Keywords:
service oriented architecture, SOA,
business processes, BPEL language

Andrzej RATKOWSKI*, Krzysztof SACHA*

ROZDZIAŁ XX
Business Process Design in Service Oriented Architecture

This chapter describes a method for transformational design and implementation of business processes in Service Oriented Architecture (SOA). The starting point of the method is an initial process specification expressed in Business Process Execution Language (BPEL). The design phase consists of a series of transformations, which change the internal process structure without changing the observable process behavior. The goal of each transformation is to improve the quality of the initial BPEL process, defined by a set of metrics, and to benefit from the parallel structure of services and improve the efficiency of the process execution. The result is a transformed BPEL process, which can be executed on a target SOA environment using a BPEL engine.

1. INTRODUCTION

A business process is a set of partially ordered activities, which produce a specific product or service that adds value for a customer. 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. UML activity diagram [1], Business Process Management Notation [2] or Business Process Executable Language [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 such a level of quality may require restructuring of the initial process according to a series of technical decisions, which have to be made by technical people.

This chapter describes a transformational method for designing business process implementation in Service Oriented Architecture (SOA) [4]. The starting point of the method is an initial process specification, called a reference process, defined by busi-

* Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warszawa, Poland.

ness people as a program in Business Process Execution Language (BPEL). The method iterates through a series of steps, each of which makes a small transformation of the process structure. The transformations are selected manually by human designers (technical people) and performed automatically, by a software tool. Each transformation improves the quality of the process implementation, e.g. by benefiting from the parallel structure of services, but preserves the behavior of the reference process. 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.

Basic elements of the method are described in the subsequent sections of this chapter. Process behavior and the meaning of behavior preservation are defined in Section 2. Sample transformations are described in Section 3. Quality metrics are introduced in Section 4. Conclusions and plans for further research are given in Section 5.

2. THE BEHAVIOR OF A BUSINESS PROCESS

A reference process defines a correct flow of computation that is acceptable from the application viewpoint. In the transformation phase, the structure and performance characteristics of the reference process are changed. However, this must not change the behavior of the process. The most important problem is how to define this behavior and prove that it has not been changed. The methods relying on a comparison of states generated during the execution, are inappropriate in a SOA environment.

The behavior of a reference process as well as the behavior of a transformed process may not be deterministic, due to the internal concurrence of services invoked during the execution. On the other hand, the developed software must be deterministic, in that it must not work properly or improperly at random. These two statements are not contradictory, but imply that the non-determinism must not touch these aspects of the software behavior, which are essential from the application viewpoint. We assume that the observable behavior of a process in a SOA environment consists of the values of all the variables that are visible to the outside world, i.e. variables that are passed as arguments when external services are being invoked and variables that are returned at the end of the process execution. This is sufficient, because the services are stateless [4] and return the same results if invoked with the same values of the arguments.

Therefore, we assume that a transformation does not change the process behavior if it does not change the values of the observable process variables. Such a definition neglects timing aspects of the execution. This omission is justified because there are many delays in a network of a SOA system and the correctness of a software must not relay on a specific order of activities, unless they are explicitly synchronized.

To capture the behavior of a process, we use a technique called program slicing [5,6], which allows finding all the commands in a program that influence the value of

a variable in a specific point of the program. For example, the value of a variable that is used as an argument by a service invocation command or by the final reply command of the process.

The conceptual tool for the analysis is Program Dependence Graph (PDG) [7,8], which nodes are commands of a BPEL program and edges are control and data dependencies between these commands. An algorithm for constructing PDG of a BPEL program consists of the following steps (Fig. 1):

1. Define nodes of the graph, which are commands at all layers of nesting.
2. Define control edges (solid lines in Fig. 1b), which follow the nested structure of the program, e.g. an edge from <*sequence*> to <*receive*> shows that <*receive*> command is nested within the <*sequence*> element.
3. Define data edges (dashed lines in Fig 1b), which show data dependencies between commands, e.g. an edge from <*receive*> to <*invoke_1*> shows that an output value of <*receive*> is used as an input argument of <*invoke_1*>. The edges at higher levels of nesting, e.g. from <*receive*> to <*sequence*>, are derived from the existence of edges between the leaf command nodes.
4. Add data edges from <*receive*>, which is the first command in the outer <*sequence*> of the program, to each command of this <*sequence*> element. Add data edges to <*reply*>, which is the last command in the outer <*sequence*> of the program, from each command of this <*sequence*> element. These edges reflect the semantics of receive-reply construct and are not shown in Fig 1b.

(a)

```
<sequence>
    <receive>
    <sequence>
        <invoke_1>
        <invoke_2>
    </sequence>
    <sequence>
        <assign_1>
        <assign_2>
    </sequence>
    <reply>
</sequence>
```

(b)

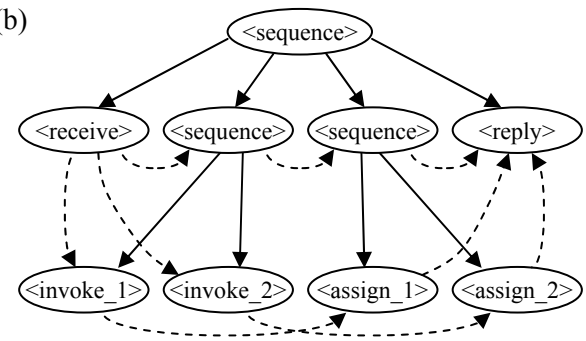

Fig. 1. A nested BPEL process: Process body (a) dependence graph (b)

A path composed of data edges in a program dependence graph expresses the data flow relationships between the commands and implies that the result of the command at the end of a path depend only on the results of the earlier commands of this path. Hence, a transformed program, which preserves the data paths of the reference program dependence graph and guarantees that the order of execution complies with the data paths, preserves the observable behavior of the reference program.

3. TRANSFORMATIONAL DESIGN

BPEL process consists of structured elements, such as <*sequence*>, <*flow*>, etc. that can be nested in each other. The behavior of a process results from the behavior and the order of execution of particular structured elements. A transformation applies to an element and consists in replacing one element, e.g. <*sequence*>, by another element, e.g. <*flow*>, composed of the same commands. If the behavior of both elements, i.e. the element before a transformation and the replacing element, and their position in a process are the same, then the behavior of the process stands also unchanged.

It is obvious from the above definition that a superposition of two or more transformations, which do not change the behavior of the transformed elements, preserves the behavior of the process.

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

Transformation 1: Permutation

Consider a BPEL <*sequence*> element, which contains n arbitrary commands C_1 through C_n (Fig. 2a) that are executed in a strictly sequential order. The particular commands can be simple actions, e.g. <*assign*> or <*invoke*>, as well as structured elements, e.g. <*sequence*> or <*flow*>. Transformation 1 changes the order of commands by exchanging two commands C_i and C_j (Fig. 2b).

(a)
```
<sequence>
    <C1> </C1>
    ......
    <Ci> </Ci>
    ......
    <Cj> </Cj>
    ......
    <Cn> </Cn>
</sequence>
```

(b)
```
<sequence>
    <C1> </C1>
    ......
    <Cj> </Cj>
    ......
    <Ci> </Ci>
    ......
    <Cn> </Cn>
</sequence>
```

Fig. 2. Permutation of commands: Before (a) and after (b) the transformation

***Theorem* 1**: Exchanging commands C_i and C_j does not change the behavior of the <*sequence*> element, if for each command C_k, such that $i < k \le j$, neither a path from C_i to C_k nor a path from C_k to C_j exists in the process dependence graph:

$$(\forall k: i < k \le j) \sim[\, C_i \rightarrow C_k \vee C_k \rightarrow C_j \,]$$

Proof: The dependence graph of the element before the transformation (Fig. 2a) is shown in Fig. 3. Commands $C_1 ... C_n$ are executed sequentially from left to right. The order of commands C*i* and C*j* has no influence on the result of any command C_l, $l < i$, which is executed before either C_i or C_j is started, as well as on any command C_m, $j < m$, which is executed later. However, permutation of C_i and C_j can influence the commands that are between. If a path from C_i to C_k exists in the graph, then permutation of C_i and C_j moves C_i after C_k, which depends on the result of C_i. Similarly, if a path from C_k to C_j exists in the graph, then permutation of C_i and C_j moves C_k after C_j, which depends on the result of C_k. If no paths between C_i, C_k and C_j exist in the graph, then the permutation can not change the result of any of these commands.

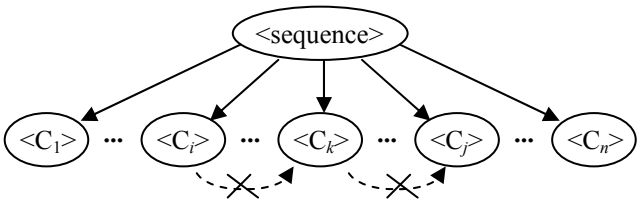

Fig. 3. Dependence graph of a sequence (Fig. 1a)

Transformation 2: Parallelization

Consider a BPEL <*sequence*> element, which contains *n* arbitrary commands C_1 through C_n (Fig. 4a). The particular commands can be simple actions as well as structured elements. Transformation 2 parallelizes the execution of commands by replacing BPEL <*sequence*> element by <*flow*> element, composed of the same commands (Fig. 4b), which – according to the semantics of <flow> – are executed in parallel.

(a)
```
<sequence>
    <C1> </C1>
    ......
    <Cn> </Cn>
</sequence>
```

(b)
```
<flow>
    <C1> </C1>
    ......
    <C2> </Cn>
</flow>
```

Fig. 4. Parallelization of commands: Before (a) and after (b) the transformation

***Theorem* 2**: Parallelization of commands C_1 through C_n does not change the behavior of the transformed element, if for each pair of commands C_i,C_j, $i,j \le n$, neither a path from C_i to C_j nor a path from C_j to C_i exists in the process dependence graph:

$$(\forall i, j \le n) \sim [\, C_i \rightarrow C_j \vee C_j \rightarrow C_i \,]$$

Proof: The lack of paths between the commands means that no dependencies between these commands exist and the result of any command does not depend on the

order and position of other commands from C_1 through C_n. Hence, all the commands can be executed in any order, also in parallel.

Transformation 3: Serialization

Consider a *<flow>* element, which contains *n* arbitrary commands C_1 through C_n (Fig. 5a) that are executed in parallel. The particular commands can be simple actions as well as structured elements. Transformation 3 serializes the execution of commands by replacing BPEL *<flow>* element by *<sequence>* element, composed of the same commands (Fig. 5), which are now executed in parallel.

(a)
```
<flow>
    <C1> </C1>
    ......
    <Cn> </Cn>
</flow>
```

(b)
```
<sequence>
    <C1> </C1>
    ......
    <C2> </Cn>
</sequence>
```

Fig. 5. Serialization of commands: Before (a) and after (b) the transformation

***Theorem* 3**: Serialization of commands C_1 through C_n does not change the behavior of the transformed element.

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

Transformations 1 through 3 can be composed in any order, resulting in a complex transformation of the process structure. Transformations 4 and 5 play an auxiliary role and facilitate such a superposition. The proof of safeness of these two transformations is obvious, because neither of them changes the order of execution of commands.

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

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

Fig. 6. Partitioning of a set of commands: Sequential (a) and parallel (b)

Transformation 4: Sequential partitioning

Transformation 4 divides BPEL <*sequence*> element (Fig. 4a) into a nested structure of <*sequence*> elements (Fig. 6a).

***Theorem* 4**: Partitioning does not change the behavior of the transformed element.

Transformation 5: Parallel partitioning

Transformation 5 divides BPEL <*flow*> element (Fig. 4b) into a nested structure of <*flow*> elements (Fig. 6b).

***Theorem* 5**: Partitioning does not change the behavior of the transformed element.

To illustrate superposition of transformations, consider a process, which invokes two external services and makes two pieces of computation (Fig. 7a), with data dependencies between commands described by a process dependency graph (Fig. 7b).

(a)

```
<sequence>
    <receive>
    <invoke_1>
    <invoke_2>
    <assign_1>
    <assign_2>
    <reply>
</sequence>
```

(b)

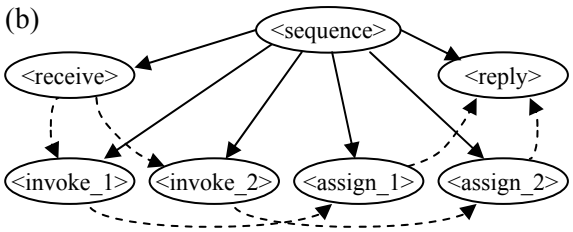

Fig. 7. Reference process: Process body (a) dependence graph (b)

To enable parallel execution of commands within the process, one can use transformation 4 first (Fig. 8a) and then transformation 2 (Fig 8b). The advantage of the transformed process is parallel execution of external services and parallel execution of the computations, which can result in faster response and increased efficiency.

(a)

```
<sequence>
    <receive>
    <sequence>
        <invoke_1>
        <invoke_2>
    </sequence>
    <sequence>
        <assign_1>
        <assign_2>
    </sequence>
    <reply>
</sequence>
```

(b)

```
<sequence>
    <receive>
    <flow>
        <invoke_1>
        <invoke_2>
    </flow>
    <flow>
        <assign_1>
        <assign_2>
    </flow>
    <reply>
</sequence>
```

Fig. 8. Transformed process: After transformation 4 (a), and then after transformation 2 (b)

One another possibility of the process transformation is to reorder the commands (transformation 1), then partition the process using transformation 4 (Fig. 9a), and finally parallelize the sub-sequences of commands using transformation 2 (Fig. 9b). The advantages of the transformed process are similar as in the previous case. However, intuition suggests that the structure in Fig. 9b is better than the one in Fig. 8b.

(a)
```
<sequence>
  <receive>
  <sequence>
    <sequence>
      <invoke_1>
      <assign_1>
    </sequence>
    <sequence>
      <invoke_2>
      <assign_2>
    </sequence>
  </sequence>
  <reply>
</sequence>
```

(b)
```
<sequence>
  <receive>
  <flow>
    <sequence>
      <invoke_1>
      <assign_1>
    </sequence>
    <sequence>
      <invoke_2>
      <assign_2>
    </sequence>
  </flow>
  <reply>
</sequence>
```

Fig. 9. Transformed process: After transformations 1 and 4 (a), and then after transformation 2 (b)

In order to verify this impression, the reference process and the candidate processes obtained as result of transformations 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.

4. QUALITY METRICS

There are many metrics to measure various characteristics of software [9,10]. In this research we use simple metrics that characterise 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.

Size of a process is measured by the number of commands in the BPEL program. More precisely, the value of this metric equals the number of leaf nodes in the program dependence graph of this process. For example, the size of a process in Fig. 7a, 8b and 9b is the same and equals 6.

Complexity of a process is measured by a relation of the number of nodes in PDG to the size of the process. For example, the complexity of a process in Fig. 7a equals 1,17 (7/6), while the complexity of processes in Fig 8b is 1,5 and in Fig. 9b is 1,67.

Number of threads is counted by assigning weights to the nodes of the program dependence graph of a BPEL process, starting from the leaves up to the root, according to the following rules:

- the weight of a simple BPEL command is 1,
- the weight of a <*flow*> element is the sum of weights of the descending nodes (i.e. the directly lower nodes in the hierarchy of nesting),
- the weight of a <*sequence*> element is the maximum of weights of the descending nodes (i.e. the directly lower nodes in the hierarchy of nesting).

The value of this metric equals the weight assigned to the top <*sequence*> node of the graph. For example, the maximum number of threads in a process in Fig. 7a is 1, while the maximum number of threads in processes in Fig. 8b and Fig. 9b equals 2.

Length of parallel threads is counted by assigning weights to the nodes of the program dependence graph of a BPEL process, starting from the leaves up to the root, according to the following rules:

- the weight of a simple BPEL command is 1,
- the weight of a <*sequence*> element is the sum of weights of the descending nodes (i.e. the directly lower nodes in the hierarchy of nesting),
- the weight of a <*flow*> element is the maximum of weights of the descending nodes (i.e. the directly lower nodes in the hierarchy of nesting).

The value of the metric equals the maximum weight assigned to a <*flow*> node or is 0, if such a node does not exist in the graph. For example, the maximum length of a thread in a process in Fig. 7a is 0, while the maximum length of a thread in processes in Fig. 8b is 1 and Fig. 9b is 2.

The thread metrics can be used to predict the efficiency of the process execution. One can expect that a higher number of threads will result in faster response time of the service, due to the internal parallelism of execution. Similarly, a higher length of parallel threads can result in faster response of the process.

To justify the lates statement, denote the execution time of command <*cmd*> by *cmd*. The execution time of the process in Fig. 8b is now:

$$t_1 = receive + \max[invoke\_1, invoke\_2] + \max[assign\_1, assign\_2] + reply$$

while the execution time of the process in Fig. 8b equals:

$$t_2 = receive + \max[invoke\_1 + assign\_1, invoke\_2 + assign\_2] + reply$$

It is easy to see that $t_2 \leq t_1$.

5. CONCLUSIONS

The transformational method for designing business process implementation in SOA, described in this chapter, promotes separation of concerns and allows making

business decisions by business people and technical decisions by technical people. The former relates to the definition of a reference process, which reflects the flow of business process used in an organization and does not take into account the technical characteristics of the execution environment. The latter relates to the design phase, in which the reference process is transformed in order to improve efficiency and benefit from the parallel structure of services in a SOA environment. Other quality features, such as modifiability or reliability, can also be considered.

Transformations exemplified in Section 3 are correct in that they do not change the observable behavior of the reference process. This is a very restrictive assumption, which not always is justified in reality. Therefore, a real challenge is to find a method that would allow small changes to the reference process. The supporting tool would warn the designer that such a change have been made by a transformation and show precisely the consequences. The decision whether to allow or to deny such a change would be made by the human designer.

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

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

[3] ANDREWS T. et al, *Business Process Execution Language for Web Services, Version 1.1*, 2003, http://www.ibm.com/developerworks/library/specification/ws-bpel/.

[4] ERL T., *Service-oriented Architecture: Concepts, Technology, and Design*, Prentice Hall, Englewood Cliffs, 2005.

[5] Weiser M.., *Program slicing*, IEEE Trans. Software Eng., Vol. 10, No. 4, 1984, 352–357.

[6] BINKLEY D., GALLAGHER K.B., *Program slicing*, Advances in Computers, Vol. 43, 1996, 1–50.

[7] OTTENSTEIN K.J., OTTENSTEIN L.M., *The program dependence graph in a software development environment*, Proc. ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, ACM, 1984, 177–184.

[8] MAO C., *Slicing web service-based software*, IEEE International Conference on Service-Oriented Computing and Applications, IEEE, 2009, 1 –8.

[9] McCABE, T.J., *A Complexity Measure*, IEEE Trans. Software Eng., Vol. 2, No. 4, 1976, 308-320.

[10] PARIZI R. M., AZIM A., GHANI A., *An ensemble of complexity metrics for BPEL web processes*, Proc. ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, IEEE Computer Society, 2008, 753–758.