Fostering the Reuse of TOSCA-based Applications by Merging BPEL Management Plans

Sebastian Wagner[1(✉)], Uwe Breitenbücher[1], Oliver Kopp[2], Andreas Weiß[1], and Frank Leymann[1]

[1] IAAS, University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany
{wagner,breitenbucher,Weib,leymann}@informatik.uni-stuttgart.de
[2] IPVS, University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart, Germany
kopp@informatik.uni-stuttgart.de

Abstract. Complex Cloud applications consist of a variety of individual components that need to be provisioned and managed in a holistic manner to setup the overall application. The Cloud standard TOSCA can be used to describe these components, their dependencies, and their management functions. To provision or manage the Cloud application, the execution of these individual management functions can be orchestrated by executable management plans, which are workflows being able to deal with heterogeneity of the functions. Unfortunately, creating TOSCA application descriptions and management plans from scratch is time-consuming, error-prone, and needs a lot of expert knowledge. Hence, to save the amount of time and resources needed to setup the management capabilities of new Cloud applications, existing TOSCA description and plans should be reused. To enable the systematic reuse of these artifacts, we proposed in a previous paper a method for combining existing TOSCA descriptions and plans or buildings blocks thereof. One important aspect of this method is the creation of BPEL4Chor-based management choreographies for coordinating different plans and how these choreographies can be automatically consolidated back into executable plans. This paper extends the previous one by providing a much more formal description about the choreography consolidation. Therefore, a set of new algorithms is introduced describing the different steps required to consolidate the management choreography into an executable management plan. The method and the algorithms are validated by a set of tools from the Cloud application management ecosystem OpenTOSCA.

1 Introduction

Due to the steadily increasing use of information technology in enterprises, accurate development, provisioning, and management of applications becomes of crucial importance to align business and IT. While developing application components and modelling application architectures and designs is supported by sophisticated tools, application management still presents major challenges: Especially

M. Helfert et al. (Eds.): CLOSER 2016, CCIS 740, pp. 232–254, 2017.
DOI: 10.1007/978-3-319-62594-2_12

in Cloud Computing, management automation is a key prerequisite since manual management is (i) too slow to preserve Cloud properties such as elasticity and (ii) too error-prone as human operator errors account for the largest fraction of failures in distributed systems [1,2]. Thus, management automation is a key incentive in modern IT.

While various management technologies[1] exist that are capable of automating *generic* management tasks, such as automatically scaling application components or installing single software components, the automation of *complex, holistic, and application-specific management processes* is an open issue. Automating complex management processes, e. g., migrating an application component from one Cloud to another while avoiding downtime or acquiring new licenses for employed software components, typically requires the orchestration of multiple heterogeneous management technologies. Therefore, such management processes are mostly implemented using workflows languages [6], e. g., BPEL [7] or BPMN [8], since other approaches such as scripts are not capable of providing the reliability and robustness of the workflow technology [9].

Creating management processes, however, requires integrating the different invocation mechanisms, data formats, and transport protocols of each employed technology, which needs enormous time and expertise on the conceptual as well as on the technical implementation level [10].

To avoid continually reinventing the wheel for problems that have been already solved multiple times for other applications, developing new applications by reusing and combining proven (i) structural application fragments as well as (ii) the corresponding available management processes would pave the way to increase the efficiency and quality of new developments. However, while automatically combining and merging individual application structures is resolved [11], integrating the associated management processes is a highly non-trivial task that still has to be done manually. Unfortunately, similarly to manually authoring such processes, this leads to error-prone, time-consuming, and costly efforts, which is not appropriate for modern software development and operation.

In this paper, we tackle these issues. We first present a method that describes how to employ choreographies to systematically reuse existing management workflows. Choreography models enable coordinating the distributed execution of individual workflows without the need to adapt their implementation. Thus, they provide a suitable integration basis to combine different management workflows without the need to dive into or change their technical implementation.

Since choreographies are not intended to be executed on a single workflow engine—which is a mandatory requirement in application management as typically sensitive data such as credentials or certificates have to be exchanged between the coordinated workflows—they must be transformed into an executable workflow model. Therefore, we introduced in our former work [12], that was presented at the *6th International Conference on Cloud Computing and Services Science (CLOSER)*, a process consolidation approach that transforms

[1] E.g., configuration management technologies such as Chef [3] or Puppet [4], or Cloud management platforms such as Heroku [5].

a choreography including all coordinated workflow models into one single executable workflow model.

The consolidation results also in a faster execution due to reduced communication over the wire. It also simplifies deployment as only a single workflow has to be deployed instead of various interacting workflows along with the choreography specification itself. Thus, reusing management workflows following this approach leads to significant time and cost savings when developing new applications out of existing building blocks. In this paper, which is an extended version of the original paper [12], we a provide in Sect. 4 a more detailed and comprehensive description of the consolidation approach. Therefore, the new contributions are an additional set of algorithms describing the consolidation steps in a much more formal way than in the original paper. To discuss these algorithms also the choreography meta-model was extended.

To validate the presented approach, we apply the developed concepts to the choreography modelling language BPEL4Chor [13] and the Cloud standard TOSCA [14,15]. For this purpose, we developed a standard-based, open-source Cloud application management prototype by extending the OpenTOSCA ecosystem [16–18] in order to support managing applications based on choreographies, that are transparently transformed into executable workflows behind the scenes.

The remainder is structured as follows. Section 2 provides background and related work information along with a motivating scenario. In Sect. 3, we conceptually describe the method for reusing TOSCA-based applications and their management plans by introducing management choreographies. In Sect. 4 we formally discuss the step of the method, that transforms a choreography into an executable management plan. Section 5 validates the method proposed in Sect. 3 and Sect. 6 concludes the work.

2 Background and Related Work

This section discusses background and related work about (i) the Cloud standard TOSCA, (ii) management workflows, and (iii) the transformation and consolidation of choreographies. In Sect. 2.3, we introduce a motivating scenario that is used throughout the paper to explain the approach.

2.1 TOSCA and Management Plans

In this section, we introduce the *Topology and Orchestration Specification for Cloud Applications (TOSCA)*, which is an emerging standard to describe Cloud applications and their management. We explain the fundamental concepts of TOSCA that are required to understand the contributions of this paper and simplify constructs, where possible, for the sake of comprehension. For more details, we refer interested readers to the TOSCA Specification [14] and the TOSCA Primer [15]. TOSCA defines a meta-model for describing (i) the structure of an application, and (ii) their management processes. In addition, the standard introduces an archive format that enables packaging applications and all required files, e. g., installables, as portable archive that can be consumed by TOSCA runtimes to provision and manage new instances of the described application. The structure of the application is described in the form of an *application topology*, a directed graph that consists of vertices representing the components of the application and edges that describe the relationships between these components, e. g., that a *Webserver* component is *installed* on an *operating system*. Components and relationships are typed and may specify properties and management operations to be invoked. For example, a component of type *ApacheWebserver* may specify its *IP-address* as well as the *HTTP-port* and provides an operation to *deploy* new applications. In addition, required artifacts, e. g., installation scripts or binaries implementing the application functionality, may be associated with the corresponding components, relationships, and operations. Thereby, TOSCA enables describing the entire structure of an application in the form of a self-contained model, which also contains all information about employed types, properties, files, and operations. These models can be used by a TOSCA runtime to fully automatically provision instances of the application by interpreting the semantics of the modeled structure [15,19].

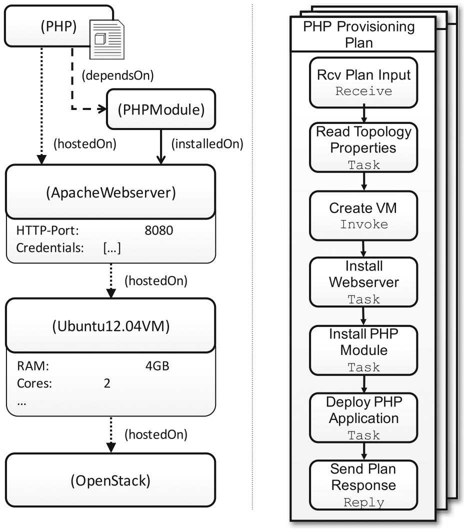

Fig. 1. TOSCA example: topology (left) and provisioning plan (right).

Figure 1 shows an example on the left rendered using Vino4TOSCA [20]. The shown topology describes a deployment consisting of a *PHP* application that is hosted on an *ApacheWebserver* running on a virtual machine (VM) of type *Ubuntu12.04VM*. This VM is operated by the Cloud management system *OpenStack*. To run the PHP application on an Apache Webserver, a *PHPModule* needs to be installed. In the topology the component types and relationship types, e. g., the desired *hostedOn*, of the VM, are put in brackets. The component properties, e. g., the desired *RAM* of the VM, are depicted below the component types. The actual application implementation, i. e., the PHP files implementing the functionality, is attached to the PHP component.

While the provisioning of simple applications can be described *implicitly* by such topology models, TOSCA also enables describing complex provisioning and management processes in the form of *explicitly* modeled *management plans*. Management plans are executable workflows that specify the (i) activities to be executed, (ii) the control flow between them, i. e., their execution order, as well as (iii) the data flow, e. g., that one activity produces data to be consumed by a subsequent activity [6]. There exists standardized workflow languages and corresponding engines, for example, BPEL [21] or BPMN [22], that enable describing workflows in a portable manner. Standard-compliant workflow engines can be employed to automatically execute these workflow models. The workflow technology is well-known for features such as reliability and robustness [6], thus, providing an ideal basis to automate management processes [7]. In addition, there are extensions of workflow standards which are explicitly tailored to the management of applications. For example, BPMN4TOSCA [8] is an extension to easily describe management plans for applications modeled in TOSCA. TOSCA supports using arbitrary workflow languages for describing executable management plans [14].

Figure 1 shows a simplified management workflow on the right that automatically provisions the application (data flow modeling is omitted for simplicity). The first activity reads properties of components and relationships from the topology model, which enables customizing the deployment without adapting the plan. Other information, e. g., the endpoint of Open Stack, are passed via the plan's start message. Using these information, the plan instantiates a new virtual machine by invoking the HTTP-API of Open Stack. Afterwards, the plan uses SSH to access the virtual machine and installs the Apache Webserver and the PHP module using Chef [3], a configuration management technology. Finally, the application files, which have been extracted from the topology, are deployed on the Webserver and the application's endpoint is returned.

The TOSCA standard additionally defines an exchange format to package topology models, types, management plans and all required files in the form of a *Cloud Service Archive (CSAR)* [14,15]. These archives are portable across standards-compliant TOSCA runtimes and provide the basis to automatically provision and manage instances of the modeled application. Runtimes such as OpenTOSCA [16] also enable automatically executing the associated management workflows, thereby, enabling the automation of the entire lifecycle of Cloud applications described in TOSCA. Thus, TOSCA provides an ideal basis for systematically reusing (i) proven application structures as well as their (ii) management processes as both can be described and linked using the standard.

2.2 Choreography Transformation

There exist manual approaches for transforming choreographies to executable processes (plans). Hofreiter et al. [23] suggest for instance a top-down approach where business partners agree on a global choreography by specifying the interaction behavior the processes of the partners have to comply with. The choreography and the corresponding processes have to be modeled in UML and the authors propose a manual transformation to BPEL. Mendling et al. [24] use the Web Service Choreography Description Language (WS-CDL) [25] to model choreographies and to generate BPEL process stubs out of it. However, these process stubs have to be also completed manually. Another drawback of WS-CDL is that it is an interaction choreography which is less expressive than interconnection models as we will briefly discuss in Sect. 3.3.

In Sect. 4 a process consolidation algorithm is presented to generate an executable process from a choreography. Existing process consolidation techniques, e. g., from Küster et al. [26] or Mendling and Simon [27], focus on merging semantically equivalent processes, which is different from the proposed consolidation algorithm that merges *complementing* processes of a choreography into a single process.

In contrast to our approach Herry et al. [28] aim to execute a former centralized management workflow in a decentralized fashion. To accomplish that they are describing an approach to decompose the management workflow into a set of different interacting agents coordinating its execution.

2.3 Motivating Scenario

This section describes a motivation scenario based on the previous example to explain the difficulties of implementing executable management plans and the significant advantage that would be enabled by an approach that facilitates systematically reusing and combining existing workflows. As described before, for provisioning the PHP-based example application several management tasks have to be performed: Open Stack's HTTP-API has to be invoked for instantiating the VM while SSH and Chef are used to install the Webserver. However, already this simple example impressively shows the difficulties: Two low-level management technologies including their invocation mechanisms, data formats, and transport protocols have to be (i) understood and (ii) orchestrated by a workflow. This requires complex data format transformations, building integration wrappers to invoke the technologies, and results in many lines of complex workflow code [10]. Thus, implementing such management plans from scratch is a labor-intensive, error-prone, and complex task that requires a lot of expertise in very different fields of technologies - reaching from *high-level* orchestration to *low-level* application management. Therefore, systematically reusing existing plans and combining them and coordinating them would significantly improve these deficiencies.

Figure 2 shows an example how TOSCA may support this vision. On the left, the provisioning plan and the topology of the TOSCA example introduced in Sect. 2.1 is shown. On the right, a topology is shown that describes the deployment of a MySQL database including the corresponding provisioning plan. This plan automatically provisions a new VM, installs the MySQL database management system, creates a new database, and inserts a specified schema, which is

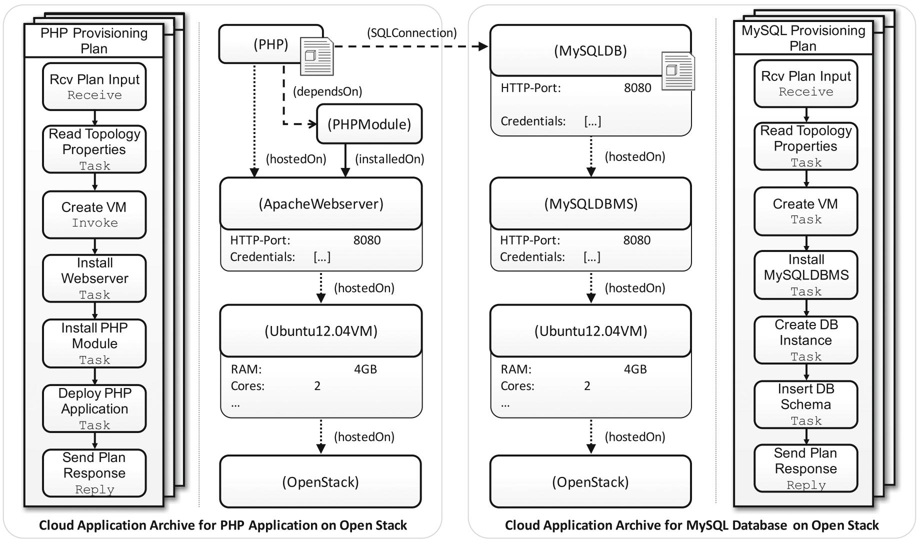

Fig. 2. Motivating scenario showing that management plans have to be combined to reuse existing topology models and management processes.

attached to the MySQL component. Thus, if a LAMP-application[2] has to be developed, the two topologies could be merged and connected with a new relationship of type *SQLConnection*. Obviously, to provision the combined stack, also their provisioning plans have to be combined. However, while merging TOSCA topology models can be done easily using tools such as Winery [17], manually combining workflow models is a crucial and error-prone task since (i) the individual control flows and possible violations have to be considered, (ii) low-level artifacts, e. g., XML schema definitions, have to be imported, and (iii) typically hundreds of lines of workflow code have to be integrated. Handling these issues manually is neither efficient nor reliable. Therefore, a systematic approach for combining TOSCA topologies and management plans is required that enables combining plans without the need to deal with their actual implementation.

3 A Method to Reuse TOSCA-based Applications

This section presents a generic method to systematically reuse existing TOSCA-based topology models and their management plans as building blocks for the development of new applications. The method is subdivided in two phases and shown in Fig. 3: (i) a *manual modeling phase*, which describes how application developers and manager model new applications by reusing existing topology

[2] An application consisting of Linux, Apache, MySQL, PHP components.

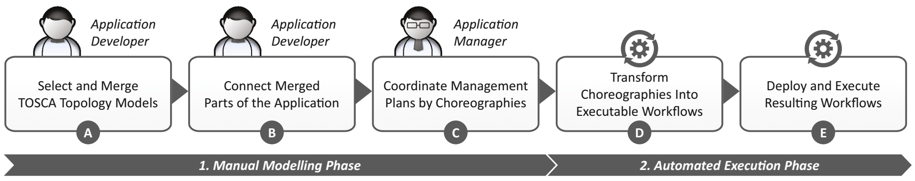
Application Developer
Select and Merge TOSCA Topology Models
A
Application Developer
Connect Merged Parts of the Application
B
Application Manager
Coordinate Management Plans by Choreographies
C
Transform Choreographies Into Executable Workflows
D
Deploy and Execute Resulting Workflows
E
1. Manual Modelling Phase
2. Automated Execution Phase

Fig. 3. Steps of the method to systematically reuse TOSCA-based (i) application topologies and (ii) their corresponding management plans.

models and plans, and (ii) an *automated execution phase*, which enables automatically deploying and managing the modeled application. The five steps of the method are explained in detail in the following.

3.1 Select and Merge TOSCA Topology Models

In the first step, the application developer sketches the desired deployment and selects appropriate TOSCA topology models from a repository to be used for its realization. The selected topologies are merged by copying them into a new topology model, which provides a *recursive aggregation model* as the result is also a topology that can be combined with others again. This is a manual step that may be supported by TOSCA modeling tools such as the open-source implementation Winery [17]. In previous works, we showed how multiple application topologies can be merged automatically while preserving their functional semantics [11] and how valid implementations for custom component types can be derived automatically from a repository of validated cloud application topologies [29]. These works support technically merging individual topologies, but the general decisions which topologies to be used are of manual nature as only developers are aware of the desired overall functionality of the application to be developed.

3.2 Connect Merged Parts of the Application

The resulting topology model contains isolated topology fragments that may have to be connected with each other. For example, the motivating scenario requires the insertion of a *SQLConnection* relationship to syntactically connect the merged topology models. Using well-defined relationship types enables specifying the respective semantics. This is also a manual step as these connections exclusively depend on the desired functionality. Moreover, TOSCA enables specifying *requirements* and *capabilities* of components, which can be used to automatically derive possible connections [15]. Modeling tools may use these specifications to support combining the individual fragments, but in many cases the final decisions must be made manually by the application developers. For example, if multiple business components and databases exist, in general, a modeling tool cannot derive with certainty which component has to connect to which database.

3.3 Coordinate Management Plans by Choreographies

Similarly to connecting isolated topology fragments, their management plans need to be combined for realizing holistic management processes that affect larger parts of the merged application at once, for example, to terminate the whole application. However, as discussed in Sect. 2.3, manually merging workflow models is a highly non-trivial and technically error-prone task. Therefore, we propose using *interconnection choreographies* to coordinate the individual workflows without changing their actual implementation. Interconnection choreographies define interaction specifications for collaborating processes by interconnecting *communication activities*, i. e., *send* and *receive* activities, of these processes via set of *message links*[3]. This enables modeling different interaction styles between the individual management workflows, e. g., asynchronous and synchronous interactions. Thus, in this step, (i) application managers analyse required management processes, (ii) select appropriate management workflows of the individual topology models, and (iii) coordinate them by modeling choreographies. In addition, (iv) depending on required input and output parameters of the individual workflows, the data flow between the workflow invocations has to be specified. For example, the MySQL provisioning workflow of the motivating scenario outputs the endpoint and credentials of the database, which are required to invoke a management plan of the PHP model that connects the

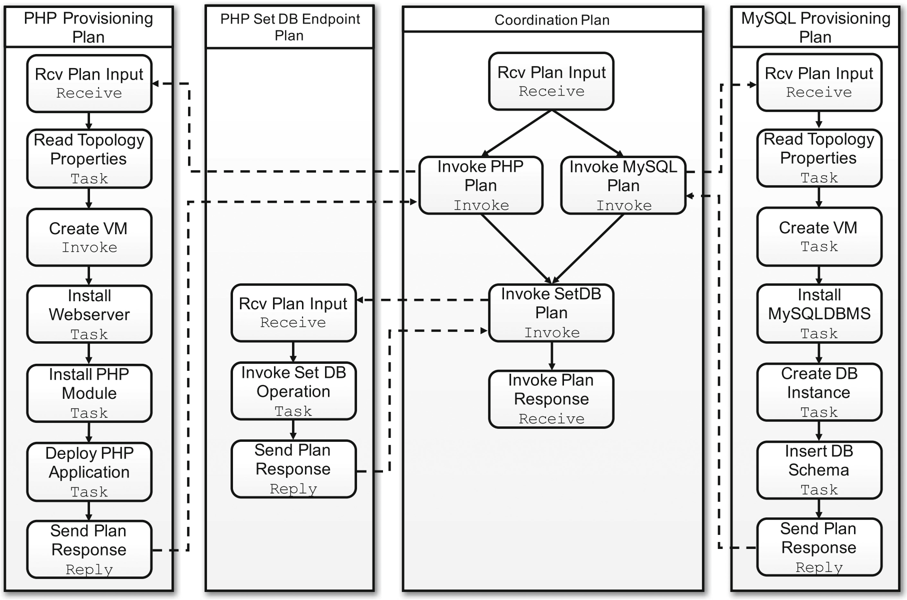

Fig. 4. Provisioning choreography coordinating management plans.

[3] In contrast to interaction choreographies that model message exchanges as abstract interactions not considering the workflow implementation.

PHP frontend to this database[4]. This is a manual step as the desired functionality, in general, cannot be derived automatically for application-specific tasks. For example, the individual provisioning plans of the motivating scenario can be used to model the overall provisioning of the entire application as well as to implement management plans that scale out parts of the application to handle changing workloads.

Figure 4 shows an example choreography that coordinates three management workflows from the motivating scenario. The coordination plan invokes in parallel the provisioning workflows of the PHP and MySQL topology models, respectively, by specifying message links to their receive activities. After their execution, messages are sent back to the coordination plan, which continues with invoking the aforementioned management workflow for transferring the database information (endpoint, database name, and credentials) to the PHP application by invoking the corresponding management operation.

3.4 Transform Choreographies into Executable Workflows

After manually modeling the choreography, the resulting model has to be transformed into an executable workflow. This has to be done as choreographies are not suited to be executed on a single workflow machine: unnecessary communication effort between the different workflows would slow down the execution time [30] and passing sensitive data over the wire, e. g., the database credentials, is not appropriate. Therefore, in this step, the choreography is automatically translated into an executable workflow model. This is described in detail in the next section and implemented by our prototype.

3.5 Deploy and Execute Resulting Workflows

In the last step, the generated workflow model is deployed on an appropriate workflow engine. Afterwards, the plan can be triggered by sending the start message to the workflow's endpoint. TOSCA runtimes such as OpenTOSCA [16] explicitly support management by executing such workflows.

4 Process Consolidation

To transform the management choreography into an executable workflow we provide a set of algorithms in Sect. 4.2 implementing the process consolidation approach described in [31] and [32]. Compared to the original paper, which is language-agnostic, the algorithms and the corresponding meta-model are focused on BPEL. This is because BPEL supports block-structured and graph-based

[4] Such management workflows can be realized in a generic manner by binding them exclusively to operations defined by the respective component type. TOSCA enables exchanging the implementations of these operations on the topology layer to implement application-specific management logic.

modeling [33]. Hence, the consolidation can be also applied to other block-structured languages such as BPMN. Moreover, BPEL has a well-defined operational semantics and it is still one of the most prominent executable workflow languages [34].

4.1 Choreography Meta-model

The algorithms base on the interconnection choreography meta-model that is defined in the following. The choreography meta-model uses the process meta-model introduced in [6] and the formalization introduced in [35] as foundation. For simplicity reasons BPEL constructs such as compensation handlers, event handlers, termination handlers or loops are omitted in this meta-model and left for future works.

Definition 1 (Process). *A process $P = (A, E, \mathcal{E}^f, \mathcal{H}, \mathcal{V}, Cond)$ is a directed acyclic graph where A is the set of activities within the process. $E \subseteq A \times A \times Cond$ is the set of directed control links between two activities. $\mathcal{E}^f$ denotes the set of fault events that may occur during the execution of a process. $\mathcal{H} \subseteq A_{scope} \times \{\perp, \mathcal{E}^f\} \times A$ denotes the set of hierarchy relations between a scope activity and its child activities. The nesting $\perp$ indicates that an activity is a direct child of a scope activity and the nesting $\mathcal{E}^f$ indicates that an activity is a fault handling activity of a scope. $Cond$ denotes the set of transition conditions and join conditions used within P that can be evaluated to* `true` *or* `false`*.*

The transition condition of a control link is evaluated if the execution of the source activity of the link (i) completed, (ii) faulted, or (iii) if it is affected by dead path elimination [6,36].

Definition 2 (Activity). *The operational semantics of an activity $a \in A$ is implied by its type which can be assigned to an activity with the function $\mathsf{type} : A \rightarrow \mathcal{T}$. The following sets of activities are distinguished:*

$$A = A_{invoke} \cup A_{receive} \cup A_{reply} \cup A_{task} \cup A_{empty} \cup A_{opaque} \cup A_{assign} \cup A_{scope}$$

An activity with incoming control links has a join condition referring to the states of its incoming control links. It can be assigned with the function $\mathsf{joinCond} : A \rightarrow Cond$. The condition must evaluate to `true` *to start the execution of the activity. If no join condition is defined explicitly, the activity execution is started when one of its in incoming control links is activated.*

Invoke activities A_{invoke} within a process send messages to another process over a message link. These messages can be received by receive activities $A_{receive}$. An invoke activity supports either the asynchronous or synchronous one-to-one interaction[5] pattern [37]. The asynchronous invoke activity sends a message to the receive within the other process in a "fire and forget" manner, i. e., after the

[5] In one-to-one interactions an invoke activity sends a message to exactly one receive activity, while in one-to-many interactions an invoke activity communicates with multiple receives, e. g., via loops.

message was sent the invoke completes and its successor activities are performed. A synchronous interaction is modeled at the sender side with an invoke activity, that waits until it receives a response from the called partner process before it completes, i. e., it "blocks" until the response is received. At the receiver's side a synchronous interaction is modeled with a receive activity that is followed in the control flow by a set of one or more reply activities A_{reply} sending the response back to the calling invoke. During runtime just one reply activity must be executed, hence, the reply activities for a single receive must reside on mutual exclusive branches in the control flow.

Task activities A_{task} implement the actual management logic, such as executing human tasks, calling scripts etc. This activity type is not part of BPEL specification but introduced here to indicate that a management operation is performed. Empty activities A_{empty} do not perform any business functions but act a synchronization point for control links. Assign activities A_{assign} perform data assignments, such as copying a value from one variable to another. As data flow is out of scope of this work, we will not further discuss how these assignments are performed in detail. Opaque activities A_{opaque} act as placeholder for other activities.

A scope activity $s \in A_{scope}$ defines an execution context for its direct and indirect child activities and defines a common fault handling behavior on them. Therefore, a scope activity defines a set of fault handlers. Each handler contains one fault handling activity to process the thrown fault. A fault handler reacts on a certain fault event $\mathit{fault} \in \mathcal{E}^f$ thrown by the direct or indirect child activities of s. A scope activity has exactly one standard fault handler that reacts on all faults not being caught by the other fault handlers. This standard fault handler can be defined with the function $\mathsf{catchAll} : \mathcal{E}^f \rightarrow \{\texttt{true}, \texttt{false}\}$. Scopes can be arbitrarily nested where the root scope is the process, i. e., a process is just a special type of scope. In contrast to scopes whose fault handlers may rethrow faults to their parent scopes, the process scope must not rethrow any faults.

The following further functions are used in the algorithms in Sect. 4.2. The function $\mathsf{incoming} : A \rightarrow \wp(E)$ returns the incoming control links and the function $\mathsf{outgoing} : A \rightarrow \wp(E)$ the outgoing control links of an activity a. The function $\mathsf{parentHR} : A \rightarrow \mathcal{H}$ returns the hierarchy relation between the given activity and its direct parent activity. To denote the projection to the i^{th} component of a tuple π_i is used.

Definition 3 (Choreography). *A choreography $C \in \mathcal{C}$ is defined by the tuple $C = (\mathcal{P}, \mathcal{ML})$, i. e., it consists of a set of interacting processes $\mathcal{P}$ and the message links $\mathcal{ML}$ between them. A message link ml connects a sending and a receiving activity: $\mathcal{ML} \subset A_{invoke} \cup A_{reply} \times A_{invoke} \cup A_{receive}$. $\forall ml \in \mathcal{ML} : P_1 \neq P_2 \wedge P_1, P_2 \in \mathcal{P}$ where $\pi_1(ml) \in \pi_1(P_1) \wedge \pi_2(ml) \in \pi_1(P_2)$. A message link is activated when the sending activity is started. A receiving activity cannot complete until its incoming message link was activated.*

In a choreography the processes interact just via message exchanges. Hence, activities originating form different processes are isolated form each other, i. e., state changes of activities within one process, such as faults, are not affecting activities originating from other processes directly.

4.2 Choreography-Based Process Consolidation

The process consolidation operation gets a choreography as input and returns a single process P_μ. The operation ensures that P_μ contains all activities A_{task} defined within the processes of C and that the execution order between these activities is preserved. P_μ is able to generate the same set of activity traces of A_{task} during runtime as C [31,32]. Since the consolidation was just described on a conceptual level, we provide a more formal description in the following. Algorithm 1 acts as entry point for the consolidation. It creates the consolidated process P_μ and calls the algorithms implementing the consolidation steps. Note that we only address the control flow aspects of the consolidation here. Data flow aspects are just briefly discussed.

Algorithm 1. Process Consolidation.

```
1: $P_\mu \leftarrow new\ Process$
2: $A_{P_\mu} \leftarrow \pi_1(P_\mu)$, $E_{P_\mu} \leftarrow \pi_2(P_\mu)$, $\mathcal{E}^f_{P_\mu} \leftarrow \pi_3(P_\mu)$
3: $\mathcal{H}_{P_\mu} \leftarrow \pi_4(P_\mu)$, $\mathcal{V}_{P_\mu} \leftarrow \pi_5(P_\mu)$, $Cond_{P_\mu} \leftarrow \pi_6(P_\mu)$
4: procedure CONSOLIDATE($C$)
5:     ADDPROCESSELEMENTS($C$)
6:     MATERIALIZECONTROLFLOW($C$)
7:     $A_s = \{a \in A_{P_\mu} \mid a \in A_{scope}\}$
8:     RESOLVEVIOLATIONS($A_s$)
9: end procedure
```

4.3 Adding Process Elements

Algorithm 2 adds the activities, control links, variables etc. being defined in the processes of choreography C to P_μ (line 3). The activities originating from different processes in C have to be isolated from each other in P_μ. This ensures that the original property of a choreography is preserved, that faults occurring in one process are not directly propagated to activities in another processes. The isolation is guaranteed by adding a scope s for each process P to be merged (lines 4 to 9). The attached fault handler catches and suppresses all faults that may be thrown from the activities within the scope.

Algorithm 2. Add elements of process to be merged to P_μ.

```
1: procedure ADDPROCESSELEMENTS($C$)
2:     for all $P \in \pi_1(C)$ do
3:         $A_{P_\mu} \leftarrow A_{P_\mu} \cup \pi_1(P), \ldots, Cond_{P_\mu} \leftarrow Cond_{P_\mu} \cup \pi_6(P)$
4:         $s \leftarrow new$ scope
5:         $fault \leftarrow new\ \mathcal{E}^f$
6:         catchAll($fault$) $\leftarrow$ true
7:         $a_{fh} \leftarrow new$ empty
8:         $\mathcal{H}_{P_\mu} \leftarrow \mathcal{H}_{P_\mu} \cup \{(s, fault, a_{fh}), (P_\mu, \perp, s)\}$
9:         $A_{P_\mu} \leftarrow A_{P_\mu} \cup \{s, a_{fh}\}$
10:    end for
11: end procedure
```

4.4 Control Flow Materialization

The control flow materialization shown in Algorithm 3 derives the control flow between activities originating from different processes from the interaction patterns defined in C. Here the materialization of asynchronous and synchronous one-to-one interactions is discussed. The materialization of one-to-many interactions is described in [32].

Algorithm 3. Control Flow Materialization.

```
1: procedure MaterializeControlFlow($C$)
2:     $\mathcal{ML}_{inv} = \{ml \in \pi_2(C) \mid \pi_1(ml) \in A_{invoke}\}$
3:     for all $ml_{inv} \in \mathcal{ML}_{inv}$ do
4:         $inv \leftarrow \pi_1(ml_{inv})$
5:         $rcv \leftarrow \pi_2(ml_{inv})$
6:         $ML_{rp} \leftarrow \{ml \in \pi_2(C) \mid \pi_1(ml) \in A_{reply}\}$
7:         if $ML_{rp} = \emptyset$ then
8:             MaterializeAsyn($inv, rcv$)
9:         else
10:            MaterializeSyn($inv, rcv, \pi_1(ML_{rp})$)
11:        end if
12:    end for
13: end procedure
```

To determine the interaction pattern Algorithm 3 checks in line 7 for each invoke activity if it is also a target of one or more message links ML_{rp} originating from reply activities. If this is the case the synchronous control flow materialization is called, otherwise the asynchronous materialization is called.

The materialization for asynchronous interactions is implemented by Algorithm 4. The algorithm replaces the invoke activity inv and receive activity rcv with the *synchronization activities* syn_{inv} and syn_{rcv}. Activity syn_{inv} serves as synchronization point for the control links of the former invoke activity inv. Thus, it inherits the control links and join condition of the invoke inv. The activity also emulates the message transfer by assigning the data that were transported in message before to the variable, where the message content was copied to by the receive activity. The new activity syn_{rcv} gets the control links and join condition of rcv assigned. This preserves the control flow order between the predecessor and successor activities of the former rcv. To emulate the control flow constraint implied by an asynchronous interaction, i. e., that successor activities of the former activity rcv are not started before the message was sent over the message link, a new control link $e_{inv2Rcv}$ is created between syn_{inv} and syn_{rcv}.

To perform the synchronous consolidation, beside invoke inv and receive rcv, also the set $A_{replyInv}$ of possible reply activities for inv is passed to Algorithm 5. An example for a synchronous interaction with two reply activities is depicted in Fig. 5. If a fault occurs during the execution of $b4$ reply activity $b5'$ within fault handler fh is executed. In the standard faultless flow $b5$ is performed.

Algorithm 4. Asynchronous Control Flow Materialization.

```
1: procedure MaterializeAsyn($inv, rcv$)
2:     $syn_{inv} \leftarrow new$ assign
3:     ReplaceActivity($inv$, $syn_{inv}$)
4:     $syn_{rcv} \leftarrow new$ empty
5:     ReplaceActivity($rcv$, $syn_{rcv}$)
6:     $e_{inv2Rcv} \leftarrow new$ link($syn_{inv}, syn_{rcv}$, true)
7:     joinCond($syn_{rcv}$) $\leftarrow$ joinCond($rcv$) AND $e_{inv2Rcv}$ = true
8: end procedure
9: procedure ReplaceActivity($oldAct, newAct$)
10:     $A_{P_\mu} \leftarrow (A_{P_\mu} \cup \{newAct\}) \setminus \{oldAct\}$
11:     $\forall e \in$ incoming($oldAct$) : $\pi_2(e) \leftarrow newAct$
12:     $\forall e \in$ outgoing($oldAct$) : $\pi_1(e) \leftarrow newAct$
13:     joinCond($newAct$) $\leftarrow$ joinCond($oldAct$)
14:     $hr_{parent} \leftarrow$ parentHR($oldAct$)
15:     $\mathcal{H}_{P_\mu} \leftarrow (\mathcal{H}_{P_\mu} \cup \{(\pi_1(hr_{parent}), \pi_2(hr_{parent}), newAct)\}) \setminus \{hr_{parent}\}$
16: end procedure
```

Algorithm 5. Synchronous Control Flow Materialization.

```
1: procedure MaterializeSyn($inv, rcv, A_{replyInv}$)
2:     $syn_{inv} \leftarrow new$ assign
3:     ReplaceActivity($inv$, $syn_{inv}$)
4:     $syn_{rcv} \leftarrow new$ empty
5:     ReplaceActivity($rcv$, $syn_{rcv}$)
6:     $e_{inv2Rcv} \leftarrow new$ link($syn_{inv}, syn_{rcv}$, true)
7:     $syn_{rcRp} \leftarrow new$ empty
8:     $\forall e \in$ outgoing($inv$) : $\pi_1(e) \leftarrow syn_{rcRp}$
9:     $e_{inv2RcRp} = new$ link($syn_{inv}, syn_{rcRp}$, true)
10:     for all $rp \in A_{reply}$ do
11:         $syn_{rp} \leftarrow new$ assign
12:         ReplaceActivity($rp$, $syn_{rp}$)
13:         $e_{rp2RcRp} \leftarrow new$ link($syn_{rp}, syn_{rcRp}$, true)
14:         $E_{P_\mu} \leftarrow E_{P_\mu} \cup \{e_{rp2RcRp}\}$
15:     end for
16:     joinCond($syn_{rcRp}$) $\leftarrow$ $e_{rp2RcRp_i}$ = true $OR \ldots OR$ $e_{rp2RcRp_n}$ = true    ▷ $(n = |A_{reply}|)$
17: end procedure
```

In a first step Algorithm 5 replaces *inv* and *rcv* with synchronization activities and creates a control link between these activities. This is implemented in the same way as for asynchronous interactions (lines 2 to 6). In the second step, the loop in line 10 replaces each reply activity *rp* being an origin of a message link targeting the former activity *inv* with another synchronization activity syn_{rp}. The created reply activities syn_{rp} assign the response data to the same variable, where the content of the reply message (transported over message link ml_{rp}) was copied to by activity *inv*. In the example in Fig. 5 two synchronization

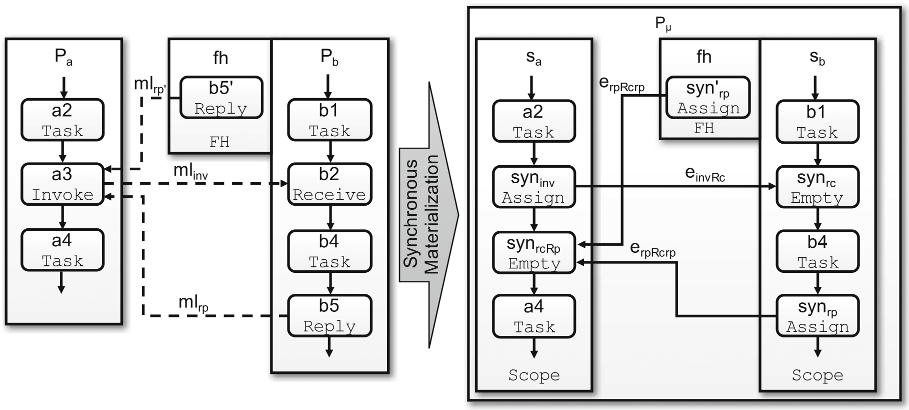

Fig. 5. Synchronous control-flow materialization.

activities syn_{rp} and syn'_{rp} are created from the two reply activities. To emulate the control flow constraint that the successor activities of a synchronous invoke are not started before the response message is sent from exactly one of the reply activities, a control link is created from each synchronization activity syn_{rp} to the new activity syn_{rcRp} (created in line 7). The join condition of syn_{rcRp} set in line 16 ensures that syn_{rcRp} is executed when one of the synchronization activities syn_{rp} completed.

4.5 Resolving Cross-Boundary Link Violations

Workflow languages supporting block-structured and graph-based modeling impose certain restrictions on control links crossing the boundaries of block-constructs such as loops, subprocesses, error handlers etc. For instance, in BPMN it is forbidden that the source activity of a control link lays outside of a BPMN subprocess while the target of this link is pointing to an activity inside the subprocess. In BPEL control links must not pass the boundary of loops, event handlers or compensation handlers. For fault handlers just control links pointing outside the handler are allowed. The control flow materialization, however, may create control links crossing the boundary of block-constructs if the sending or receiving activity, where control link is created from, is located in such a construct. In [38] we proposed algorithms for resolving these *cross-boundary violations* for links crossing BPMN and BPEL loops. Here an algorithm is provided for solving these violations for fault handlers.

An example scenario for a cross-boundary violation involving a fault handler is shown on the left side in Fig. 6. There activity a_{fhRoot} is the fault handling activity of fault handler fh. It contains the fault handling logic including synchronization activities (not depicted in Fig. 6). The control flow materialization created the invalid control link e_{cbl} pointing to activity a_{fhRoot} and causing a cross-boundary violation.

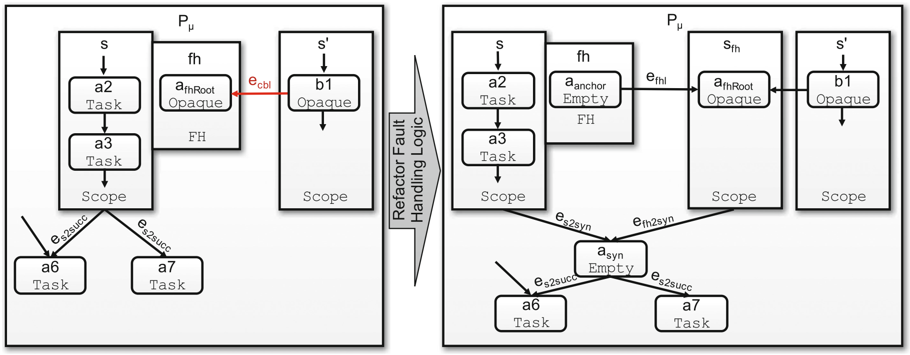

Fig. 6. Resolving fault handler cross-boundary violation by refactoring fault handling logic.

An informal discussion about resolving the violations for fault handlers was provided in [12]. Based on this discussion Algorithm 6 is introduced. The algorithm needs to resolve the violations while preserving the control flow semantics of BPEL scopes and fault handlers. Before the algorithm is introduced this semantics is briefly discussed.

If the execution of a scope completes successfully without throwing a fault, all of its fault handlers are uninstalled. However, if a scope throws a fault the normal processing within the scope stops. All control links originating from activities within the scope which were not activated before the faulted occurred are deactivated (dead path elimination). Then the fault handler catching the fault is installed, i. e., the fault handling activities within the fault handler are performed. All other fault handlers are uninstalled as well. After fault handling completed the control flow continues at the scope boundaries and the transition conditions of all control links originating from the scope are evaluated as usual. The deactivation of a fault handler implies that the root activity of the fault handler and also its child activities cannot be executed anymore or, spoken in terms of BPEL, are marked as *dead*. Consequently, all control links originating from within an uninstalled fault handler are deactivated.

Algorithm 6 performs two steps. In a first step procedure ResolveViolations checks for each of the provided scopes if there exist cross-boundary violations caused by control links pointing into one or more fault handlers. Therefore, all fault handlers of scope s are determined in line 6. In line 9 the actual violation check is performed. In the example in Fig. 6 only fault handler fh violates the cross-boundary constraint.

In the second step procedure RefactorFaulHandlerLogic resolves the violation by moving the fault handling logic a_{fhRoot} out of the fault handler into a new *fault handling scope* s_{fh}, such as the one depicted at the right side of Fig. 6. Thereby, for each violated fault handler a separate fault handling scope s_{fh} is created. All fault handlers not being affected by a cross-boundary violation remain unchanged. To ensure that the fault handling logic a_{fhRoot} is executed if the

Algorithm 6. Removing Control Links Pointing into Fault Handlers.

```
1: procedure ResolveViolations($A_s$)
2:     for all $s \in A_s$ do
3:         $hr_{parentS} \leftarrow \mathsf{parentHR}(s)$
4:         $a_{parentS} \leftarrow \pi_1(hr_{parent})$
5:         $a_{syn} \leftarrow \varnothing$
6:         $FH = \{hr \in \mathcal{H}_{P_\mu} \mid s = \pi_1(hr) \wedge \pi_2(hr) \in \mathcal{E}^f\}$
7:         for all $fh \in FH$ do
8:             $A_{FH} = \pi_3(FH)$
9:             if $\exists e \in E : \pi_1(e) \notin A_{FH} \wedge \pi_2(e) \in A_{FH}$ then
10:                if $a_{syn} = \varnothing$ then
11:                    $a_{syn} \leftarrow new$ empty
12:                    $\mathcal{H}_{P_\mu} \leftarrow \mathcal{H}_{P_\mu} \cup \{(a_{parentS}, \perp, a_{syn})\}$
13:                    $e_{s2syn} \leftarrow new$ link($s, a_{syn}$, true)
14:                    $\forall e \in \mathsf{outgoing}(s) : \pi_1(e) \leftarrow a_{syn}$
15:                end if
16:                RefactorFaultHandlerLogic($s, fh, a_{parentS}, a_{syn}$)
17:            end if
18:        end for
19:    end for
20: end procedure
21: procedure RefactorFaulHandlerLogic($s, fh, a_{parentS}, a_{syn}$)
22:    $s_{fh} \leftarrow new$ scope
23:    $fault \leftarrow \pi_2(fh)$
24:    $a_{fhRoot} \leftarrow \pi_3(fh)$
25:    $a_{anchor} \leftarrow new$ empty
26:    $e_{fhl} \leftarrow new$ link($a_{anchor}, s_{fh}$, true)
27:    $e_{fh2syn} \leftarrow new$ link($s_{fh}, a_{syn}$, true)
28:    $E_{P_\mu} \leftarrow E_{P_\mu} \cup \{e_{fhl}, e_{fh2syn}\}$
29:    $\mathcal{H}_{P_\mu} \leftarrow (\mathcal{H}_{P_\mu} \cup \{(a_{parentS}, \perp, s_{fh}), (s_{fh}, \perp, a_{fhRoot}), (s, fault, a_{anchor})\}) \setminus (s, fault, a_{fhRoot})$
30:    $\mathsf{joinCond}(a_{syn}) \leftarrow \mathsf{joinCond}(a_{syn})\ OR\ e_{efh2syn} =$ true
31: end procedure
```

corresponding fault is caught, a new control link e_{fhl} is created in line 26. It connects the newly created empty activity a_{anchor} (line 25) within the fault handler with the fault handling logic a_{fhRoot}. The additional activity a_{syn} created in line 11 becomes the source of the outgoing control links of scope s (line 14) to ensure that the successor activities of s are not started before either s or its fault handling scopes completed. Activity a_{syn} is just created once per scope s. To preserve the property that control links originating from scope boundaries are always performed no matter whether the scope s completed successfully or not, a_{syn} becomes the target for the set of control links originating from each created fault handling scope. The join condition of a_{syn} created in line 30 ensures that a_{syn} and its successors (i. e., the former successors of s) are performed if one these links is activated.

The resulting example process P_μ is depicted at the right side of Fig. 6. Moving the fault handling logic out of the fault handlers of scope s resolves the violations but also preserves the original control flow. If scope s completes successfully, all of its fault handlers are uninstalled and the dead path elimination marks the activities within the fault handlers as dead. Hence, also activity a_{anchor} is marked as dead and its successor activity a_{fhRoot}. The the successors of scope s can be executed as link e_{s2syn} is activated.

If a fault occurs during the execution of scope s the execution of s is interrupted and the matching fault handler is installed. If the matching fault handler was not changed by Algorithm 6 its fault handling logic is directly performed. After the completion of the fault handling logic link e_{s2syn} is activated and thus also the successors of s. In case a fault handler is called whose fault handling logic was moved out of the fault handler, activity a_{anchor} is executed. As a_{anchor} is an empty activity, it completes directly and its outgoing control link causes the execution of the actual fault handling logic a_{fhRoot}. Since the BPEL control link semantics requires that all incoming control links of an activity are evaluated, it is guaranteed that the former successor activities (in the example $a6$ and $a7$) of scope s are not started before either s or its fault handling logic completed.

4.6 Consolidation Example

The single process *LAMP Provisioning Plan* shown in Fig. 7 results from the application of Algorithm 1 on the provisioning choreography. As all plans interact

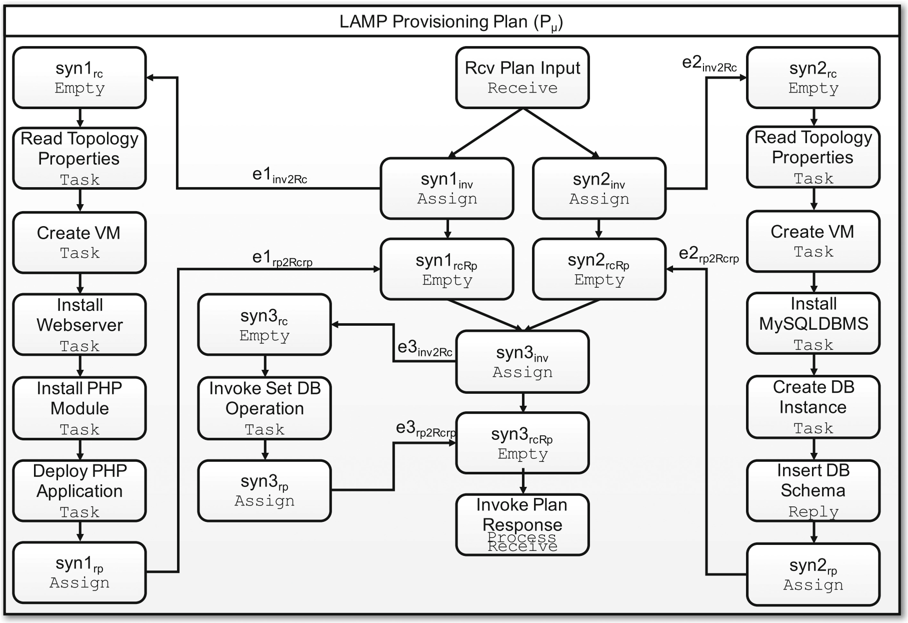

Fig. 7. Consolidation of a provisioning choreography into single LAMP provisioning plan.

synchronously only the synchronous control flow materialization is applied. Thus, the sending and receiving activities related to each message link are replaced with synchronization activities as described in Sect. 4.4.

5 Validation

In this section, we validate the practical feasibility of the presented method by a prototypical implementation. We applied the method and merge algorithms to the choreography modeling language BPEL4Chor and extended the OpenTOSCA Cloud management ecosystem to support choreographies. This ecosystem consists of (i) the graphical TOSCA modelling tool *Winery* [17], the (ii) *OpenTOSCA container* [16], and (iii) the self-service portal *Vinothek* [18]. An overview of the entire prototype is shown in Fig. 8: Application developers use Winery to merge existing topology models, while application managers use the choreography modelling tool ChorDesigner [39] to coordinate the associated management workflows.

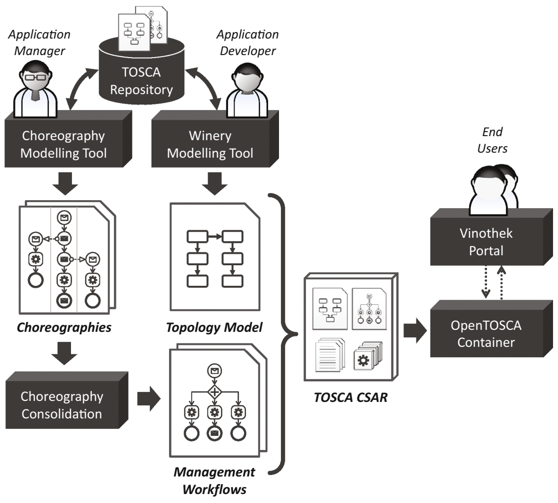

Fig. 8. Architecture of the open-source Cloud management prototype.

Based on the merge algorithm described in Sect. 4 a process consolidation tool was developed for generating a single executable BPEL processes out of a choreography[6]. Therefore the algorithm was extended to accommodate the language idiosyncrasies of BPEL. This includes the emulation of the choreography's data flow in the merged process and the elimination of cross-boundary violations. Beside asynchronous and synchronous one-to-one interactions the tool does also support the consolidation of one-to-many interactions [32,37].

[6] The prototype is available as Open-source: https://github.com/wagnerse/chormerge.

The merged topology model as well as the generated management plans can be packaged as CSAR using Winery. The resulting CSAR can be installed on the OpenTOSCA container, which internally deploys the workflows and, thereby, makes them executable. To ease the invocation of provisioning and management workflows, we employ our TOSCA self-service portal Vinothek, which wraps the invocation of workflows by a simple user interface for end users. All tools are available as open-source implementations, thus, the developed prototype provides an end-to-end Cloud application management system supporting choreographies for modelling coordinated management processes.

6 Conclusion and Future Work

In this work, we proposed a method for developing new TOSCA-based applications in a more efficient way by reusing topologies and management plans from existing applications. This method describes the steps to be performed to combine existing application topology artifacts or parts thereof into a single topology and how the plans for managing these artifacts can be coordinated by BPEL4Chor-based management choreographies. To provide the choreography again as plan for reusing it in other topologies and for the efficient execution of the choreography on a single workflow engine, this method encompasses a step to consolidate the choreography into a single management plan. Therefore, we introduced a set of new algorithms describing the consolidation of interacting BPEL processes in a formal way. To validate the method different tools of the OpenTOSCA ecosystem were used. Each of these tools enables one or more steps of the proposed method to be performed in a semi-automatic manner.

As BPEL4Chor has the same modeling capabilities as BPMN collaborations [40] the proposed algorithms can be also applied on BPMN collaborations. In the near future also the OpenTOSCA ecosystem will be extended to model and enact BPMN-based management plans and collaborations. Failures during the execution of a management plan may need already completed tasks in other plans of the same choreography to be compensated, i. e., the effects of these tasks must be undone. Therefore, BPEL and BPMN offer compensation constructs to implement this behavior. To support the consolidation of plans interacting via compensation constructs, we plan to extend process consolidation approach accordingly. To provide a comprehensive set of reusable management plans, we also plan to transform low-level management scripts into plans.

Acknowledgements. This work was partially funded by the BMWi projects SmartOrchestra (01MD16001F), NEMAR (03ET40188), and by the DFG project SitOPT (610872).

References

1. Brown, A.B., Patterson, D.A.: To Err is Human. In: EASY, p. 5 (2001)
2. Oppenheimer, D., Ganapathi, A., Patterson, D.A.: Why do internet services fail, and what can be done about it? In: USITS (2003)

3. Opscode, Inc.: Chef official site (2015). http://www.opscode.com/chef
4. Puppet Labs Inc.: Puppet official site (2015). http://puppetlabs.com/puppet/what-is-puppet
5. Coutermarsh, M.: Heroku Cookbook. Packt Publishing Ltd., Birmingham (2014)
6. Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques. Prentice Hall PTR, New Jersey (2000)
7. Keller, A., Badonnel, R.: Automating the provisioning of application services with the BPEL4WS workflow language. In: DSOM (2004)
8. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: BPMN4TOSCA: a domain-specific language to model management plans for composite applications. In: BPMN (2012)
9. Herry, H., Anderson, P., Wickler, G.: Automated planning for configuration changes. In: The Past, Present, and Future of System Administration, Proceedings of the 25th Large Installation System Administration Conference, LISA, USENIX Association (2011)
10. Breitenbücher, U., Binz, T., Kopp, O., Leymann, F., Wettinger, J.: Integrated cloud application provisioning: interconnecting service-centric and script-centric management technologies. In: CoopIS (2013)
11. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F., Weiß, A.: Improve resource-sharing through functionality-preserving merge of cloud application topologies. In: CLOSER. SciTePress (2013)
12. Wagner, S., Kopp, O., Leymann, F.: Consolidation of interacting bpel process models with fault handlers. In: Proceedings of the 5th Central-European Workshop on Services and their Composition (ZEUS), CEUR (2013)
13. Decker, G., Kopp, O., Leymann, F., Weske, M.: BPEL4Chor: Extending BPEL for modeling choreographies. In: ICWS. IEEE (2007)
14. OASIS: TOSCA v1.0 (2013). http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html
15. OASIS: TOSCA Primer v1.0 (2013). http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.html
16. Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., Wagner, S.: OpenTOSCA – a runtime for TOSCA-based cloud applications. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 692–695. Springer, Heidelberg (2013). doi:10.1007/978-3-642-45005-1_62
17. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: Winery – a modeling tool for TOSCA-based cloud applications. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 700–704. Springer, Heidelberg (2013). doi:10.1007/978-3-642-45005-1_64
18. Breitenbücher, U., et al.: Vinothek - a self-service portal for TOSCA. In: Proceedings of the 6nd Central-European Workshop on Services and their Composition (ZEUS), CEUR (2014)
19. Breitenbücher, U., et al.: Combining declarative and imperative cloud application provisioning based on TOSCA. In: IC2E (2014)
20. Breitenbücher, U., et al.: Vino4TOSCA: a visual notation for application topologies based on TOSCA. In: CoopIS (2012)
21. OASIS: Web Services Business Process Execution Language (WS-BPEL) Version 2.0. OASIS (2007)
22. OMG: Business Process Model and Notation (BPMN), Version 2.0 (2011)
23. Hofreiter, B., Huemer, C.: A model-driven top-down approach to inter-organizational systems: from global choreography models to executable BPEL. In: CEC (2008)

24. Mendling, J., Hafner, M.: From WS-CDL choreography to BPEL process orchestration. J. Enterp. Inf. Manage. **21**, 525–542 (2008)
25. Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y., Barreto, C.: Web Services Choreography Description Language Version 1.0 (2005)
26. Küster, J., Gerth, C., Förster, A., Engels, G.: A tool for process merging in business-driven development. In: Proceedings of the Forum at the CAiSE (2008)
27. Mendling, J., Simon, C.: Business process design by view integration. In: Eder, J., Dustdar, S. (eds.) BPM 2006. LNCS, vol. 4103, pp. 55–64. Springer, Heidelberg (2006). doi:10.1007/11837862_7
28. Herry, H., Anderson, P., Rovatsos, M.: Choreographing configuration changes. In: Proceedings of the 9th International Conference on Network and Service Management, CNSM (2013)
29. Soldani, J., Binz, T., Breitenbücher, U., Leymann, F., Brogi, A.: TOSCA-MART: a method for adapting and reusing cloud applications. Technical report, University of Pisa (2015)
30. Wagner, S., Roller, D., Kopp, O., Unger, T., Leymann, F.: Performance optimizations for interacting business processes. In: IC2E. IEEE (2013)
31. Wagner, S., Kopp, O., Leymann, F.: Towards verification of process merge patterns with Allen's Interval Algebra. In: Proceedings of the 4th Central-European Workshop on Services and their Composition (ZEUS). CEUR (2012)
32. Wagner, S., Kopp, O., Leymann, F.: Choreography-based consolidation of multi-instance BPEL processes. In: Proceedings of the 4th International Conference on Cloud Computing and Service Science (CLOSER). SciTePress (2014)
33. Kopp, O., Martin, D., Wutke, D., Leymann, F.: The difference between graph-based and block-structured business process modelling languages. Enterp. Model. Inf. Syst. **4**, 3–13 (2009)
34. Leymann, F.: BPEL vs. BPMN 2.0: should you care? In: BPMN (2010)
35. Kopp, O., Mietzner, R., Leymann, F.: Abstract syntax of WS-BPEL 2.0. Technical report computer science 2008/06, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany (2008)
36. OASIS: Web Services Business Process Execution Language Version 2.0 - OASIS Standard (2007)
37. Barros, A., Dumas, M., Hofstede, A.H.M.: Service interaction patterns. In: Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 302–318. Springer, Heidelberg (2005). doi:10.1007/11538394_20
38. Wagner, S., Kopp, O., Leymann, F.: Choreography-based consolidation of interacting processes having activity-based loops. In: Proceedings of the 5th International Conference on Cloud Computing and Service Science (CLOSER). SciTePress (2015)
39. Weiß, A., Karastoyanova, D.: Enabling coupled multi-scale, multi-field experiments through choreographies of data-driven scientific simulations. Computing **98**, 439–467 (2014)
40. Kopp, O., Leymann, F., Wagner, S.: Modeling choreographies: BPMN 2.0 versus BPEL-based approaches. In: EMISA (2011)