve efficiency and benefit from the parallel execution of services in a SOA environment. The quality characteristics of the process implementation are measured by means of quality metrics, which account for the process size, complexity and the response time of the process as a service. Other quality features, such as modifiability or reliability, are not covered in this paper.

The correct-by-construction approach opens the way towards automatic process optimization. However, the approach has also some practical limitations. If the external services invoked by a process have an impact on the real world, as is usually the case, a specific ordering of these services may be required, regardless of the dataflow dependencies between the service invocation activities within a program. In our approach, a designer can express the necessary ordering conditions adding supplementary edges to the program dependence graph. Therefore, the approach cannot be fully automated and a manual supervision over the transformation process is needed.

It is also possible that small changes to a process behavior can be acceptable within the application context. Therefore, part of our research was aimed at finding a verification method capable not only of verifying the process behavior, but also of showing the designer all the potential changes, if they exist. A decision on whether to accept the changes or not is made by a human.

References

[1] K. Sacha and A. Ratkowski, “Implementation of Business Processes in Service Oriented Architecture,” Proc. The Seventh International Conference on Software Engineering Advances (ICSEA 2012), IARIA, 2012, pp. 129–136.

[2] T. H. Davenport and J. E. Short, “The New Industrial Engineering: Information Technology and Business Process Redesign,” Sloan Management Review, 1990, pp. 11–27.

[3] OMG, “Business Process Model and Notation (BPMN), Version 2.0,” Jan. 2011, http://www.omg.org/spec/BPMN/2.0/PDF/,10.06.2013.

[4] A. W. Scheer, ARIS – Business Process Modeling, Springer, Berlin Heidelberg, 2007.

[5] D. Jordan and J. Evdemon, “Web Services Business Process Execution Language Version 2.0,” OASIS Standard, Apr. 2007, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, 10.06.2013.

[6] OMG, “OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2,” Nov. 2007, http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF, 10.06.2013.

[7] P. Zave, “An Insider's Evaluation of Paisley,” IEEE Trans. Software Eng., vol. 17, 1991, pp. 212–225.

[8] K. Sacha, “Real-Time Software Specification and Validation with Transnet,” Real-Time Systems J., vol. 6, 1994, pp. 153–172.

[9] F. J. Duarte, R. J. Machado, and J. M. Fernandes, “BIM: A methodology to transform business processes into software systems,” SWQD 2012, LNBIP vol. 94, 2012, pp. 39–58.

[10] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, and R. Metz, “Reference Model for Service Oriented Architecture 1.0,” OASIS Standard, Oct. 2006, http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html, 10.06.2013.

[11] J. A. Estefan, K. Laskey, F. G. McCabe, and D. Thornton, “Reference Architecture for Service Oriented Architecture Version 1.0,” OASIS Public Review Draft 1, Apr. 2008, http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra-pr-01.pdf, 10.06.2013.

[12] S. A. White, “Using BPMN to Model a BPEL Process,” BPTrends 3, 2005, pp. 1–18.

[13] J. Recker and J. Mendling, “On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages,” The 18th International Conference on Advanced Information Systems Engineering (CAISE 2006), Proc. Workshops and Doctoral Consortium, Namur University Press, 2006, pp. 521–532.

[14] Bpmn2bpel, “A tool for translating BPMN models into BPEL processes,” http://code.google.com/p/bpmn2bpel/, 10.06.2013

[15] M. Weiser, “Program slicing,” IEEE Trans. Software Eng., vol. 10, 1984, pp. 352–357.

[16] D. Binkley and K. B. Gallagher, “Program slicing,” Advances in Computers, 43, 1996, pp. 1–50.

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

[18] J. K. Hollingsworth and B. P. Miller, “Parallel program performance metrics: A comparison and validation,” Proc. ACM/IEEE Conference on Supercomputing (SC 92), IEEE Computer Society Press, pp. 4–13.

[19] A. S. Van Amesfoort, A. L. Varbanescu, and H. J. Sips, “Parallel Application Characterization with Quantitative Metrics,” Concurrency and Computation: Practice and Experience, vol. 24, 2012, pp. 445–462.

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

[21] 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.