Implementation of Business Processes in Service Oriented Architecture

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 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. The transformations are selected manually and executed by an automatic tool. Each transformation changes the process structure to improve the quality characteristics. The method applies a correct-by-construction approach and defines a set of transformations, which do not change the process behavior. The quality of the process implementation is assessed using a set of metrics.**

Keywords-business process; BPEL language; service oriented architecture; SOA; transformational implementation.

I. INTRODUCTION

A business process is a set of logically related activities performed to achieve a defined business outcome [1]. 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 [2] or the notation of ARIS [3]. 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 of, e.g., Business Process Executable Language [4] or UML activity diagrams [5].

This paper describes a transformational method for implementing 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 transformation process. In this paper we apply a correct-by-construction approach, and define a set of safe transformations, which do not change the process behavior. If each transformation is safe, the resulting program will also be correct, i.e., semantically equivalent to the original reference process.

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. An illustrative case study is provided in Sections IV and VI. Safe transformations are introduced in Section V. 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 [6] and formal modeling by means of Petri nets [7].

An approach for transformational implementation of business processes was developed in [8]. 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 [2]. The reference model of SOA [9,10] and the specification of BPEL [4] are standardized by OASIS. An informal mapping of BPMN to BPEL was defined in [2] and a comprehensive discussion of the translation between BPMN and BPEL can be found in [11,12]. An open-source tool is available for download at [20].

The techniques of building program dependence graph and program slicing, which we adopted for proving safeness of transformations, were developed in [13,14] and applied to BPEL programs in [15].

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 [16]. We do not use this graph, nevertheless, our metrics Length of thread and Response time can be viewed as an approximation of Critical path metric described in [16]. Similarly, our Number of threads metric is similar to Available concurrency defined in [17].

Our work on the implementation of business processes in a service oriented architecture is to the best of our knowledge, original. An early version of our approach was published in [18]. A definition of safeness, an extended set of transformations, the proofs of transformation safeness, a revised algorithm for building program dependence graph and performance metrics 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 Business Process Modeling Notation. An example BPMN process, which shows 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 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 [19], 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.

To capture the influence of a process structure into the process behavior, we use a technique called program slicing [13,14], which allows finding all the instructions in a program, which influence the value of a variable in a specific

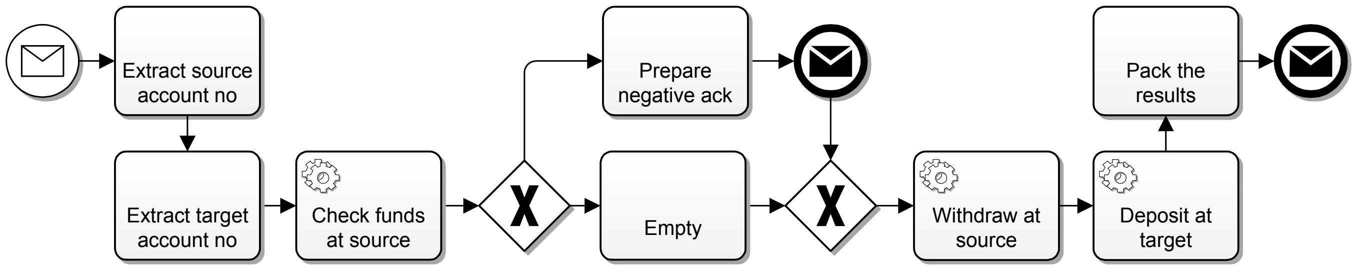

Figure 1. BPMN specification of a business process

point of the program. For example, finding the instructions, which 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), which 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 data 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. Convert "Yes" and "No" edges that output <if> activities into data edges (Fig. 3).
5. Add data 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).

Data edges within a program dependence graph reflect the dataflow dependencies between activities, which determine values of the program variables. Data edges added in step 5 reflect the semantics of the process as a service, which starts after receiving an invocation message. The flow of control within a BPEL program complies with data 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, which preserves the process behavior will be 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 data 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 data edges in a program dependence graph reflects the data flow 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 data 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. CASE STUDY

Consider a process of transferring funds between two different bank accounts, shown in Fig. 1, implemented by a BPEL 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)

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. A skeleton of the simplified BPEL program of this process is shown in Fig. 2.

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 data 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 data 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 data 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.

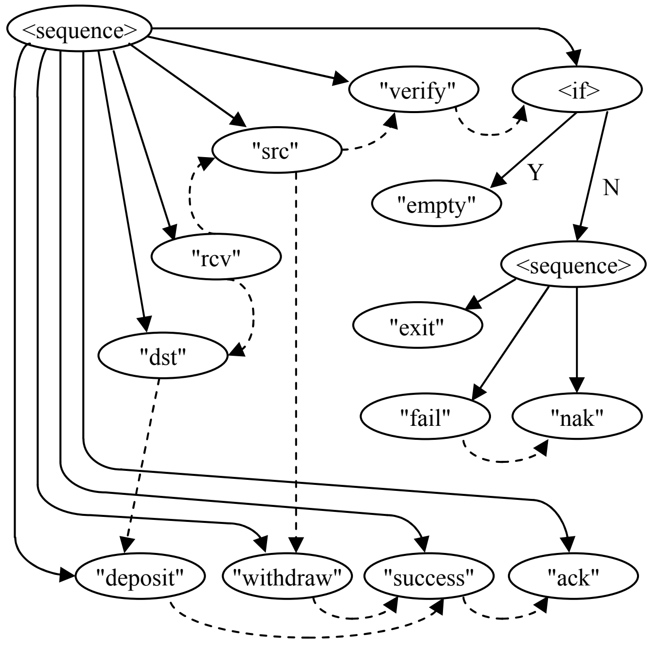

Figure 3. Program dependence graph of the bank transfer process

V. 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 $i+1, \dots j$ 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 data flow from A to those activities existed. However, if no data paths from A to the activities placed on positions $i+1, \dots j$ 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 data 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 data 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.

***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 data 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.

```
<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)

Proof: The transformation has no influence on activities executed prior to <invoke> activity. Data edges from these activities to <invoke> remain unchanged. The transformation has no influence on activities executed after <invoke>, as well. Data 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 data 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, 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).

```
<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

VI. CASE STUDY (CONTINUED)

Consider BPEL program of a bank transfer process described in Section IV. The analysis of the program dependence graph in Fig. 3 reveals that no data flow 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 data flow 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.

```
<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

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 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.

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.

```
<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

VII. Quality Metrics

Many metrics to measure various characteristics of software have been proposed in literature [16,17]. 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 Fig. 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, size of the process structure of the program shown in Fig. 2 is 15, size of the process structure of the program in Fig. 6 is 18, and size of the process structure of the program 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

TABLE I. 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

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., 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 Fig. 2, 6 and 7 are shown in Table I. Program dependence graph and calculation of the metric for the program in Fig. 6 is shown in Fig. 8 (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),

TABLE II. 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 III. 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 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 Fig. 2, 6 and 7 are shown in Table II.

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

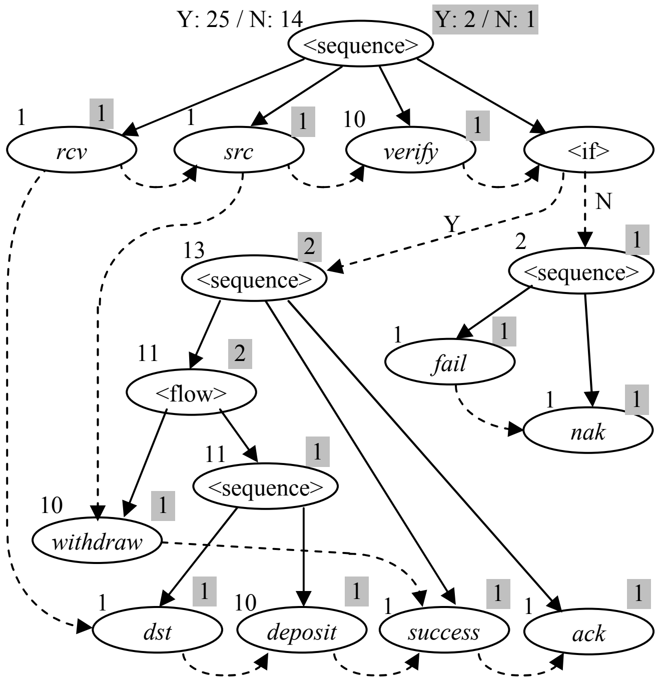

Figure 8. Program dependence graph of the program in Fig. 6 and calculation of metrics: Number of threads (grey numbers right to the nodes) and length of execution (left to the nodes)

and a service invocation. Values of the metric for the processes in Fig. 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. 8 (numbers left to the nodes).

Comparing the values of metrics calculated for the processes considered in the case study in Sections IV and VI, one can note that both transformed processes are faster than the original reference process (smaller value of the response time 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 is appealing for the implementation designer because it can open 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 is 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. The results of this research are not covered in this paper.

Acknowledgments

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] T. H. Davenport and J. E. Short, The New Industrial Engineering: Information Technology and Business Process Redesign, Sloan Management Review, pp. 11-27 (1990)

[2] OMG, Business Process Model and Notation (BPMN), Version 2.0, (2011) http://www.omg.org/spec/BPMN/2.0/PDF/ 20.09.2012

[3] A. W. Scheer, ARIS - Business Process Modeling, Springer, Berlin Heidelberg (2007).

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

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

[6] P. Zave, An Insider's Evaluation of Paisley. IEEE Trans. Software Eng., vol. 17 (3), pp. 212-225 (1991)

[7] K. Sacha, Real-Time Software Specification and Validation with Transnet. Real-Time Systems J., vol. 6, pp. 153-172 (1994)

[8] F. J. Duarte, R. J. Machado, and J. M. Fernandes, BIM: A methodology to transform business processes into software systems, SWQD 2012, LNBIP 94, pp. 39-58 (2012)

[9] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, and R. Metz, Reference Model for Service Oriented Architecture 1.0. Technical report, OASIS (2006)

[10] J. A. Estefan, K. Laskey, F. G. McCabe, and D. Thornton, Reference Architecture for Service Oriented Architecture Version 0.3. Working-draft, OASIS (2008)

[11] S. A. White, Using BPMN to Model a BPEL Process, BPTrends 3, pp. 1-18 (2005) www.bptrends.com

[12] J. Recker and J. Mendling, On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. In: T. Latour, M. Petit (Eds.): Proc. 18th International Conference on Advanced Information Systems Engineering, pp. 521-532 (2006)

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

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

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

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

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

[18] A. Ratkowski and K. Sacha, 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 (2011)

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

[20] Bpmn2bpel Project Home, A tool for translating BPMN models into BPEL processes, http://code.google.com/p/bpmn2bpel/, 22.10.2012