J Grid Computing
DOI 10.1007/s10723-013-9278-z

A Transformation-Based Approach to Business Process Management in the Cloud

Evert Ferdinand Duipmans · Luís Ferreira Pires · Luiz Olavo Bonino da Silva Santos

Received: 15 November 2012 / Accepted: 17 September 2013

Abstract Business Process Management (BPM) has gained a lot of popularity in the last two decades, since it allows organizations to manage and optimize their business processes. However, purchasing a BPM system can be an expensive investment for a company, since not only the software itself needs to be purchased, but also hardware is required on which the process engine should run, and personnel need to be hired or allocated for setting up and maintaining the hardware and the software. Cloud computing gives its users the opportunity of using computing resources in a pay-per-use manner, and perceiving these resources as unlimited. Therefore, the application of cloud computing technologies to BPM can be extremely beneficial specially for small and middle-size companies. Nevertheless, the fear of losing or exposing sensitive data by placing these data in the cloud is one of the biggest obstacles to the deployment of cloud-based solutions in organizations nowadays. In this paper we introduce a transformation-based approach that allows companies to control the parts of their business processes that should be allocated to their own premises and to the cloud, to avoid unwanted exposure of confidential data and to profit from the high performance of cloud environments. In our approach, the user annotates activities and data that should be placed in the cloud or on-premise, and an automated transformation generates the process fragments for cloud and on-premise deployment. The paper discusses the challenges of developing the transformation and presents a case study that demonstrates the applicability of the approach.

Keywords Business processes · Cloud computing · BPM in the cloud · Process decomposition

E. F. Duipmans · L. Ferreira Pires (✉)
Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, P.O. Box 217, 7500 AE, Enschede, The Netherlands
e-mail: l.ferreirapires@utwente.nl

E. F. Duipmans
e-mail: e.f.duipmans@student.utwente.nl

L. O. Bonino da Silva Santos
BiZZdesign BV, Capitool 15, 7521 PL, Enschede, The Netherlands
e-mail: l.bonino@bizzdesign.nl

1 Introduction

Business Process Management (BPM) [35] has gained a lot of popularity in the last two decades, since it allows organizations to manage and opti-

Published online: 18 October 2013

mize their business processes. A business process consists of activities, which are performed by either humans or information systems. In a Business Process Management System (BPMS), a process engine is responsible for coordinating and monitoring running instances of business processes. The introduction of the Service-Oriented Architecture (SOA) paradigm [27] has led to an increased use of BPM, especially since the SOA paradigm provides standardized interfaces for defining services and communication between services. Consequently, executable process languages such as WS-BPEL [24], have been introduced for describing executable business processes that integrate existing services, stressing the link between BPM and SOA.

Purchasing a BPMS can be an expensive investment for a company. Not only the software itself needs to be purchased, but also hardware is required on which the process engine should run and personnel need to be hired or allocated for setting up and maintaining the hardware and the software. In addition, scalability can be a concern for companies that use BPM, since a process engine is only able to coordinate a limited number of business process instances simultaneously. As a consequence, organizations might need to purchase additional servers, to ensure that all their customers can be served during peak load situations. Especially when these additional servers are only rarely needed, buying and maintaining the servers might become expensive.

Cloud computing [3] gives its users the opportunity of using computing resources in a pay-per-use manner and perceiving these resources as unlimited. The NIST definition of cloud computing [19] mentions three service models for cloud computing: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). For example, organizations may choose cloud-based BPM systems, in which a BPM system is offered as a service (SaaS) over the Internet [30]. Instead of having to buy hardware and software, the BPM system can be used in a pay-per-use manner. This cloud solution should also offer scalability to the organization, so that in peak load situations, additional resources can be instantiated, and when the rush is over, the additional resources can be terminated. However, the fear of losing or exposing sensitive data in the cloud is one of the biggest obstacles to the deployment of cloud-based solutions in organizations nowadays.

In Duipmans et al. [8] we introduced our initial ideas on an architecture based on Han et al. [12], in which traditional BPM is combined with cloud-based BPM. By splitting up a business process into two groups of interacting processes, one group to run on-premise and another one to run in the cloud, organizations can place their sensitive data and activities that are not computation-intensive within the borders of the organization, whereas non-sensitive data and computation-intensive activities can be placed in the cloud. In our approach, the original (monolithic) business process is transformed according to a user-defined activity distribution list. This gives organizations the possibility of distributing activities and data in a controlled way, depending on performance and sensitivity requirements.

This paper builds on Duipmans et al. [8] and discusses the challenges and design decisions we encountered when designing and implementing the automated transformation support necessary to split business processes according to data and activity distributions. In order to realize this transformation support, we defined an intermediate language and a transformation chain. The intermediate language allows the definition of a core transformation that is independent of the specific business process language used to describe these processes (e.g., WS-BPEL [24], BPMN [25], etc.). The paper motivates and discusses our intermediate language, and shows that this language and the core transformation support the most popular workflow patterns of van der Aalst et al. [2]. This paper shows the applicability of our approach with a case study performed with the tooling that we built to implement our transformations. This paper is based on the work reported in the Master thesis Duipmans [7].

In this paper we refrain from discussing methods to determine which specific activities and data items should be assigned to the cloud, and which should remain on-premise. Here we assume that these assignments have been determined somehow, for example, with the technique described in Han et al. [12], which allows the optimal distribution of activities and data items to be determined

based on the time, monetary and privacy risk costs of the different distribution alternatives.

The reminder of this paper is organized as follows. Section 2 introduces and justifies our extension to Han et al. [12] and explains the purpose of our transformation-based approach. Section 3 introduces and justifies the transformation chain that characterizes our transformation-based approach. Section 4 introduces the intermediate model we defined in order to facilitate the development of our transformations. Section 5 systematically analyzes the decomposition alternatives by considering patterns of the intermediate model and activity allocations. Section 6 discusses the design and implementation of the decomposition transformation. Section 7 presents a case study that illustrates and shows the applicability of our transformation-based approach. Section 8 discusses related work. Section 9 gives our final remarks.

2 Activity and Data Distribution

In most commercial BPM solutions available nowadays, the process engine, the activities and the process data are placed on the same side, either on-premise or in the cloud. In Han et al. [12] the distribution possibilities of BPM in the cloud have been investigated by means of the so-called PAD model, in which the process engine, the activities involved in a process and the data involved in a process are separately distributed. In the PAD model, *P* stands for the process enactment engine, which is responsible for activating and monitoring all the activities, *A* stands for activities that need to be performed in a business process, and *D* stands for the storage of data that is involved in the business process. By distinguishing the process engine, the activities and the data, cloud users gain the flexibility to place activities that are not computation-intensive and sensitive data at the end-user side (on-premise), and all the other activities and non-sensitive data in the cloud.

The PAD model introduced in Han et al. [12] defines four possible distribution patterns: (1) the traditional BPM pattern in which everything is placed at the end-user side; (2) a pattern in which a user runs a process engine on-premise, but computation-intensive activities are placed in the cloud; (3) a pattern in which users do not have a process engine and utilize a cloud-based process engine in a pay-per-use manner, while some activities that are not computation-intensive and sensitive data are placed at the end-user side, and (4) the fully cloud-based BPM pattern in which all elements are placed in the cloud. In the architecture proposed in Han et al. [12], the cloud-side engine deals with data flows only by means of references to data identifiers, instead of the actual data. When an activity needs sensitive data, the transfer of the data to the activity is handled under user surveillance through an encrypted tunnel. Sensitive data is stored at the end-user and non-sensitive data is stored in the cloud.

In Duipmans et al. [8] we proposed an extension of the PAD model of Han et al. [12] with a fifth pattern in which process engines, activities and data are placed in both the cloud and on-premise. The architecture proposed in Han et al. [12] also considers process engines on both the cloud and on-premise sides, but the decomposition of the original process is not addressed there. We proposed the use of two separate process engines in order to minimize the amount of data that has to be exchanged between the cloud and on-premise, so that each process engine regulates both the control flows and data flows of a process.

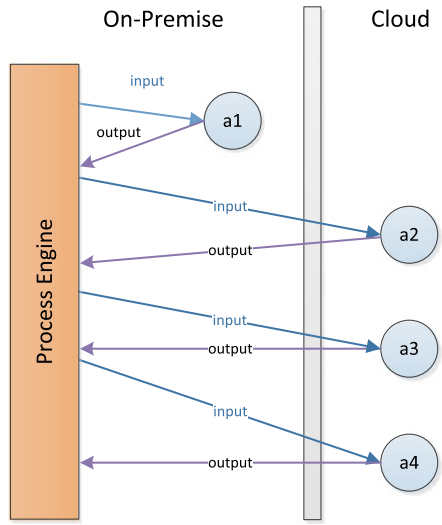


Fig. 1 Single (on-premise) process engines and activity distribution

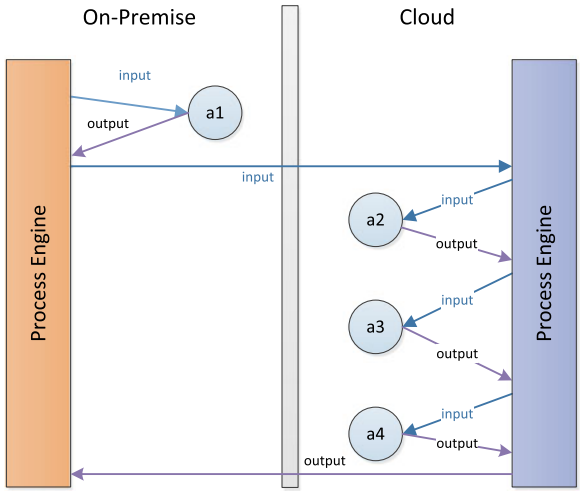


Fig. 2 Two process engines

We justify the potential benefits of distributing a business process with a simple example in which the output of one activity is the input for the following activity. Figure 1 shows a situation in which a process is executed by a single process engine situated on-premise, where some of the activities within the process are placed in the cloud. Since the process is coordinated by the process engine, data are not directly sent from activity to activity, but instead are sent to the process engine first. In the case of cloud activities that are in sequence, using one process engine on-premise may lead to unnecessary data exchange between the process engine and the cloud. Figure 2 shows that by introducing a second process engine in the cloud, we can avoid this problem. Activities in sequence with the same distribution location do not have to send their data from cloud to on-premise, or vice versa, since the coordination can be performed by the process engine in the cloud.

As a consequence of our extension to Han et al. [12], we devised the general goal of developing a transformation framework in which users can automatically decompose a business process into collaborating business processes for distribution on-premise and in the cloud, based upon a list in which activities and data are marked with their desired distribution location. In addition, users should be able to define data restrictions, to ensure that sensitive data stays within the premises of an organization. Figure 3 gives a schematic overview of our transformation approach.

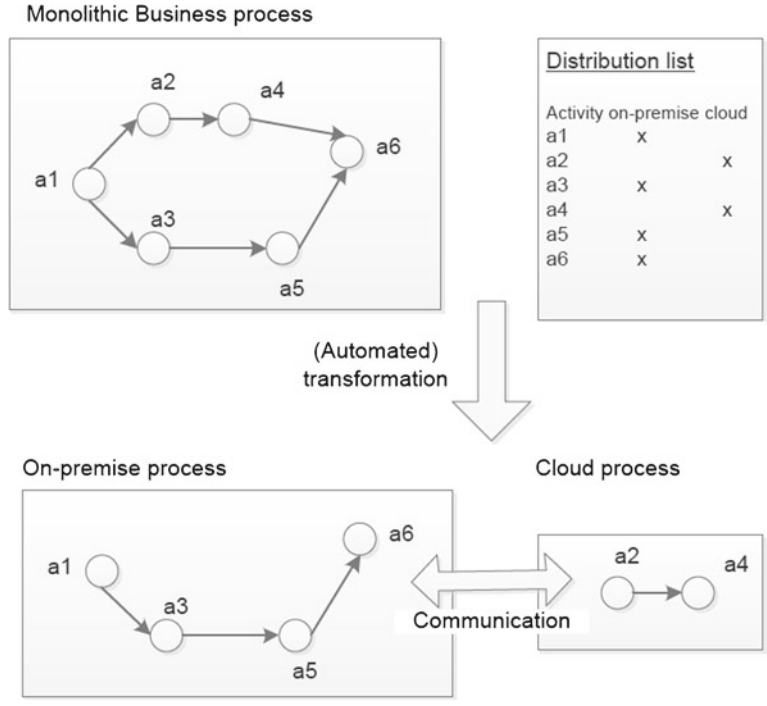


Fig. 3 Process decomposition with activity and data distribution

3 Transformation Approach

In order to realize the goal of decomposing business processes based on the allocation of activities and data on-premise or in the cloud, we investigated techniques that would allow the automated transformation of process models. Initially we intended to define model transformations based on the metamodel elements of the business process language, as discussed in Miller and Mukerji [20], and supported by languages like ATL [14] and QVT [26]. These transformations are defined in terms of transformation rules that define how patterns of the source model are transformed into patterns of the target model. However, we noticed that when applying this transformation approach to our problem, the resulting transformations tended to get quite complex and difficult to manage. Furthermore, these transformations would depend too much on the specific abstract syntax elements of the business process language at hand, with low reusability as a drawback. Therefore we realized that in our case it would be better to define the transformation rules based on the control flow and data flow relations that determine the semantics of processes, instead of language syntax. In addition, the resulting decompositions must comply with the original business process (be functionally correct) and with the intended activity and data distribution.

Instead of building a solution for each specific business process language, we opted then for defining and using an *intermediate model* in which the structure and semantics of business processes are captured. We found two main benefits of using an intermediate model: (1) a business process is defined in a business process language using the syntax of the language, but the decomposition transformation has to be defined in terms of the semantics of the business process language (the control and data flows) that it preserves. This implies that we have to lift the original business process to a model in which the intended semantics of the model is preserved, which can be done with an intermediate model; (2) by using an intermediate model, we can purely focus on the decomposition tasks, without having to consider language-specific problems. As a corresponding drawback, extra transformations are needed for converting a business process to the intermediate model and back.

Figure 4 shows that our approach consists of a transformation chain with three transformations:

1. *Lifting transformation*: transforms a business process defined in some business process language into an instance of the intermediate model. Data analysis is performed during this transformation phase to capture data dependencies between activities in the process. This information is needed for ensuring that no data restrictions are violated during the decomposition transformation.
2. *Core transformation*: transforms an instance of the intermediate model according to an activity distribution list into a new instance of the intermediate model that represents the decomposed processes and the communication between the processes. The activity distribution list defines the distribution locations of

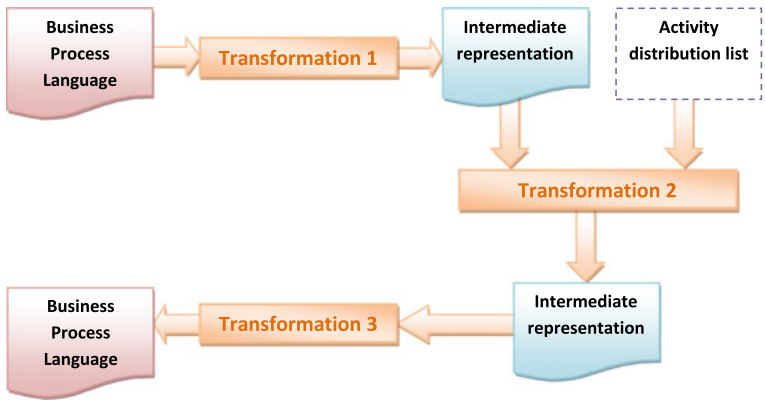


Fig. 4 Transformation chain considered in our approach

each of the activities in the resulting process. Furthermore, data restrictions can be defined in the list. The distribution location of each data element used within the process can also be defined, to ensure that the data element stays within the borders of the defined location.

3. *Grounding transformation*: transforms a decomposed intermediate model back to the (original) business process language. Depending on the language that is used, the transformation creates separate orchestrations for each of the processes, and optionally a choreography in which the cooperation between the processes is described.

4 Intermediate Model

Our challenge in the definition of the intermediate model was to obtain a model that is reasonably simple, but is still able to capture complex business process situations. We used the control-flow workflow patterns defined in van der Aalst et al. [2] for selecting the most common workflow patterns. We decided not to support all of the control-flow workflow patterns at first, since the intermediate model would get too complex. Instead, we identified the patterns that are present in the business process languages WS-BPEL [24], WS-CDL [34], Amber [9] and BPMN 2.0 [25]. We selected the most common patterns from the ones we studied and used them as requirements for the intermediate model. The following patterns of van der Aalst et al. [2] have been considered as requirements for our intermediate model:

WP1: Sequence. The intermediate model should have a mechanism for modeling control flows, in order to be able to express the sequence of execution of activities within a process.

WP2: Parallel split. The intermediate model should support parallel execution of activities. A construct is needed for splitting up a process into two or more branches, which are executed simultaneously.

WP3: Synchronization. The intermediate model should have a mechanism for synchronizing two simultaneously executing branches. A synchronization construct is needed in which multiple branches are joined into one executing branch.

WP4: Conditional choice. The intermediate model should have a construct for executing a branch, based upon an evaluated condition.

WP5: Simple merge. The intermediate model should have a construct for joining multiple alternative branches, from which one is executed.

WP10: Arbitrary cycles. The intermediate model should support a construct for modeling recursive behavior.

The requirements identified so far are all based on control-flows. The following additional requirements should also be supported by our intermediate model:

Data dependencies. Since we might have to deal with sensitive data, it is crucial that the consequences of moving activities around are measurable. By explicitly representing data dependencies between activities, the flow of data through the process can be monitored.

Facilitate decomposition. Since the original process needs to be decomposed into collaborating processes, the intermediate model should have capabilities to define the communication between the resulting processes, i.e., to define how these processes invoke each another. Furthermore, the intermediate model should be relatively easy to manipulate, allowing decomposition transformations to be specified and automated.

4.1 Model Definition

We compared existing process models for their suitability to support the requirements of our intermediate model. The models we compared were mainly taken from similar decentralization approaches. We considered the following models: Program Dependency Graphs (PDGs) [11], Control Flow Graphs (CFGs) [22], Protocol Trees [31] and Petri nets [21].

We analyzed these models and came to the following conclusions:

– PDGs support data dependencies between nodes. Control dependencies, however, are not directly visible in these graphs. This means

that complex behaviors, such as parallel execution of nodes, cannot be properly described by a PDG.
- CFGs can be used for modeling the control flow within a process. The data dependencies between nodes, however, are not visible in these graphs.
- Traditional Petri nets are not able to support all the requirements we set for our intermediate model. For example, data dependencies cannot be modeled in traditional Petri nets, although they can be modeled in Petri net variants such as Colored Petri Nets [13]. The downside of using Petri nets for modeling processes is that many different nodes are needed for representing a process. A transition between two nodes is modeled with places, transitions, tokens and arrows, which would bring an overhead to our intermediate model.
- Protocol Trees are able to capture only block-structured processes. Since we also want to be able to capture graph-based structures, Protocol Trees are not directly suitable to support our requirements.

Since none of the selected models completely satisfies our defined requirements, we decided to define our own intermediate model by combining features of Control Flow Graphs, Program Dependency Graphs and Protocol Trees. The structure of a Control Flow Graph is used for defining control flows between nodes, and the model also contains data dependency edges to capture data flows. The formal definition of our intermediate model (see Section 4.2) has been inspired by the formal definition of Protocol Trees.

We use a graph-based representation for processes, since the base languages we targeted are either block-structured or graph-structured [18]. Another reason for this choice is that graph-based representations are relatively easy to manipulate by means of graph transformations, which has proven to be quite beneficial when we defined our transformations.

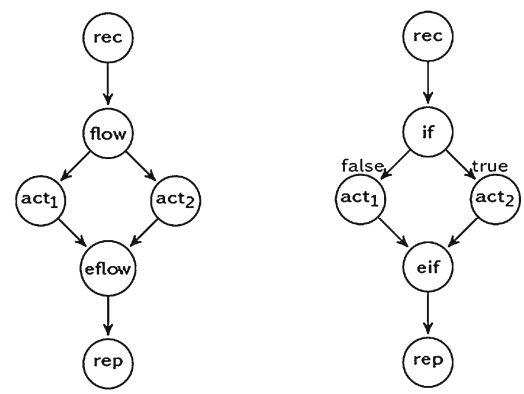
A graph consists of nodes and edges. In our model, a node represents either an activity or a control element. An edge defines a relation between two nodes. In order to be able to capture complex constructs and data dependencies between nodes, we introduce specializations of nodes and edges. For each of the nodes and edges we also define a graphical representation.

Node Types Node types have been defined to represent activities, parallel behaviour, conditional behaviour, loops and communication between processes.

Activities can be modeled by so-called *activity nodes*. Every activity node has at most one incoming control edge, with the exception of one additional control edge coming from a loop node to represent recursive behavior (see Fig. 6b), and at most one outgoing control edge.

Figure 5a shows an example of parallel behavior modeled in the intermediate model. A process with parallel behavior can be modeled using so-called *flow* and *end-flow nodes*. A flow node splits an execution branch into multiple branches, which are executed simultaneously. A flow node has at least two outgoing control edges. Multiple parallel branches can be joined into one execution branch by using the end-flow node. The end-flow node has two or more incoming control edges and at most one outgoing control edge.

Figure 5b shows an example of conditional behavior modeled in the intermediate model. Branch selection based upon an evaluated condition can be modeled by using so-called *conditional nodes*. The conditional node (if-node) has two outgoing control edges. One edge is labeled with "true" and is used when the evaluated



(a) Parallel behaviour (b) Conditional behaviour

Fig. 5 Modeling parallel and conditional behaviors

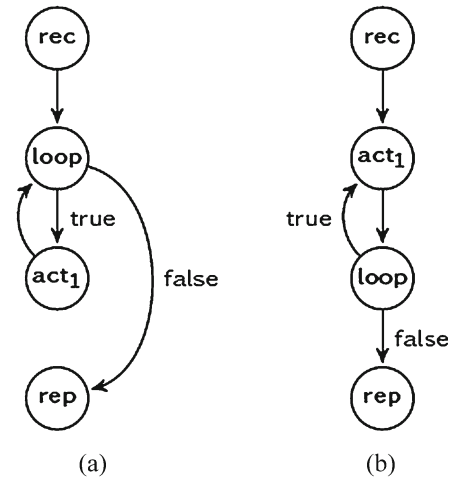


Fig. 6 Loop node **a** before and **b** after a loop branch

condition yields true. The other edge is labeled with "false" and is used otherwise. After the condition in the if-node is evaluated, only one of the outgoing branches can be taken. Conditional branches can be joined by using an end-conditional node (eif-node). This node converts multiple incoming branches into one outgoing branch.

We defined one single node type for modeling repetitive behavior in the intermediate model, the so-called *loop node*. Figure 6 shows two usages of loop nodes. A loop node evaluates a loop condition and according to the result of the evaluation, the loop branch is either executed or denied. The loop-node is comparable to the if-node, since it also evaluates a condition and has outgoing "true" and "false" edges. The outgoing branches, however, are never joined. Instead, one of the branches ends with an outgoing edge back to the loop-node. This branch is called the loop-branch. The other branch points to the behavior which should be executed as soon as the loop-condition does not hold anymore.

The loop node can be placed in the beginning or at the end of the loop branch. The first situation, shown in Fig. 6a, results in zero or more executions of the loop-branch, since the loop condition needs to be evaluated before the loop branch is executed. In the second situation, shown in Fig. 6b, the loop branch is executed at least once, since the loop condition is evaluated after execution of the loop branch.

Communication nodes model the communication between two processes. The intermediate model supports four possible communication nodes: invoke-request, invoke-response, receive and reply. These nodes can be used to model synchronous and asynchronous communication. Figure 7 shows both situations.

The invoke-request-node (ireq) is used for invoking a process. The node has one outgoing communication edge, which points to a receive-node (rec), located in the process that is invoked. The invoke-request-node does not wait until the execution of the invoked process is finished, instead it proceeds to the successor node.

The invoke-response-node (ires) is used as a synchronization node for communication with other processes. This node has one incoming communication edge, which originates from a reply-node (rep) located in another process. The invoke-response-node waits for the response from the other process, before continuing its execution.

In the case of synchronous communication, as shown in Fig. 7a, a process (P1) uses an invoke-request-node to invoke another process (P2). The invoke-request-node follows its outgoing control edge, which is connected to an invoke-response-node. This node, in turn, waits until process P2 is finished, before continuing with process P1. In the case of asynchronous communication, as shown in Fig. 7b, a process (P1) invokes another process (P2) by using an invoke-request-node. After calling P2, P1 continues its execution.

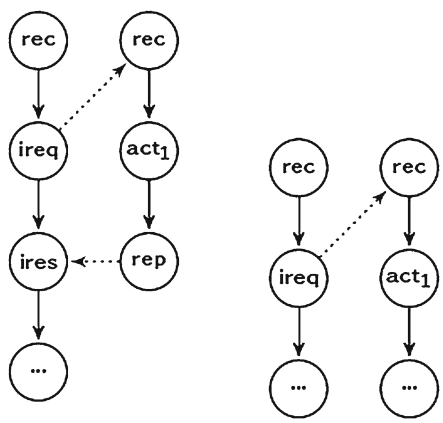


Fig. 7 Communication with the intermediate model

Edge Types Our intermediate model distinguishes between *control edges*, *data edges* and *communication edges*.

Control flow is modeled in the intermediate model by control flow edges, which are represented by solid arrows in our graphical notation. The node from which the edge originates triggers the edge as soon as the execution of the node's action has been finished. The node in which the edge terminates waits for a trigger, caused by an incoming edge, before it starts executing the action of the node. A control edge can be labeled with "true" or "false", in case the control edge originates from a conditional-node. When the evaluated condition matches the label of the edge, the edge is triggered by the conditional node.

In business process languages such as WS-BPEL [24], data flow between activities is defined implicitly. Instead of sending data from activity to activity, activities can access variables directly, provided that the activity has access to the scope in which the variable is defined. By introducing data edges in our intermediate model, we are able to investigate the consequences of moving activities from one process to another for data exchange. This information is needed to verify if data constraints are violated during the transformation. A data link is represented by a dashed arrow between two nodes in our graphical notation. A data edge from node $N1$ to node $N2$ implies that data defined in node $N1$ is used in node $N2$. Each data edge is provided with a label, in which the name of the shared data item is defined.

Communication edges are defined between communication nodes, and are represented as dotted arrows. A communication edge sends control and data to a different process. Communication edges are labeled with the names of the data items that are sent over the edge.

4.2 Formal Definition

Formally, our intermediate model I is a tuple $(A, C, S, ctype, stype, E, L, nlabel, elabel)$, where:

- A is a set of activity nodes.
- C is a set of communication nodes.
- S is a set of structural nodes (flow nodes, end-flow nodes, if nodes, end-if nodes and loop nodes).
- *ctype*: is a function that assigns the communicator type to a communication node.
- *stype*: is a function that assigns a control node type to a control node.
- E is the set of all edges in the graph. Let $E = E_{\text{ctrl}} \cup E_{\text{data}} \cup E_{\text{com}}$. An edge is defined by a tuple $(n_1, etype, n_2)$ where *etype* denotes the type of the edge and $n_1, n_2 \in A \cup C \cup S$, and E_{ctrl}, E_{data} and E_{com} are the sets of control flow edges, data edges and communication edges, respectively.
- L is the set of text labels that can be assigned to nodes and edges.
- $nlabel : N \rightarrow L$, where $N = A \cup C \cup S$ is a function that assigns a textual label to a node.
- $elabel : E \rightarrow L$ is a function that assigns a textual label to an edge.

Below we illustrate our intermediate model with an example that shows how a loop and a flow construct in WS-BPEL are mapped to the intermediate model. Listing 1 and Fig. 8 show the WS-BPEL example specification and the graphical representation of the corresponding intermediate model, respectively.

The while element in the BPEL example is mapped onto a loop construct in which the condition is evaluated before the execution of the loop branch. The loop branch consists of a flow element, which is mapped in the intermediate model onto a parallel construct with a flow and an end-

Listing 1 BPEL loop example

```
<sequence>
  <receive ... variable="A" />
  <while>
    <condition>...</condition>
    <flow>
      <invoke name="act1" inputVariable="A"
              ... />
      <invoke name="act2" inputVariable="A"
              outputVariable="B" />
    </flow>
  </while>
  <reply ... />
</sequence>
```

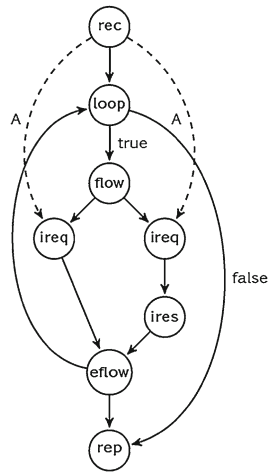


Fig. 8 Graphical representation of the BPEL example in our intermediate model

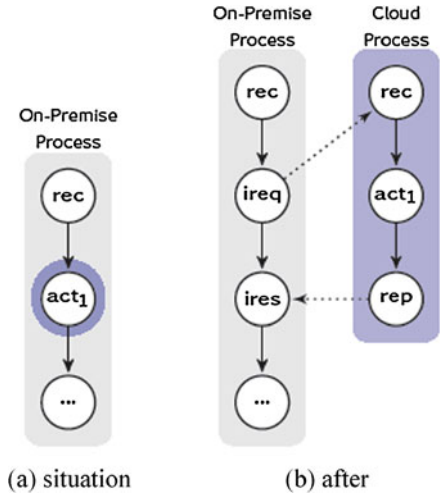


Fig. 9 Single activity moved from on-premise to the cloud

flow node. The invoke activities executed within the parallel construct are mapped onto communication nodes. Invocation "act1" is mapped onto an asynchronous invocation element, since it expects no response, while invocation "act2" is mapped onto two synchronous invocation nodes, since the invocation needs to wait for a response from the invoked service. Data dependencies are introduced between the receive node and the invocation nodes, since the invocation nodes use the variable that was received by the receive element.

5 Decomposition Analysis

In order to define our transformation, we defined transformation rules for each of the constructs defined in the intermediate model, by considering all possible allocations of activities to the cloud and on-premise locations. In this analysis we took into account processes that are hosted on-premise and have activities that should be allocated in the cloud, or vice-versa.

5.1 Single Activity

When a single activity is marked for allocation to the cloud as shown in Fig. 9a, the solution shown in Fig. 9b is suitable. In this solution, the activity is moved to a new cloud process and called by the on-premise process via synchronous invocation nodes. By using synchronous invocation, the execution sequence of the processes can be preserved, since the node following the activity in the original process has to wait until the cloud process is finished.

5.2 Sequential Activities

When multiple sequential activities are marked for allocation to the cloud, the sequential activities can be placed in two separate cloud processes, or the sequential activities can be placed together in one cloud process. Figure 10 shows the allocation of sequential activities to separate processes.

In this case there are two possible solutions, which depend on the distribution of the control links between the activities:

1. *Maintain control links on-premise*: in Fig. 10b, for each marked activity a new cloud process is created. Synchronous invocation nodes are introduced in the on-premise process for invoking the activities in the cloud. In the original process shown in Fig. 10a, a control link is present between the activities. Since both activities are placed in new processes, there is no direct control link any more between the activities. In our first solution, a control link is introduced between the created communication nodes in the on-premise process, to keep the on-premise process together. The drawback of this solution is that there is unnec-

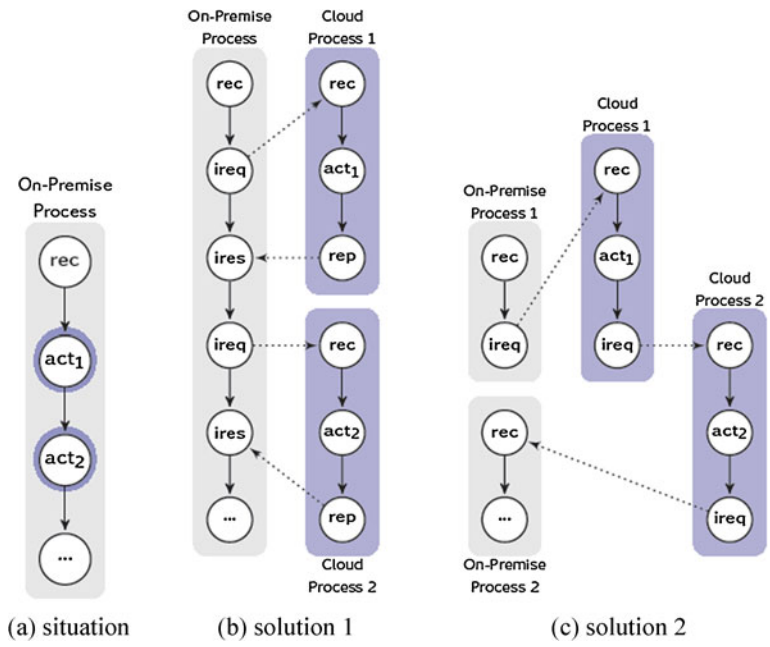


Fig. 10 Alternatives for sequential nodes

essary communication between the cloud and on-premise, since the result of the first cloud process is sent to the second cloud process via the on-premise process, instead of sending it directly.

2. *Move control links to the cloud*: in Fig. 10c, both activities are moved to individual cloud processes. The on-premise process calls the first cloud process. After execution of the activity in the first cloud process, the second cloud process is called directly. The second cloud process eventually gives a call back to the on-premise process. The control link between the two activities in the original process is moved to the cloud and placed between the invoke and receive of the first and second cloud process. As a consequence, the on-premise process is no longer a single process, but is decomposed into two separate processes.

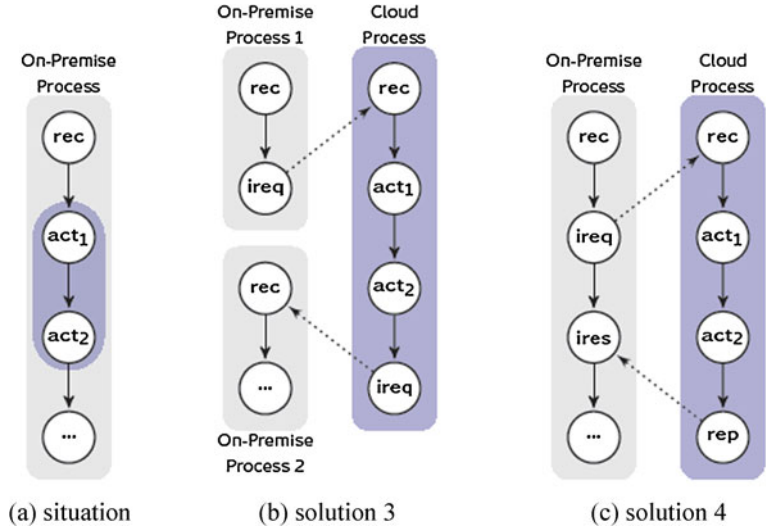


Fig. 11 Sequential activities moved as a block

Figure 11 shows the solutions for the second situation, in which two sequential activities are moved together to one cloud process. The following two solutions are applicable in this situation:

3. *Splitting up on-premise processes*: in this solution the sequential activities are moved to a new cloud process. By moving these activities, a gap arises between the nodes that are placed before the cloud process and the nodes after the cloud process. Figure 11b shows that this solution leads to more additional processes, since every time a sequence of nodes is placed in the cloud, the on-premise process is split up.
4. *Replace by synchronous invocation node*: in this solution, shown in Fig. 11c, the moved part in the on-premise process is replaced by a control edge, preserving in this way the control structure of the on-premise process. Replication of the control link between the processes leads to more complex processes, but the overall structure of the on-premise process is preserved, since the cloud nodes are replaced by invocation nodes. This makes the on-premise process more robust, since the execution of the overall process is coordinated by the on-premise process.

5.3 Composite Constructs

Parallel constructs and conditional constructs can be generalized as composite constructs since their syntax structure is quite similar. Both constructs start with a node that splits the process into several branches. Eventually, the branches join in an end-node, which closes the composite construct. However, from a semantics perspective the behavior of these constructs is completely different. We analyzed all the decomposition possibilities for composite constructs and came up with the following possibilities: (1) the start and end node (e.g., flw and eflw) and all the contents within the composite construct are allocated to the same destination, and are kept together as a whole; (2) the start and end node have the same distribution location, but activities within the composite construct need to be maintained locally, and (3) the start and the end node have different distribution locations.

In Section 5.2 we show that sequential activities of the on-premise process can be either split up into individual processes or kept together. This strategy can also be applied to composite nodes. Therefore, we can keep the on-premise process together, when the start and end node of a composite node have the same distribution location, to reduce the number of solutions. For activities within branches of the composite constructs, we move activities as a block and keep the surrounding process together, to reduce the number of solutions. This allows the possible decomposition rules to be applied recursively on each of the branches of the composite constructs. Each resulting decomposition alternative is discussed in the sequel.

Category 1: Moving the Composite Construct as a Whole Figure 12a shows the situation where the start and end node and all the nodes in the branches of the composite node are marked for allocation to the cloud. Figure 12b shows the solution that is applicable in this situation. The construct is moved as a whole to a new cloud process. In the on-premise process, synchronous invocation nodes are introduced to call the cloud process.

Category 2: Start/End Nodes with the Same Distribution Location Three possible situations exist with composite nodes, where the start and end

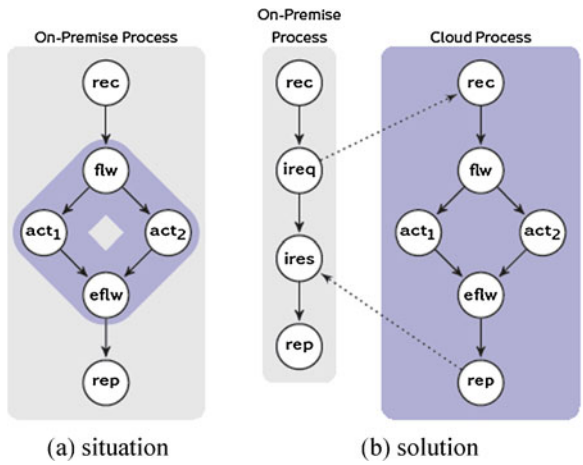


Fig. 12 Moving the whole composite construct

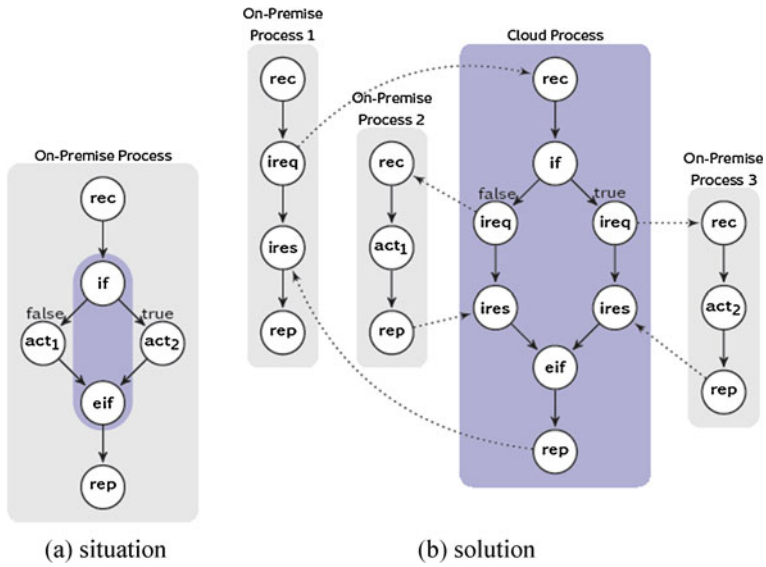


Fig. 13 Composite constructs are marked to move but activities stay on-premise

node have the same distribution location, and contents within the construct have a different distribution location. In the Master thesis Duipmans [7] each of these situations have been analyzed, and solutions have been presented in which the composite construct itself is placed in only one process, either on-premise or in the cloud. The branches within the composite construct are treated as subprocesses, and the rules defined earlier are recursively applied on these subprocesses. Activities within the branches of the composite construct with the same distribution location as the construct itself are placed directly within the construct. Activities with a different distribution location are placed in new processes. Figure 13 shows the situation where the composite constructs are marked for moving and the activities are not. The other alternatives are omitted here due to space limitations.

Category 3: Start/End Node with Different Distribution Location The last category consists of situations in which the start-node and the end-node of the composite construct have different distribution locations. This category has also been thoroughly analyzed in the design of the transformation and reported in Duipmans [7], but is omitted here due to space limitations, and because it has not been considered in our prototype.

5.4 Loops

In the intermediate model we distinghished two categories of loops, namely loops in which the loop condition is evaluated before or after the execution of the loop branch. Below we analyze the possible decomposition solutions for each category.

Condition Evaluation Before Branch Execution There are two situations possible when dealing with loop constructs in which the conditional node is evaluated before the execution of the loop branch:

1. *Move construct as a whole*: we omit the solution for this situation here, since it is comparable to moving a composite construct as a whole, which is explained in Section 5.3. The complete construct can be moved to a new process and is replaced in the original process by synchronous invocation nodes.
2. *Conditional node and nodes within loop branch have different distribution locations*: we can treat the loop branch within the loop construct as a separate process, since it is executed after a conditional node. A loop branch is only connected to the original process through the conditional node. Treating the branch as a separate process gives the

opportunity to apply the other decomposition rules recursively on the branch. Figure 14 shows the situation in which the condition of a loop construct is moved to the cloud, whereas the activities within the loop branch are allocated to an on-premise process.

Condition Evaluation After Branch Execution When dealing with loops in which the condition of the loop is evaluated after execution of the loop branch, we identified two situations:

1. *Move construct as a whole*: this situation is comparable to moving a composite construct as a whole. We omit the discussion of this situation here, since it is similar to the solution presented in Section 5.3.
2. *Conditional node and nodes within loop branch have different distribution locations*: there are two possible solutions for dealing with this situation. In the first solution, the loop branch and loop condition node are moved to a new process and are replaced in the original process by synchronous invocation nodes. In the newly created process (loop process), the loop branch is taken and moved to a separate process and called by the loop process via synchronous invocation nodes. The second solution is to move the loop branch to a separate process and rewrite the loop construct to a loop construct in which the condition is evaluated before the execution of the loop branch. The original process then first calls the loop branch, to execute the branch once before evaluation of the condition. After this invocation, a new invocation is used to call the loop process. In this loop process, the loop condition is evaluated first and depending on the result of the evaluation, the loop branch is executed in which the on-premise process is invoked.

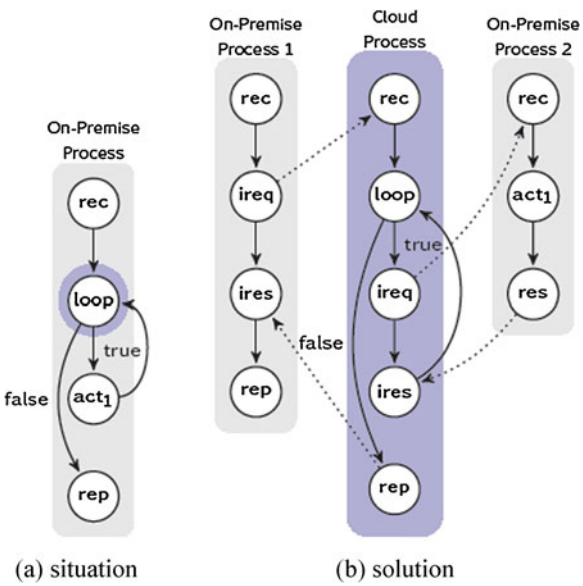


Fig. 14 Conditional node moved to the cloud and loop branch on-premise

6 Core Transformation Design

In order to define and implement our core transformation and keep it manageable, we have selected some situations from the decomposition analysis discussed in Section 5. These choices have simplified our transformation considerably, without imposing strong limitations to their applicability, as we observed in our case studies. We specified and implemented the core transformation in both Java and as a graph transformation in the tool Groove [29].

6.1 Design Decisions

We took the following decisions concerning the decomposition design:

Process completeness. The input process for the transformation is restricted with the following constraints: (1) the input process has at most one start node; (2) the input process has at most one end node, and (3) the start-node of each composite construct should have a corresponding end-node, in which all of the branches of the composite construct are merged. This decision was taken to avoid complex situations with multiple receive nodes at the beginning of a process, or multiple reply nodes at the end of a process. In addition, the restrictions ensure that a process will never start or end with alternative or simultaneously executing branches.

Group sequential activities. Sequential activities with the same distribution location are always placed together in a process and are moved as a block to a new process, instead of being placed in separate processes. This decision was taken to reduce the number of processes that are generated during the decomposition transformation.

Keep process together. When a sequence of activities is moved from one side to another, the surrounding process is kept together, as in solution 4 for the sequential activities shown in Fig. 11c. By keeping the process together, the original process is not split, and only new processes are generated for activities with a different distribution location than the original process. In the original process, these nodes are replaced by communication nodes. Since the structure of the process is maintained, the calling behavior of the process does not change either.

Treat branched activities as separate processes. Each branch within a composite construct is treated as a separate process. Nodes with the same distribution location as the surrounding composite construct stay within the branch of the construct. Nodes with a different distribution location are moved to separate processes. This decision gives us the opportunity to use the decomposition rules recursively on the branches of the composite constructs.

Keep composite constructs start and end together. When dealing with composite constructs, we only allow the situation in which the composite construct is kept together. Different distribution locations for start and end nodes of composite constructs are not allowed. This decision was taken to avoid complex situations with composite constructs. By keeping the start and end node together in the same process, we can treat them as block-structured elements [18] and perform the decomposition operations recursively on the branches of the construct.

Only allow a single incoming edge for activity nodes. Activity nodes can have at most one incoming edge. This implies that loop nodes after loop branches (see Fig. 6b) are not supported. This choice is not really harmful, since behaviors with loop nodes after the loop branches can be rewritten to equivalent behaviors with loop nodes before the loop branches, with the addition of a copy of the loop branch before the loop node.

Treat loop branches as separate processes. Loop branches are treated as separate processes. By treating the loop branch as a separate subprocess we are able to use the decomposition rules recursively on the branch and treat the loop construct as a block-structured element.

6.2 Data Structures

Figure 15 shows an excerpt of the class diagram we used in the implementation of our Java transformations. We briefly explain each class below:

- *Graph* is the main class for defining processes. A graph consists of a list of nodes, a list of edges, and a couple of functions for performing operations on the graph. Function getAllGraphsAndSubGraphs can be used to get a list of graphs, in which the current graph is placed along with all the subgraphs that are available within the graph. Subgraphs are branches within composite constructs, such as loop branches or branches of a parallel/conditional constructs. Classes Graph, Node and Edge all have a hash map of attributes in which additional information about the objects can be stored.
- *Node* is the parent class for all the nodes. Each node has a unique name and distribution location, which indicates where the node should be located. Attribute executionGuaranteed is used during the lifting transformation for optimizing the data dependency analysis.
- *Edge* connects two nodes to each other. Each edge consists of a 'from' attribute, which represents the node from which the edge originates, and a 'to' attribute, which represents the node in which the edge terminates. Each edge has a specific edge type, which is either Control, Data or Communication. In addition, a label can be attached to the edge by using the label attribute.
- *ActivityNode* is used to define activities within the process.
- *CommunicatorNode* is used to communicate with another process. The communicator type

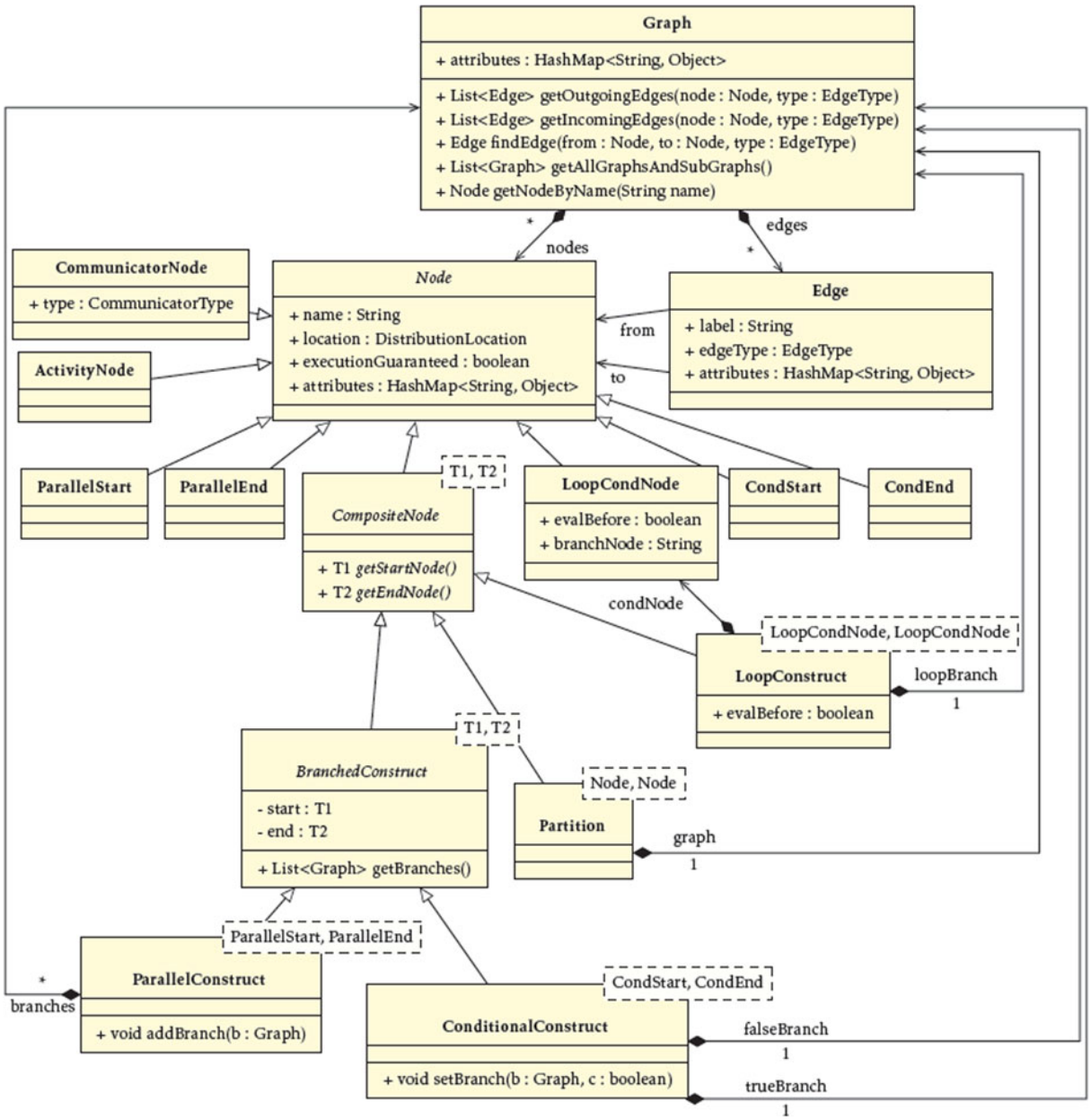


Fig. 15 Data structures of the decomposition transformation

of each communicator can be set by using the type attribute.

– *CompositeNode* consists of at least one subgraph. Each composite node has functions for getting the start and end nodes of the construct. These functions are implemented by each child of the CompositeNode. The template defined for CompositeNode is used for defining the type of the start and end node.
– *Partition* is used to group adjacent nodes with the same distribution location.
– *BranchedConstruct* allows an execution branch to be split up into multiple executing branches. We defined two subtypes of this construct: ParallelConstruct and ConditionalConstruct. A ParallelConstruct uses ParallelStart and ParallelEnd as the start and end node for the construct, respectively;

in the case of a ConditionalConstruct, the CondStart and CondEnd nodes are used as start and end node, respectively.
- *LoopConstruct* can be used for modeling loops. The loop construct has a reference to a LoopCondNode, which is the conditional node, and a loopBranch, which is the graph that is executed when the condition that is evaluated yields true. Attribute evalBefore is used for defining if the condition is evaluated before or after the execution of the loop branch (always set to "true" in this case).

6.3 Core Transformation Definition

Our core transformation takes a process as a graph as input, and transforms it into multiple collaborating processes. The transformation has rather complex goals, and could not be implemented in a single shot, so we split the transformation into four consecutive phases that modify the input graph to produce the decomposition. We briefly discuss the operation of each phase.

Phase 1: Identification. This phase collects all the subgraphs, branched constructs and loop constructs that are nested in the graph, and mark each node with its desired distribution location. In addition, temporary nodes are added to the beginning of branches of branched constructs and loop constructs. These temporary nodes have the same distribution location as the surrounding construct, and are necessary for correctly transforming the branches in later phases. This phase gets as input a graph that defines the original process and the activity distribution list, and it returns a list with all the subgraphs that represent process fragments (subprocesses of the original process), a list with all the branched constructs and a list with all the loop constructs.

Phase 2: Partitioning. This phase partitions adjacent nodes with the same distribution location. These nodes should be placed together in one process, and are therefore grouped together in a partition. This phase gets as input the list with all the identified process fragments from the previous phase, and groups the nodes within the process fragments in partitions.

Phase 3: Communicator node creation. This phase walks through all the graphs and creates communicators between partitions. The first two partitions found are examined by the algorithm. In both processes, communicator nodes are introduced and communication edges are added to the graph. The control edge that was present between the two partitions is deleted from the graph. If there is a third partition, the algorithm removes the edge between the second and third partition, and merges the third partition with the first partition, since the third partition always has the same distribution location as the first partition. After the merge, the algorithm is repeated, until all the communicators are created between the partitions and there are no partitions left to be merged. This phase gets as input the list with all the identified process fragments from the previous phase, and makes adjustments to the inserted graphs.

Phase 4: Choreography creation. This phase removes the temporary nodes from the branched constructs and collects all the created processes, the communication edges and the data edges. This phase gets as input the list with all the identified graphs, the list with all the branched constructs and the list with all the loop constructs, and returns a graph on which the transformations are performed, a list with the communication edges and list with all the data edges.

We discuss each transformation phase in detail in the sequel.

6.4 Identification Phase

The input of the decomposition transformation is one single process and an activity distribution list. In this step, an algorithm examines the process and identifies composite constructs, loop constructs and branches within these constructs.

During the identification process, two additional tasks are performed: (1) each node of the graph is marked with its distribution location, as defined in the distribution list. In the case of conditional and flow constructs, the distribution location of the start node of the construct is used as distribution location for both the start and the

end node of the construct; and (2) for each branch of a conditional or flow construct, and for each loop branch, a new temporary start node is added to the beginning of the branch. The temporary start node is marked with the distribution location of the start node of the composite construct. This node is used during the merging phase.

Algorithm 1 shows the pseudo code for the identification phase. The algorithm is started with a call to the IdentifyProcessesAndMark procedure, with as parameters the start node of the input graph and the input graph itself. DistrLoc is a function that returns the desired distribution location for each node.

Algorithm 1 Identification and marking algorithm

```
1: BranchedConstructs ← {}
2: LoopConstructs ← {}
3: Processes ← {}

4: procedure IDENTIFYPROCESSESANDMARK(n, g)
5:     if n of type BranchedConstruct then
6:         BranchedConstructs ← BranchedConstructs ∪ {n}
7:         d ← distrLoc(n.start)
8:         n.location, n.start.location, n.end.location ← d    ▷ Mark with distribution location
9:         for all b ∈ n.getBranches() do
10:            if |b.nodes| > 0 then
11:                WorkOnBranch(b, d)
12:            end if
13:        end for
14:    else if n of type LoopConstruct then
15:        LoopConstructs ← LoopConstructs ∪ {n}
16:        d ← distrLoc(n.condition)          ▷ Mark with distribution location
17:        n.condition.location, n.location ← d          ▷ Mark with distribution location
18:        WorkOnBranch(n.loopBranch, d)
19:    else
20:        n.location ← distrLoc(n)          ▷ Mark with distribution location
21:    end if
22:    for all e ∈ g.getOutgoingEdges(n, Control) do          ▷ Follow outgoing edges
23:        IdentifyProcessesAndMark(e.to, g)
24:    end for
25: end procedure

26: procedure WORKONBRANCH(branch, d)
27:    Processes ← Processes ∪ {branch}
28:    oldStart ← branch.start
29:    newNode ← new ActivityNode()          ▷ Add temp start node
30:    newNode.location ← d          ▷ Mark with distribution location
31:    branch.nodes ← branch.nodes ∪ {newNode}
32:    branch.start ← newNode
33:    branch.edges ← branch.edges ∪ {new Edge(newNode, Control, oldStart)}
34:    IdentifyProcessesAndMark(oldStart, branch)
35: end procedure
```

6.5 Partitioning Phase

During the partitioning phase, adjacent nodes with the same distribution location are allocated to the same partition. The algorithm is performed on each of the identified processes. The algorithm walks through each process fragment and compares the distribution location of a node with the distribution location of its successor node. When the distribution locations are the same, the nodes are placed in the same partition. Otherwise, both nodes are placed in different partitions, and a new control edge is created to connect the partitions to each other.

By applying this algorithm, odd partitions are merged, whereas even partitions become separate processes. In a possible optimization step to be defined as future work, even partitions could be merged. Algorithm 2 shows the pseudo code of the partitioning algorithm.

Algorithm 2 Partitioning algorithm

```
1: procedure PARTITIONGRAPHS(Processes)
2:     for all g ∈ Processes do
3:         startNode ← g.start
4:         p ← new Partition()          ▷ Create initial partition
5:         p.location ← startNode.location
6:         g.nodes ← g.nodes ∪ {p}
7:         g.start ← p
8:         partitionGraph(startNode, g, p)
9:     end for
10: end procedure

11: procedure PARTITIONGRAPH(n, g, p)
12:     p.graph.nodes ← p.graph.nodes ∪ {n}          ▷ Add node to partition
13:     g.nodes ← g.nodes − {n}          ▷ Remove node from graph
14:     Edges ← g.getOutgoingEdges(n, Control)
15:     if |Edges| = 1 then
16:         e ← Edges.get(0)
17:         g.edges ← g.edges − {e}          ▷ Remove edge from graph
18:         if e.to.location = n.location then ▷ Following node in the same partition
19:             p.edges ← p.edges ∪ {e}
20:             PartitionGraph(e.to, g, p)
21:         else
22:             newPart ← new Partition()          ▷ Following node in a new partition
23:             newPart.location ← e.to.location
24:             g.nodes ← g.nodes ∪ {newPart}          ▷ Add new partition to graph
25:             g.edges ← g.edges ∪ {new Edge(p, Control, newPart)}          ▷ Create edge between partitions
26:             PartitionGraph(e.to, g, newPart)
27:         end if
28:     end if
29: end procedure
```

Figure 16 shows a graphical example of the steps that are performed by the algorithm. The dashed blocks around the nodes in Fig. 16 represent the partitions. Activity 1 and 2 are marked for on-premise distribution, and activity 3 and 4 are marked for being moved to the cloud (colored nodes), as shown in Fig. 16a. At first, a new partition is created and the first activity (act1) is added to the partition (p1), shown in Fig. 16b. The successor node of activity 1 is examined. Since the successor node (act2) has the same distribution location as the current node (act1), the successor is added to the same partition as act1 and the control edge between the activities is also moved to the partition, as shown in Fig. 16c. The algorithm moves on, by looking at the successor of activity 2, which is activity 3 (act3). The distribution location of activity 3 is different than the distribution location of activity 2, which means that a new partition should be created for activity 3. A new partition (p2) is created and activity 3 is placed in this partition. The control edge between activity 2 and activity 3 is removed, and a new control edge is created to connect the previous partition (p1) and the newly created partition (p2). This situation is shown in Fig. 16d, where the colored arrow between p1 and p2 represents the newly created control edge. The next step is to examine the successor of activity 3, which is activity 4 (act4). Activity 4 has the same distribution location as activity 3, which means that the node can be added to the same partition. The edge between the nodes is also moved to the partition. This situation is shown in Fig. 16e. Since there are no nodes left in the process to be examined, the algorithm terminates.

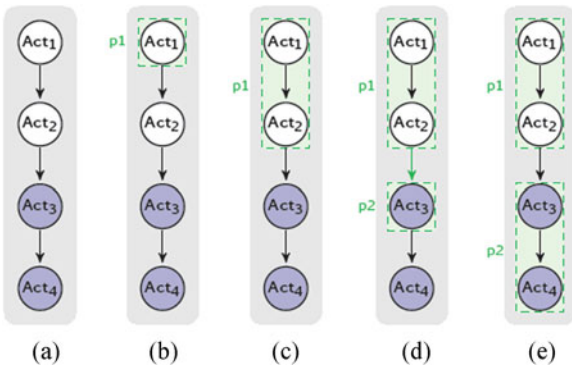


Fig. 16 Example of the partitioning algorithm

6.6 Communicator Node Creation Phase

After the nodes in the processes are partitioned, communicators must be created to allow the partitions to communicate. Communication between processes is implemented by using synchronous communication nodes. The algorithm takes the first partition of a process and identifies if there is a succeeding partition. If this is the case, communication nodes are introduced at the end of the first partition for invoking the second partition. The second partition is delimited by a receive and a reply communicator node. The control edge that was present between the partitions is removed and replaced by communication edges. If there is a third partition, this partition should have the same distribution location as the first partition, since only two distribution locations are supported. The algorithm removes the control edge between the second and the third partition, and merges the first and the third partition. The algorithm can now recur, until all communicators are created and no partitions can be merged anymore.

During the first phase of the decomposition algorithm, we introduced temporary nodes at the beginning of each branch. The function of these nodes for the decomposition process should now become clear. When the decomposition algorithm is performed, the start node of the process determines where the process is deployed. Consider a parallel construct that is marked for on-premise allocation, but the first activity within one of the branches has been marked for allocation in the cloud. Since the branches are considered as being separate processes, the decomposition algorithm is performed on the branch, and the algorithm would think that the branch should be distributed in the cloud, whereas the surrounding construct is situated on-premise. By introducing a temporary node with the same distribution location as the surrounding construct to the beginning of the branch, the algorithm knows where the process should be distributed and creates correct communicators.

This algorithm has been formalized in the Master thesis Duipmans [7], but this formalization is

omitted here due to space limitations. Figure 17 illustrates the operation of the algorithm with an example.

Consider the partitioned process depicted in Fig. 17a. The algorithm starts by taking the first partition and adding two communicator nodes (ireq, ires) connected by a control edge at the end of the first partition. The second partition is placed in the cloud and surrounded with a receive and reply node. The first partition, which is extended with invocation nodes, is merged with the third partition, in which activity 4 is placed. This situation is shown in Fig. 17b. The next step of the algorithm is to examine partition 1 again. Since there is a successor partition after partition 1, namely the partition in which activity 5 is placed, a communicator should be created. The partition in which activity 5 is placed is moved to a separate process, and is surrounded with a receive and reply node. Invocation nodes are added at the end of the first partition to invoke the newly created process. The algorithm can terminate now, since there are no other partitions to process after partition 1.

6.7 Choreography Creation Phase

In the last phase of the decomposition algorithm, all created processes, communication edges and data edges are collected. The temporary nodes that were added to the beginning of the branches are removed. The processes, communication edges and data edges together form the *choreography description* for the decomposed business process, i.e., they determine how the resulting processes cooperate.

After the previous phases, all the graphs consist of partitions that are connected to each other by communication edges. Each partition is collected and is used as a separate process. However, the first partition in a process might be part of a composite construct and, therefore, is part of another process, namely the process in which the composite construct is placed. This is why the first step of this phase is to walk through the composite constructs and collect the created processes.

The functions performed in this phase have been formalized in the Master thesis Duipmans [7], but are omitted here due to space limitations.

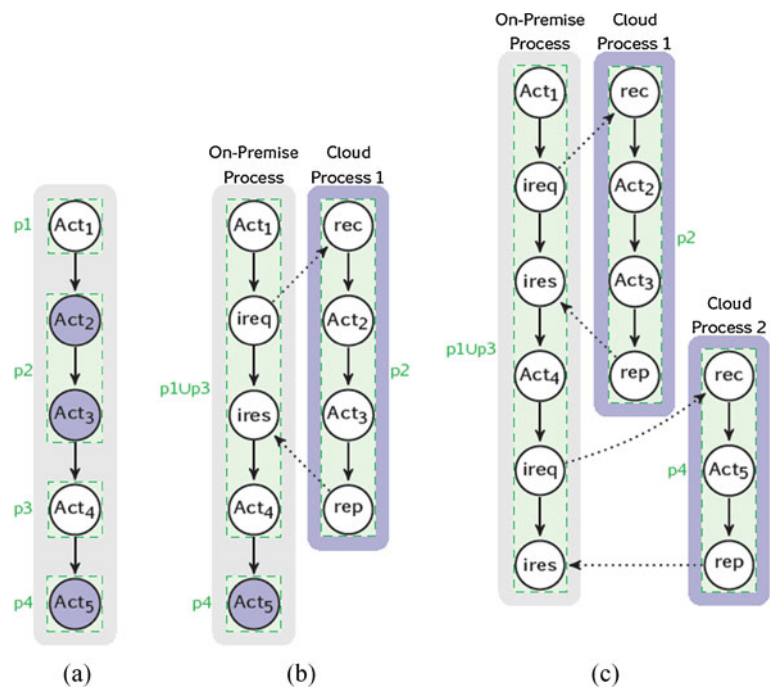


Fig. 17 Example of the communicator creation algorithm

6.8 Data Dependency Verification

Once the decomposition algorithm finishes, an algorithm for data verification has to be performed to check if no data restrictions have been violated as a result of the decomposition transformation. The algorithm we have implemented assumes that the process engine on which the process will eventually be executed uses an execution strategy in which variables are used for passing data between activities. Engines that execute WS-BPEL [24] processes, for example, comply with this assumption.

The last phase of the decomposition algorithm results in three lists: (1) list of all the created processes; (2) list of communication edges between processes; and (3) list of data dependencies. These lists form the input for the verification algorithm.

This algorithm has a Validate function that walks through the list of all the data edges. For each data edge, the 'from' (n1) and 'to' node (n2) are selected. The label on the edge identifies the data item that is involved in the data dependency relation. Function nodeInWhichGraph is used to determine in which process the nodes are used. When the nodes are not in the same process, function findNode is used to find the path that should be walked to get from n1 to n2. The nodes that were visited during the walk are collected in a list and represent a walked path. After the path is found, function validatePath is used to check if a data restriction is violated by the current data edge relation. The data restriction is violated whenever there is a node in the path list with a different distribution destination than the data restriction location for the current data item. The nodes that violate a certain data restriction are collected in a list and returned by the algorithm.

This algorithm has also been formalized in the Master thesis Duipmans [7], but is omitted here due to space limitations.

7 Case Study

We performed a case study in order to demonstrate the applicability of our transformation-based approach. The case study consists of a talent show audition process that has activities and data that should be allocated on-premise and in the cloud. Below we discuss how the transformation chain shown in Fig. 4 has been applied to the case study.

7.1 Description

Consider that a television broadcast company wants to produce a new singing competition show. The company uses an on-line registration system, in which contestants can register for the show. In order to get selected for the show, contestants need to upload an audition video, in which they are performing a song, and some personal information, so that producers can contact a contestant in case she is selected for the show. The selection procedure of the contestants is as follows. The producers and a jury first look at all the videos and directly select contestants for the show. The other video auditions are placed on the website of the show and visitors of the website can vote on the videos they like. The highest voted video auditions are selected and their performers are added to the list of contestants. Figure 18 shows the business process of the on-line registration system in Amber [9], which is the business process language used in this case study.

The process starts when the user uploads a video. After the video is uploaded, the process is split up into two separate parallel branches. One branch performs operations on the uploaded video. The video is stored in a folder on the server and, after that, a verification algorithm is used to check if the video is valid. This operation also determines the video properties, such as the format, size and quality. For the producers it is important that the videos are all in the same format, in order to speed up their selection process. In addition, videos are placed on the website in a single specific video format, therefore, a conversion step is necessary in case a video does not comply to the selected format. After conversion, a unique video identifier is assigned to the video. The other branch of the process waits for personal information that should be submitted by the user. When the information is provided, the personal information is stored in a database.

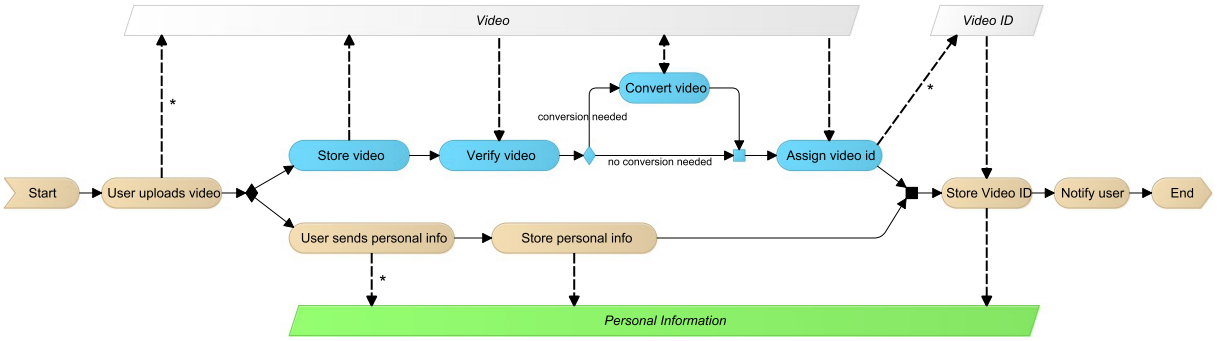


Fig. 18 Business process of the on-line registration system

After the branches merge, the video identifier should be stored in the database that contains personal information. This step is necessary to know which video audition belongs to which personal information. After the personal information is updated, a notification is sent to the user and the business process is terminated.

7.2 Marking Activities and Data Items

We assume that the television broadcast company expects a large amount of auditions. Since the storage of videos might take a lot of space, and operations on the videos, such as video conversion, are computation-intensive, we also assume that the company has decided to make use of cloud computing for storing the videos and performing operations on them. The personal information manipulated by the process, however, should stay within the premises of the television broadcast organization. Therefore, the business process mainly runs on the on-premise server, while parts of the process are outsourced to the cloud.

Figure 18 shows the markings of the distribution locations and data restrictions on the business process. Activities that should be performed in the cloud are marked with the cloud flag. In Fig. 18, these activities are marked with a dark background color. The personal information data item is marked with a data restriction, which states that the item should stay on-premise. This is represented in Fig. 18 with a shaded data item.

7.3 Lifting

Once activities have been marked with a distribution location, and a data restriction has been placed on the Personal Information data item, the transformation chain can start with the lifting transformation (Transformation 1 in Fig. 4). The first step of this transformation is to export the business process to an XML representation, which has been done with the help of BiZZdesigner,[1] which is the tool in which the business process has been edited. After that, the exported XML file has been imported by our Java application and a new instance of the intermediate model has been created. In addition to our Java transformations, we also implemented a graphical export function to show the intermediate results during the transformations. Figure 19a shows the intermediate model that has been generated from the imported XML file. Figure 19b shows the graphical representation of the intermediate model, in which the parallel and conditional nodes are captured within composite constructs.

Once composite constructs are created in the intermediate model, a data dependency analysis is performed. Two data dependencies identified in this step are discussed below as examples:

1. Node n7 has a data dependency edge to itself, which means that the data item (in this case Video) is created during the execution of the activity.

[1] http://bizzdesign.com/tools/bizzdesigner/

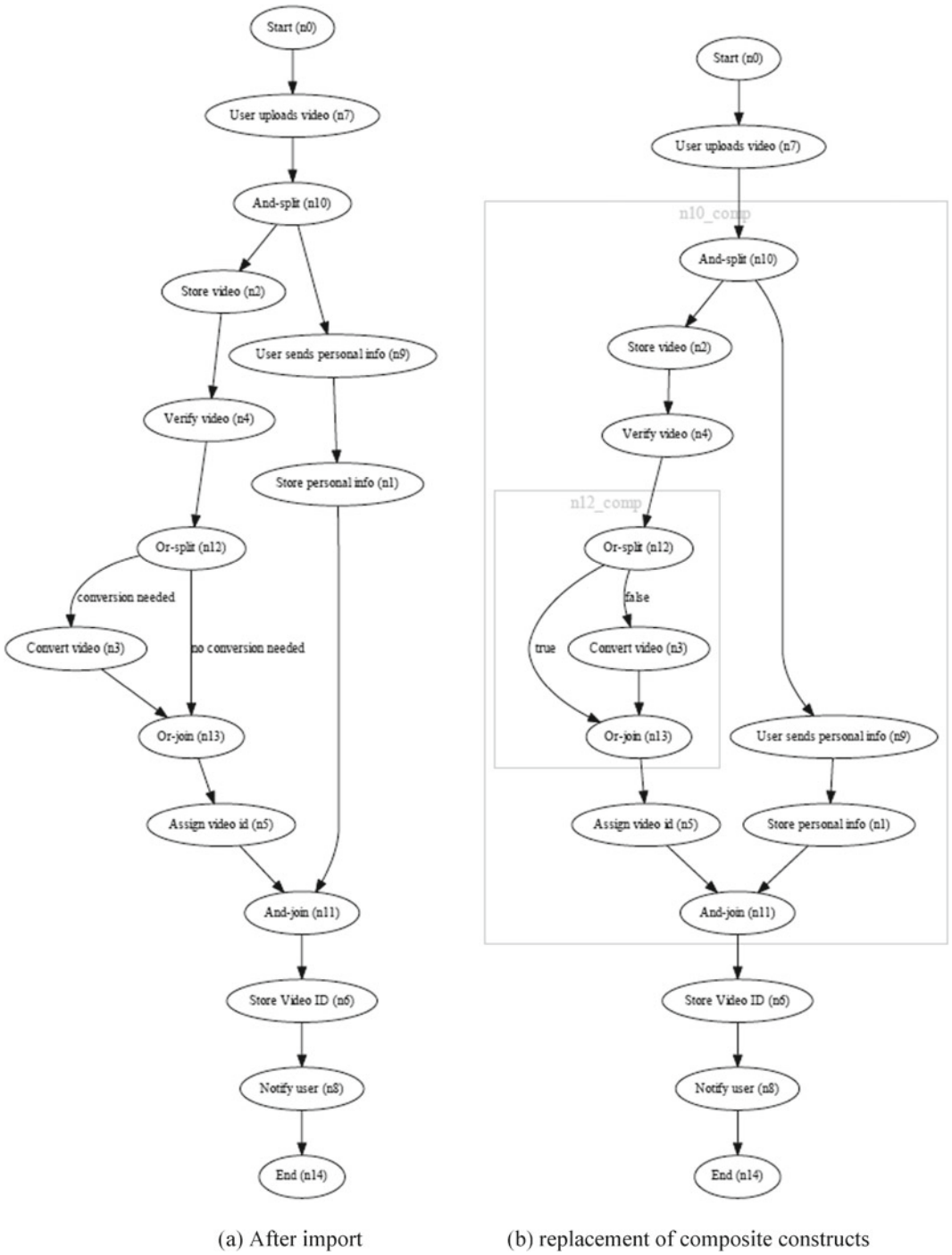


Fig. 19 Intermediate model representations

2. Activity Assign video id (n5) has two data dependencies, both relating to the Video data item. The two incoming data dependencies mean that both activities from which the data dependency edges originate are possible writers to the Video data item. Since the execution

of activity Convert Video (n3) is not guaranteed because this activity is in a conditional branch, activity n5 does not know if n3 has written to the item. Therefore, activity n2 is also a possible writer, and a data dependency edge exists between n2 and n5.

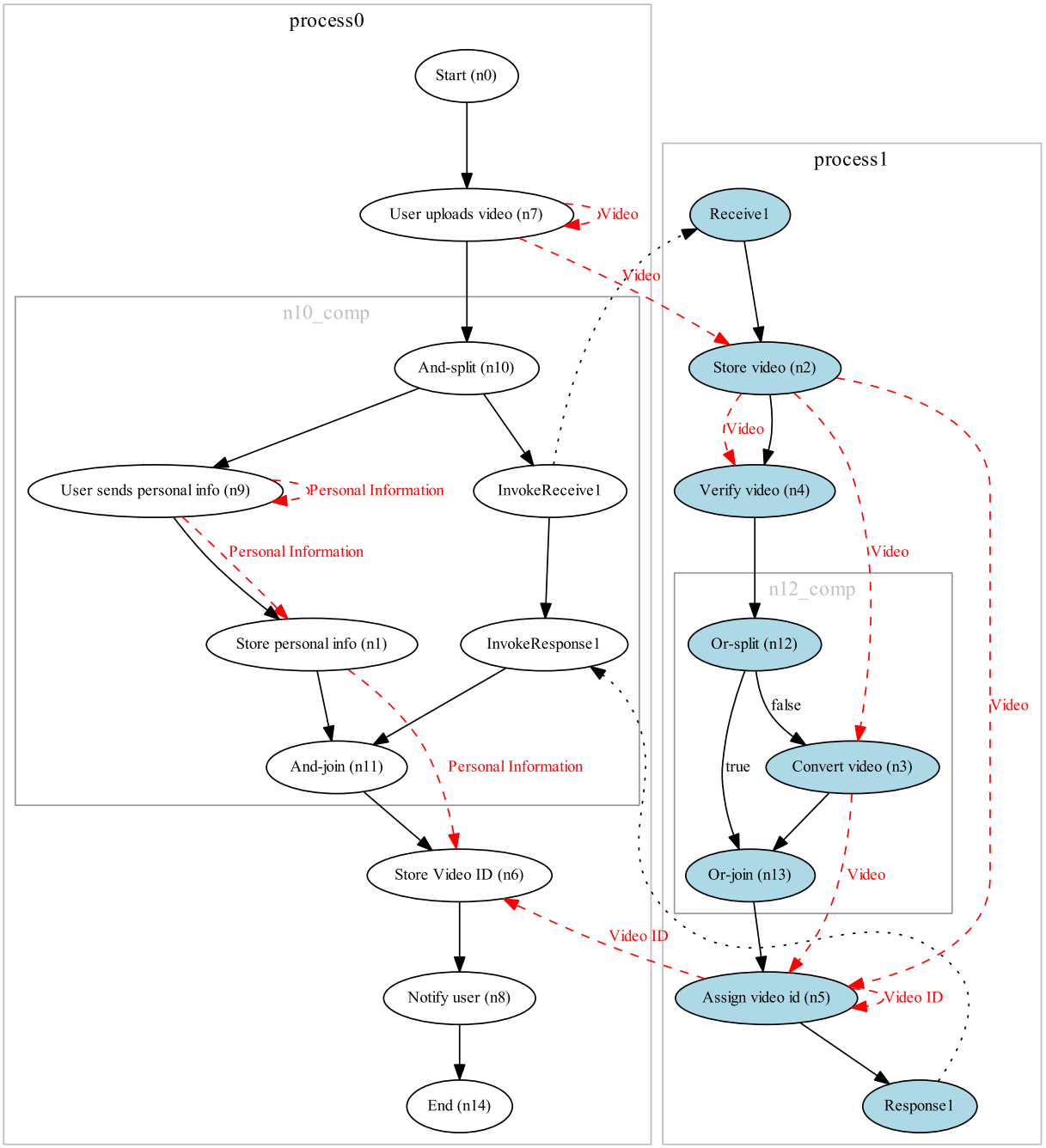


Fig. 20 Intermediate model after the choreography creation phase

7.4 Core Transformation

The core transformation can be started after the data dependencies have been determined. Each phase of the core transformation is briefly discussed below.

- *Identification*: the activities that need to be distributed in the cloud are marked. In addition, in each of the branches of a composite construct, a temporary node is added with the same distribution location as the composite construct.
- *Partitioning*: adjacent nodes marked with the same distribution location are placed together in a partition. Each of the subgraphs (branches) is treated as a separate process, therefore within a partition there might be multiple partitions within a composite construct.
- *Creation of communicator nodes*: communicators are created between partitions. Consider that Partition1 and Partition2 have to communicate, where Partition1 is allocated on-premise and Partition2 is marked for movement to the cloud. Partition1 is extended with invocation nodes, and Partition2 is extended with a receive-node at the beginning of the partition and a reply-node at the end of the partition.
- *Choreography creation*: after the communicators are created, the separate processes are collected and the temporary nodes that were added in the identification phase are removed. Figure 20 shows the result obtained after this phase.
- *Data restriction verification*: data restriction violations are verified. The verification algorithm collects the data items that were violated, and selects the activities that violate these data items. The information obtained in this phase is used during the grounding transformation. In this example no data restrictions are violated.

7.5 Grounding

The final step in the transformation chain is the grounding transformation. In this case study, the

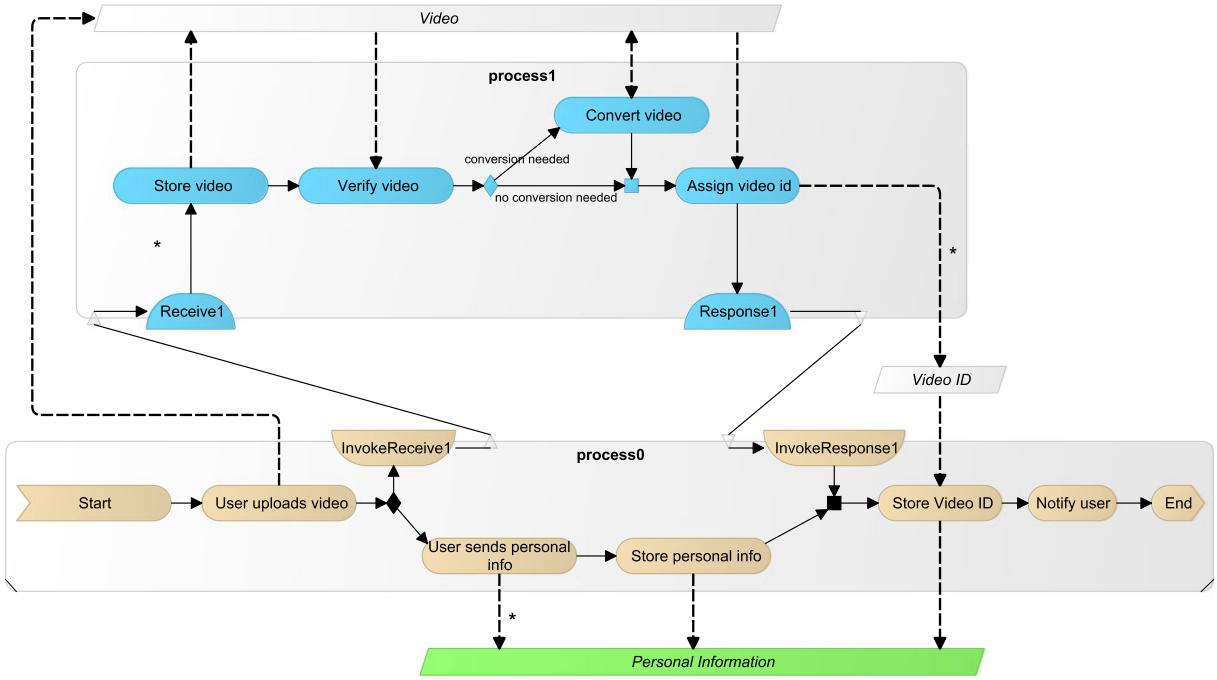


Fig. 21 Business process with marked activities and data restrictions

intermediate model is transformed back to an Amber model. The intermediate model is then transformed into a format that can be imported by BiZZdesigner. During the import phase, the XML format is converted into a new behavioral model in BiZZdesigner and the resulting process model is shown in Fig. 21. This process model consists of two collaborating processes one for deployment on-premise and another one for deployment in the cloud.

This case study demonstrates that the transformations can be performed automatically. The Java implementation has been extended with a function to export intermediate results as images. The initial process was created by hand and the marking of activities and data item was also performed by hand. The layout of the resulting model was obtained manually, since BiZZdesigner has no automatic layout functionality, but the resulting process model itself has been obtained fully automatically.

8 Related Work

Our work has been motivated by Han et al. [12], and extends this work with an additional distribution pattern in which process engines are replicated on-premise and in the cloud. This requires process decomposition techniques, so we looked for *process decomposition techniques* that could be readily applied in our work.

Several research groups have investigated the possibility of decentralizing orchestrations. In a centralized orchestration, a process is coordinated by a single orchestrator, while in decentralized orchestration, different orchestrations are distributed among several orchestrators. By distributing parts of a process over separate orchestrators, the message overhead may be reduced, which potentially leads to better response time and throughput [22].

In Nanda and Karnik [22], Nanda et al. [23] and Chafle et al. [5, 6], new orchestrations are created for each service that is used within a business process, hereby creating direct communication between services instead of having a single orchestrator to coordinate the services. The business processes are defined in WS-BPEL [24]. Their work not only defines a decomposition, but also analyzes synchronization issues. The work captures a WS-BPEL process in a control flow graph, which is used in turn to create a Program Dependency Graph (PDG, Ferrante et al. [11]). The transformations are performed on PDGs, and the newly created graphs are transformed back into WS-BPEL. The partitioning approach is based on the observation that each service in the process corresponds to a fixed node, and for each fixed node a partition is generated. In our approach we had to create processes in which multiple services can be used, so this partitioning algorithm is not particularly suitable for our case.

Research in Khalaf and Leymann [15], Khalaf et al. [16] and Kopp et al. [17] focuses on the decentralization of orchestrations by using WS-BPEL processes. The main focus of the research is to use Dead Path Elimination (DPE) [24], for ensuring the execution completion of decentralized processes. DPE is a specific feature of WS-BPEL, which is a consequence of allowing join conditions of links between activities. Since DPE is so language-specific, these approaches are only useful when WS-BPEL is selected as the input and output language of the transformation framework.

In Baresi et al. [4], decentralization of WS-BPEL processes is considered, and the authors use a graph transformation approach for transforming the WS-BPEL process. However, the transformation rules are not defined in the paper. The type graph with which the graph transformations are performed might be applicable to our situation, but this would require some additional research.

In Fdhila et al. [10], the authors state that the current research on decentralizing orchestrations focuses too much on specific business process languages. In most cases, implementation level languages, such as WS-BPEL, are used. In our situation, the decision for distributing activities and data to the cloud is not only based on performance issues, but also on safety measures, dictated by organizations or by the government. This implies that the decision to execute an activity on-premise or in the cloud might already be taken in the design phase of the BPM lifecycle, when the business processes are conceived.

Some approaches to process decomposition discussed in the literature make use of process

models represented as Petri Nets. In van der Aalst [1], a Petri Net-based approach is introduced to analyze so called *interorganizational processes*, which are processes that run in different organizations but are required to interact. This work concentrates on the consistency of the interactions between these processes, mostly to avoid deadlocks, so this approach is not directly applicable to our case. In Tan and Fan [33], a Petri Net-based approach is proposed to dynamically decompose a workflow in workflow fragments while it is being executed, so that it can be distributed and executed by different workflow engines. Although this approach supports workflow decomposition, it has been devised to be applied in a scenario that is completely different than the one which we envisaged for our decomposition support, namely that the decomposition would be performed prior to deployment (not at runtime). Similarly to Tan and Fan [33], in Sun et al. [32] dynamic process fragmentation is investigated, but in this case process mining techniques are used to optimize the fragmentation with respect to execution timing constraints.

9 Final Remarks

Below we present our conclusions and recommendations for future work.

9.1 Conclusions

In this paper we reported on our transformation-based approach to decompose monolithic business processes into multiple processes that can be executed in the cloud or on-premise. The decomposition is driven by a distribution list, in which the activities of the original business process are marked with their desired distribution locations, and data restrictions can be added, to ensure that data items stay within a certain location (on premise or in the cloud).

We defined a transformation chain for our approach. We decide to introduce an intermediate model for defining the decomposition transformation. This intermediate model is defined at a semantic level and captures the main concepts of business processes. The decomposition transformation was designed to operate on the intermediate model. By performing the operations on the intermediate model, the decomposition solution is business process language-independent and is suitable for processes defined in both the design and the implementation phase of the BPM lifecycle. In order to work with existing business process languages, transformations are needed for converting an existing business process language into the intermediate model and back, the so-called lifting and grounding transformation, respectively.

An analysis was performed to identify the decomposition rules that should be supported. From these rules, a selection was made for the implementation of the transformation. The algorithm that was used for the decomposition transformation was first prototyped using graph transformations in Groove [29]. After that, the algorithm was implemented in Java, and the graph-based transformation has been used to verify the results of the transformation implemented in Java. We also built a verification algorithm to verify if data restrictions are violated as a result of the decomposition transformation.

In this paper we reported on a case study in which Amber [9] has been used as business process language, so that we developed the lifting and grounding transformations for this language. Algorithms were designed for replacing conditional, parallel and loop nodes by block structured elements, and a data dependency analysis algorithm was designed for discovering data dependencies between activities. The case study has demonstrated the feasibility of our approach, since the transformation could be completely automated. In Povoa et al. [28] we substantiate our claim that our solution is capable of supporting different business process languages with an experiment in which lifting and grounding transformations been developed for (a subset of) WS-BPEL, allowing the central transformation and the intermediate model to be reused and applied to a simple process of the health domain.

9.2 Future Work

Since we have introduced some limitations in the business processes and activity and data allocations supported by our transformations, a natural

extension of this work is to extend the transformations in order to support more situations. However, we also suggest that some research shoud be done to investigate which process patterns are mostly used in practice, to determine which percentage of the process patterns used in practice are already covered by our transformations.

An optimization phase could be added to the decomposition transformation, to combine some of the newly created processes, to reduce the amount of data that is sent between processes. For example, in the communicator node creation phase odd partitions are merged, whereas even partitions are identified as new processes. A possible optimization would be to merge even partitions, based upon data dependency relations between nodes in partitions. Consider two even partitions (p2 and p4) with a data dependency between a node in p2 and a node in p4. When the partitions are separate processes, data need to be sent from p2 to p4 via an intermediate odd partition (p1). By merging the partitions, data are directly available for the activity in p4, and therefore, no data need to be sent from p2 to p4. We implemented a simple version of this optimization, in which also the even partitions are merged. This solution was added to the Java implementation and can be selected with an extra input parameter, in which the partition merge type can be selected. In future work, a more advanced solution can be devised in which data dependencies between partitions are used as basis for merging even partitions.

We verified the correctness of the decomposition solution informally by testing the solution on several business processes. For each of the obtained results we removed the communication nodes and replaced the communication edges with control edges, which resulted in the original processes again, i.e., we verified whether the resulting solution preserves the observable behavior of the original process. This indicates that the behavior of the process itself is not changed by the transformation and no information from the original process is lost during the decomposition transformation. In future work, a formal validation of the transformations could be performed. An approach to perform this validation could be to formally prove that each decomposition step in our transformations preserves observable behavior, and that observable behaviour is not disturbed when the transformations are applied in the transformation chain. As a matter of fact, each transformation rule discussed in Section 5 (and in the Master thesis Duipmans [7]) has been defined to preserve observable behavior, already giving an indication of the soundness of the proposed approach.

In future work we intend to develop a comprehensive environment to cover not only the decomposition of business processes in different business process languages, but also their automated deployment in process engines and their execution monitoring. An interesting line of work can also be to apply dynamic process decomposition techniques like the ones reported in Tan and Fan [33] and Sun et al. [32] to a monolithic process model, and perform on-premise and cloud deployment on-the-fly (during execution).

References

1. van der Aalst, W.: Interorganizational workflows: an approach based on message sequence charts and petri nets. Syst. Anal. Model. Simul. **34**(3), 335–367 (1999)
2. van der Aalst, W.P., ter Hofstede, A., Kiepuszewski, B., Barros, A.: Workflow patterns. Distrib. Parallel Dat. **14**(1), 5–51 (2003)
3. Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: a Berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley (2009). http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html. Accessed 2 Oct 2013
4. Baresi, L., Maurino, A., Modafferi, S.: Towards distributed BPEL orchestrations. In: Electronic Communication of the European Association of Software Science and Technology, vol. 3 (2006)
5. Chafle, G., Chandra, S., Mann, V., Nanda, M.G.: Orchestrating composite web services under data flow constraints. In: Proceedings of the IEEE International Conference on Web Services, ICWS '05, pp. 211–218. IEEE Computer Society, Washington (2005). doi:10.1109/ICWS.2005.88
6. Chafle, G.B., Chandra, S., Mann, V., Nanda, M.G.: Decentralized orchestration of composite web services. In: Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters, WWW Alt. '04, pp. 134–143. ACM, New York (2004). doi:10.1145/1013367.1013390

7. Duipmans, E.: Business process management in the cloud with data and activity distribution. Master's thesis, University of Twente (2012)
8. Duipmans, E., Ferreira Pires, L., Bonino da Silva Santos, L.O.: Towards a BPM cloud architecture with data and activity distribution. In: IEEE 16th International Enterprise Distributed Object Computing Conference Workshops, pp. 165–171. IEEE Computer Society, Washington (2012)
9. Eertink, H., Janssen, W., Luttighuis, P., Teeuw, W., Vissers, C.: A business process design language. In: FM99 Formal Methods. Springer, Heidelberg (1999)
10. Fdhila, W., Yildiz, U., Godart, C.: A flexible approach for automatic process decentralization using dependency tables. In: ICWS, pp. 847–855 (2009)
11. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987). doi:10.1145/24039.24041
12. Han, Y.B., Sun, J.Y., Wang, G.L., Li, H.F.: A cloud-based BPM architecture with user-end distribution of non-compute-intensive activities and sensitive data. J. Comput. Sci. Technol. **25**(6), 1157–1167 (2010)
13. Jensen, K.: Coloured Petri nets. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Petri Nets: Central Models and Their Properties. Lecture Notes in Computer Science, vol. 254, pp. 248–299. Springer, Heidelberg (1987). doi:10.1007/BFb0046842
14. Jouault, F., Kurtev, I.: Transforming models with atl. In: Proceedings of the 2005 International Conference on Satellite Events at the MoDELS, MoDELS'05, pp. 128–138. Springer, Heidelberg (2006). doi:10.1007/11663430_14
15. Khalaf, R., Leymann, F.: E role-based decomposition of business processes using BPEL. In: Proceedings of the IEEE International Conference on Web Services, ICWS '06, pp. 770–780 IEEE Computer Society, Washington (2006). doi:10.1109/ICWS.2006.56
16. Khalaf, R., Kopp, O., Leymann, F.: Maintaining data dependencies across bpel process fragments. In: Proceedings of the 5th International Conference on Service-Oriented Computing, ICSOC '07, pp. 207–219. Springer, Heidelberg (2007). doi:10.1007/978-3-540-74974-5_17
17. Kopp, O., Khalaf, R., Leymann, F.: Deriving explicit data links in WS-BPEL processes. In: Proceedings of the 2008 IEEE International Conference on Services Computing, SCC '08, vol. 2, pp. 367–376. IEEE Computer Society, Washington (2008). doi:10.1109/SCC.2008.122
18. Kopp, O., Martin, D., Wutke, D., Leymann, F.: The difference between graph-based and block-structured business process modelling languages. EMISA **4**(1), 3–13 (2009). http://dblp.uni-trier.de/db/journals/emisaij/emisaij4.html
19. Mell, P., Grance, T.: The NIST definition of cloud computing. National Institute of Standards and Technology (2009). http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. Accessed 28 Apr 2010
20. Miller, J., Mukerji, J.: Mda guide version 1.0.1. Tech. rep., Object Management Group (OMG) (2003)
21. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989). doi:10.1109/5.24143
22. Nanda, M.G., Karnik, N.: Synchronization analysis for decentralizing composite web services. In: Proceedings of the 2003 ACM Symposium on Applied Computing, SAC '03, pp. 407–414. ACM, New York (2003). doi:10.1145/952532.952612
23. Nanda, M.G., Chandra, S., Sarkar, V.: Decentralizing execution of composite web services. In: Vlissides, J.M., Schmidt, D.C. (eds.) Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, pp. 170–187. ACM, Vancouver (2004). doi:10.1145/1028976.1028991
24. OASIS: Web Services Business Process Execution Language Version 2.0 (2007)
25. OMG: Business Process Model and Notation (BPMN), version 2.0 (2011). http://www.omg.org/spec/BPMN/2.0
26. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1 (2011). http://www.omg.org/spec/QVT/1.1/. Accessed 3 Oct 2013
27. Papazoglou, M.P.: Web Services: Principles and Technology. Prentice Hall (2008)
28. Povoa, L.V., de Souza, W.L., do Prado, A.F., Ferreira Pires, L., Duipmans, E.F.: An approach to business processes decomposition for cloud deployment. In: Proceedings of the 27th Brazilian Symposium on Software Engineering (SBES 2013), vol. 1, pp. 124–133. Universidade de Brasília (UnB), Brasília (2013)
29. Rensink, A., Boneva, I., Kastenberg, H., Staijen, T.: User manual for the GROOVE tool set. Tech. rep., University of Twente (2011)
30. Rimal, B.P., Jukan, A., Katsaros, D., Goeleven, Y.: Architectural requirements for cloud computing systems: an enterprise cloud approach. J. Grid Comput. **9**(1), 3–26 (2011). doi:10.1007/s10723-010-9171-y
31. Seguel Perez, R.: Business protocol adaptors for flexible chain formation and enactment. Ph.D. thesis, Eindhoven University of Technology (2012)
32. Sun, S.X., Zeng, Q., Wang, H.: Process-mining-based workflow model fragmentation for distributed execution. IEEE Trans. Syst. Man Cybern. Part A **41**(2), 294–310 (2011). http://dblp.uni-trier.de/db/journals/tsmc/tsmca41.html#SunZW11
33. Tan, W., Fan, Y.: Dynamic workflow model fragmentation for distributed execution. Comput. Ind. **58**(5), 381–391 (2007). doi:10.1016/j.compind.2006.07.004
34. W3C: Web Services Choreography Description Language Version 1.0. World Wide Web Consortium, Candidate Recommendation CR-ws-cdl-10-20051109 (2005)
35. Weske, M.: Business Process Management: Concepts, Languages, Architectures. Springer, Heidelberg (2007)