FREIE UNIVERSITÄT
BERLIN

Flexible Workflows to Support Transactional Service Composition in Mobile Environments

Katharina Hahn, Heinz Schweppe

B-09-17
November 2009

FACHBEREICH MATHEMATIK UND INFORMATIK
SERIE B • INFORMATIK

Flexible Workflows to Support Transactional Service Composition in Mobile Environments

Katharina Hahn, Heinz Schweppe
Institute for Computer Science
Freie Universität Berlin
Email: {khahn, schweppe}@mi.fu-berlin.de

***Abstract*—Service oriented computing provides suitable means to technically support distributed collaboration of heterogeneous devices, for example those present in mobile environments. E.g., many applications are built on composite Web-Services. However, when executing these applications in dynamic environments, failures of participating entities have to be optimistically coped with, in order to avoid inconsistent system states and thereby provide suitable correctness guarantees. Transactional coordination for services so far lacks the possibility to adapt failure handling to the current execution context, e.g. dynamically bound services at runtime. In this paper, we employ transactional service properties to ensure reliable, i.e., *correct* execution of workflows by still respecting the autonomy of participants. We propose algorithms to verifiy and alter the structure of the composition at runtime, thus adapting the control flow to the current execution context to ensure correct execution.**

I. Introduction

When supporting collaboration in mobile networks, one has to provides suitable means to deal with the characteristics of these environments. Besides the *decreased availability* of network links and resources, this particularly refers to the *dynamics* of these networks (due to the mobility and wireless communication channels) as well as the *increased autonomy* of mobile participants as opposed to conventional fixed-wired scenarios.

Service oriented architectures provide suitable means to *loosely* coupling of components. Through composition of components, new value-added services as well as applications can be created. SOAs allow for dynamic discovery and binding of services at runtime. This forms the basis for service composition in mobile environments in which the execution context (i.e., available services) is not known at designtime and might differ from execution to execution.

Current workflow engines only allow for *static deployment*: Once deployed, the composition, i.e., its workflow, remains fixed. There already exist approaches to flexibly alter workflows at runtime according to the current execution context, e.g. the bound service providers. Stein et al. [18] provide several execution strategies in different environments to maximize the profit and minimize the costs of the workflow. Additionally, there exist approaches, which provide technical means to dynamically alter e.g. deployed BPEL processes at runtime, such as [20].

Especially due to less stability of network links in mobile environments, it is indispensable to be able to cope with failures of different kinds to guarantee *correct execution*. Coping with failures while respecting the autonomy of participants might come at the cost of relaxing correctness criteria, such as the strict atomicity and isolation of all components, which are guaranteed in databases. These strict guarantees can only be asserted by blocking of resources. However, in case of disconnections of participants, this leads to services blocking for an arbitrary long time. In order to avoid blocking, "correctness" is relaxed and e.g. assured by employing convenient *forward-* and *backward-recovery* mechanisms to avoid inconsistent and non-recoverable system states.

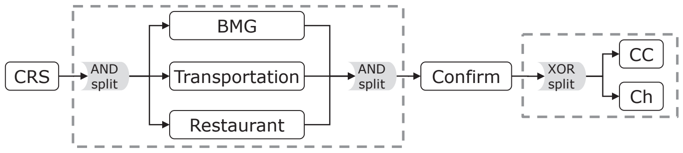

Fig. 1. Workflow of the Mobile Planet MoP.

We will recurrently refer to the following application scenario of mobile travel guide, which we call **Mo**bile **P**lanet (MoP). MoP supports the user in finding points of interest and booking activities according to defined preferences and present offers. An example of a mobile ticket vendor who sells tickets to the Blue Man Group (BMG, http://www.blueman.com) show to tourists is shown in Figure 1. Additionally, transportation facilities organized and a table at a nearby restaurant is reserved.

At first, a customer is asked to specify its requests (*CRS*), e.g. number of traveling persons and dinner preferences. The workflow then books tickets (BMG), organizes convenient transportation and reserves a table at a restaurant in parallel (intended by the AND-split). The threads are then synchronized, and if all are completed, the customer agrees on the legal terms and the booked tickets are delivered (*Confirm*). Finally, the payment is performed, either by credit card (*CC*) or by cash (*Ch*). Note that this time, as intended by the XOR-split, one and only one branch is supposed to complete. So far, the workflow specifies the execution of services, but it does not yet define the behavior in the presence of any service failure.

In our scenario, we assume nodes to be equipped with one or several wireless communication interfaces (e.g., IEEE 802.11, GSM or UMTS), which all feature quite different characteristics in terms of bandwidth, communication range and costs. Nodes may be reachable via more than one networking channel at a time. We target applications which allow for delay tolerant networking. It is thus favorable to use adhoc communication, if the costs of communicating (which explosively increase, if roaming costs are involved) exceed the *value* of the application.

In this paper, we propose a correctness criterion (semi-atomicity) for workflows with respect to transactional properties of services. The contribution of the paper is an adaptive workflow management system (AWM) which *verifies* the correctness of specified workflows at runtimes and *adapts* the structure of a workflow if the verification fails.

The paper is structured as follows: In Section II, we present related work. In Section III the formal model of transactional service composition is introduced. In Section IV, we at first present our verification algorithm (IV-A). We then state, how we adapt workflows at runtime (IV-B) in order to ensure correctness. In conclude in Section V by discussing experimental results of our approach.

II. Related Work

Many advanced transaction models (ATM), e.g. [6], [22] have been proposed which support transactional processing in distributed databases [10]. These use less strict notions of atomicity and isolation in order to avoid blocking situations. Although they are very powerful, they are not capable of integrating structural requirements of complex applications in one transaction. A variety of mobile transaction models (e.g., [7], [13], [14]) have been proposed, which are able to cope with failures due to frequent disconnections. However, they are not able to integrate different structural patterns as well.

In SOAs, workflow execution and transactional coordination (e.g., for Web-Services (WS)) are two orthogonal concerns: The execution of workflows is controlled by workflow engines. Those provide support for the design, execution, visualization and analysis of workflows however do not integrate transactional guarantees. Transactional coordination, e.g. for WS, is specified by the WS-Transaction Framework WS-Tx[1] which offers means for coordination of different services. It specifies coordination types for short- (atomic transactions: WS-AT) and long-running activities (business activities: WS-BA). WS-AT relies on blocking of resources to achieve atomic outcome, while WS-BA relies on the assumption that all components can be compensated in case of failure. Workflow languages, e.g., BPEL, additionally provide compensation and failure handlers which are used to specify custom-build failure handling. As those enable designers to specify recovery for single failure situations, no correctness guarantees can be given.

Forward-recovery for composite services, such as proposed by [17], is a promising approach to deal with the unstable availability of single participants in service oriented computing. The authors propose the use of an abstract service provider. Its responsibility is to dynamically replace a failing service at runtime with semantically equivalent ones. Thus, specific failure situations are covered. But transactional execution of the whole workflow or subparts of it, is not considered.

[1]http://www.ibm.com/developerworks/library/specification/ws-tx

Fauvet et. al [3] propose a high level operator for composing Web-Services according to transactional properties. Transactional execution relies on the tentative hold protocol (THP). Services are distinguished according to their additional capabilities: Support of 2PC, compensatability or neither. While this approach is powerful it uses a proprietary operator. Vidyasankar et. al [21] explore transactional properties in their proposed multi-level composition model. It provides a user-centric approach to describe desired transactional properties at different levels of abstraction. While we employ transactional properties at different levels of abstraction as well, as opposed to those two approaches ([3], [21]), we integrate transactional composition in existing standards.

More recently, Stein et al. [18] proposed flexible provisioning of Web-Services to maximize the benefit of workflows and reduce costs. They provide several provisioning strategies regarding utility costs and failure probabilities to achieve an optimized composition. We complement this work by providing means to ensure transactionally *correct* execution of flexible workflows, even in case of failure.

In order to verify the execution of Web-Service workflows, several formalisms have been used, such as petri nets [9] or finite-state-machines [1]. These introduce powerful means to formally verify the execution of composite Web-Services but do not consider transactional verification. Gaaloul et al. [4] use an event based-approach to model transactional composite services. They provide static verification mechanisms, however adaptation of the workflow in case the verification fails, is left to the designer. As it provides suitable means to specify transactional behavior, we use this formalism in our work. We additionally specify dynamic workflow alterations at runtime to support correct execution.

Binding services at runtime is integrated in existing workflow languages, e.g. in BPEL using *dynamic bindings using lookups* as partner links and can be performed by workflow engines. However, in mobile environments, discovery and matching of services at runtime is a challenging task. Many powerful approaches to service discovery in mobile networks exist, such as [2], [15], [16], as well as semantic description languages, e.g., OWL-S, WSDL-S or DSD [11], [12], [23] to support matching of service offers and requests.

III. Transactional Composition Formally

In this section, we briefly introduce the formal model used to specify transactional behavior of services and composite services. As it is very powerful and suits our demands, our model is based on the model proposed by Gaaloul et al. [4]. We introduce transactional properties of services and derive those for the used workflow pattern. At the end of this section, we introduce the correctness criterion which we employ for workflows.[2]

[2]As we focus on the verification and adaptation algorithms (Section IV-A), the description of the model is kept short. For a more detailed introduction, we refer to [8].

A. Service Model

The behavior of a single service is modeled as a state-machine (see Figure 2), using the following states: *Initial* indicates that it has not been activated yet, after activation its status is *active*. *Failed* and *canceled* indicate failed execution (i.e., no changes are made persistent), due to an internal error or externally triggered. If the service completed successfully, its state is *completed*. The transitions between these states are either *internally* triggered, i.e., by the service itself, or *externally* triggered, i.e., by another entity, such as a workflow engine, another service or a person.

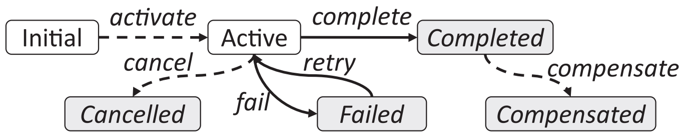

Fig. 2. State machine of a service.

B. Transactional Service Properties

We explore transactional properties of services in order to ensure correct execution in terms of transactional support. Transactional properties have already been considered in the context of flexible transactions [24] and are also important for verification of failure handling as specified by the designer [4]. We classify services according to whether they can fail (*redoability*), whether can be compensated (*compensatability*) and whether they need to be recovered in case of failure (*consistent closure*). Examples of the latter are *read-only* registered transaction participants.

A service is considered to be **compensatable** (denoted as boolean values $S.comp$ and $\neg S.comp$ respectively), if its effects can be undone after completion (e.g., by invoking a compensating service C).[3] The redoability of a service indicates whether a service can fail. I.e., a **redoable** service (denoted as $S.redo$) will eventually complete, if its activation is repeated a positive number of n times. We indicate the *need for recovery* in case of failure by the term, whether a service demands **consistent closure** wrt the outcome of the whole workflow. A service demanding consistent closure ($S.consCompl$) needs, once completed, recovery in case of failure. A service, which does not demand consistent closure does not need to be compensated in case of recovery.

For verification and adaptation purposes (see Section IV), it is important to know whether a service is redoable and whether its completion hinders the correctness of the workflow in case of failure. Thus, it is important to know, whether a service is compensatable *or* does not need consistent closure. We denote this by the derived transactional property referred to as **recoverability** as follows: A service is considered to be recoverable (denoted as $S.recover$), if it is either compensatable or it does not demand consistent closure:

$$S.recover = S.comp \vee \neg S.consCompl$$

[3]Non-compensatable services are sometimes referred to as pivot services.

In this paper, we focus on verification and adaptation (see Section IV). For this purpose it suffices to consider recoverability and redoability of services.[4] The transactional properties of a workflow element e are thus defined as follows:

Definition 1 (Transactional Properties of Element e): The transactional properties of a workflow element e are denoted as a tuple as follows:

$$P_T(e) = (e.recover, e.redo)$$

Consider again the example workflow and the transactional services properties as shown in Figure 3. *BMG* for example with $P_T(BMG) = (1, 0)$ is recoverable (i.e., bought tickets may be returned) but not redoable (if no more tickets are available).

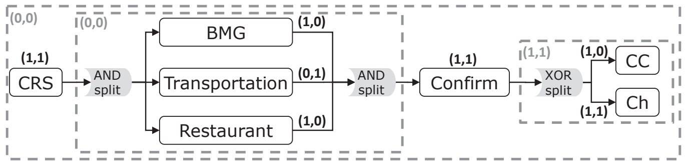

Fig. 3. Running example with transactional properties.

C. Transactional Composition Using Workflow Patterns

The structure of service compositions is defined using workflow patterns. Formally, a workflow pattern is a function which given a set of services returns their control flow [19]. Components of such patterns are either patterns themselves or services. In this paper, we limit our consideration to the following basic workflow patterns:

Sequence WP_{SEQ} $(S_i, \dots, S_{i+k})$: If services S_i, S_{i+1}, $\dots S_{i+k}$ arranged in a sequence,[5] a subsequent service is enabled after the completion of its predecessor. The pattern is also referred to as *sequential* or *serial routing*. In order to successfully complete a sequence, all components have to complete, therefore it can be expressed by the boolean formula $(S_i \wedge S_{i+1} \wedge \dots \wedge S_{i+k})$.

AND-split, AND-join WP_{AND} $(S_i, \dots, S_{i+k})$: At the split point, the execution of the composition splits in concurrent branches S_i to S_{i+k}. Each of them is executed independently of the others. At the join point, the subsequent workflow is continued as soon as *all* branches completed $(S_i \wedge \dots \wedge S_{i+k})$. This pattern also referred to *parallel split* or *fork*.

XOR-split, XOR-join[6] WP_{XOR} (S_i, S_j): At the split point, the workflow diverges into several branches, out of which one is chosen. As these are never executed in parallel, the subsequent workflow is continued as soon as one branch completes. Thus, one and only one branch completes. Successful

[4]Note, that for adding the appropriate failure handling mechanisms, *compensatability* and *consistent closure* are needed as well.

[5]The elements are ordered according to their index.

[6]As OR-split and -join can be expressed by the XOR-pattern, we do not seperately regard it.

execution of an XOR, containing two elements S_i and S_j is denoted by $(S_i \wedge \neg S_j) \vee (\neg S_i \wedge S_j)$.

In our work, we consider elements of XOR-patterns to be alternatives. Thus, the choice on which service within the pattern to execute can be based on their transactional properties.

As we want to verify the workflow with the bound services at runtime, we analyze the composite service and derive the *transactional properties* of *patterns*. Those are determined according to the pattern and the properties of the enclosed elements. The properties of a workflow pattern $WP(E)$ are denoted just as those of services, namely $WP(E).recover$ and $WP(E).redo$ (see Definition 1). As the semantics of the SEQUENCE- and AND-pattern are the same (i.e., all included services have to complete in order to complete the pattern), their transactional properties are derived in the same way:

1) Transactional Properties of the SEQUENCE- and AND-pattern: A pattern $WP_T(E)$, which is a SEQUENCE- or an AND-pattern, are derived as follows: The pattern is recoverable, if and only if all elements of the pattern are *recoverable*:

$$WP_T(E).recover \Leftrightarrow \forall e \in E : e.recover$$

The pattern, is *redoable*, if and only if all of its elements are redoable. Then the execution of the whole pattern is, once invoked, guaranteed to complete:

$$WP_T(E).redo \Leftrightarrow \forall e \in E : e.redo$$

2) Transactional Properties of the XOR-pattern: The properties of an XOR-pattern are determined differently than those of the SEQUENCE and AND-pattern, as one and only one branch is to be completed. As it cannot be previously stated which service will complete at runtime, the transactional properties can only be determined for some (not *all*) cases.

The XOR-pattern $WP_{XOR}(E)$, is *recoverable*, if all its elements are recoverable. If all elements are not recoverable, then the pattern is not recoverable:

$$WP\ _{XOR}(E).recover \Leftarrow \forall e \in E : e.recover$$
$$\neg WP_{XOR}(E).recover \Leftarrow \forall e \in E : \neg e.recover$$

Otherwise, if some elements are recoverable and some are not, the recoverability cannot be previously determined.

However it can definitely be determined, whether the pattern is redoable: If one redoable element within WP_{XOR} exists, then this element is an execution path of the pattern which is guaranteed to complete. Thus, an XOR-pattern $WP(E)_{XOR}$ is redoable, as soon as one element within the pattern is redoable. Otherwise, it is not redoable:

$$WP_{XOR}(E).redo \Leftrightarrow \exists e \in E : e.redo$$

Reconsider the example with the transactional properties as shown in Figure 3. According to these derivation rules, the AND-pattern is not recoverable and not redoable ($P_T(WP_{AND}(BMG, T, R)) = (0, 0)$). The XOR-pattern on the other hand is recoverable and redoable ($P_T(WP_{XOR}(CC, Ch)) = (1, 1)$).

3) Subtransaction Pattern: Additionally to the just mentioned control flow patterns, we employ an auxiliary pattern, called SUBTRANSACTION-pattern $WP_{subTA}(e)$. This is used, whenever autonomous execution of elements endangers the correctness of the execution. $WP_{subTA}(e)$ contains a single control flow pattern. It defines, that all elements contained by this pattern have to be coordinated in an atomic subtransaction, e.g., using a blocking protocol.[7] This can be implemented using WS-AT. The WP_{subTA} pattern decreases the autonomy of enclosed services. However, it is in some cases indispensable in order to guarantee correct execution. $WP_{subTA}(e)$ is neither recoverable, nor retrieable i.e., its transactional properties are: $p_T(WP_{subTA}(e)) = (0, 0)$.

D. Data Dependencies

In addition to the control flow, we consider given data-dependencies between elements. A data dependency as:

Definition 2 (Data-dependency: $S_i \rightarrow S_j$): A data-dependency of the form $S_i \rightarrow S_j$ exists, if S_j is dependent on the output of S_i. Thus, S_j cannot be executed before S_i.

Clearly, data-dependencies are transitive: If dependencies $S_i \rightarrow S_j$ and $S_j \rightarrow S_k$ exist, then S_k is *transitive dependent* on S_i, thus it cannot be executed before S_i. Initially, we assume that no data dependencies within parallel elements of AND-patterns and generally no cyclic data dependencies exist.

E. Semi-Atomicity in the Presence of Transactional Properties

Employing the transactional properties of services (see Section III-B), we now define the notion of *correct* execution. Intuitively, the execution of the workflow is correct, if the workflow is completed (cf. commit). Additionally, correct execution involves situations, in which the workflow is abored (i.e., unsuccessfully terminates). In these situations, all services, which demand compensation, need to be compensated.

This notion of correctness (commit or abort) seems to be closely related to atomicity of database transactions. These employ blocking protocols, such as the 2PC (two phase commit) to reach atomic outome. However, blocking of resources is counterproductive in the environment of loosely coupled components. Several notions of relaxed atomicity have been introduced in the past to increase the autonomy of participants: E.g., *semantic atomicity* [5] and *semi-atomicity* [24] allow for commitment of subtransactions at different times. Appropriate compensation mechanisms enable backward-recovery of already committed subtransactions in case of failure.

When deploying Web-Services, transactional support is specified by the WS-Transaction Framework WS-Tx. Services can either be executed as an atomic unit (WS-AtomicTransaction) which employs the 2PC to reach atomic commit. As mentioned before, we aim at avoiding blocking of resources whenever possible. The other possibility is to group

[7]Solitary exception are *indirect conflict elements*, see Section IV-A.

services into activities (WS-BusinessActivity) which assume compensatability of all components. If this assumption does not hold for only one services within such an activity, severe failures might occur, as we will discuss in Section V.

As stated in [8], we explore transactional service properties to specifiy semi-atomicity. We employ the accepted termination states model to identify representationsl sets of services whose completion reflects the successful execution of the composite service. If alternatives i.e., WP_{XOR} patterns, exist, multiple of these sets exist. We thus define semi-atomicity of an executed composite service to be, that

- either all services belonging to one execution path leading to an ATS, are completed and no other service which demands consistent completion is completed (commit) or
- no service demanding consistent completion is completed (abort).

As recovery for services which do not demand consistent completion is disregarded, this relaxes the semi-atomicity as defined for flexible transactions. Note, that semi-atomicity specifies the correctness for a termination of a workflow i.e., an *executed* workflow.

IV. Adaptive Workflow Management

Using semi-atomicity as the correctness criterion, we now define our algorithm to ensure *transactionally correct* execution of a workflow ω. We therefore perform two basic steps (see Figure 4): At first, a given workflow ω *is verified* in the current execution context by considering all transactional properties of all involved entities. If the verification fails, the control flow structure of the workflow is *adapted* to ensure correct execution in a second step. We execute the verification and adaptation every time, flexible service providers are bound at runtime. For both steps, verification and adaptation, we employ different representations of the workflow, which are introduced in the respective sections.

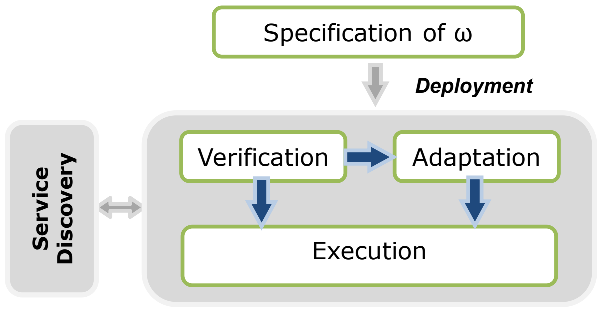

Fig. 4. System architecture of transactional workflow management system.

A. Verification

For verification purposes, we convert a given workflow ω to its tree representation (denoted as T_ω). Simply for clarity, inner nodes are round, leaf nodes are rectangular shaped. Inner nodes represent the patterns of the workflow, leaf nodes represent the included services. The all-embracing pattern of the workflow is the root node of the tree.

Note, the children of the SEQUENCE-node have to be ordered according to their position in the sequence. For all other patterns, the order of child nodes is irrelevant. In Figure 5, the tree representation of the running example is depicted.

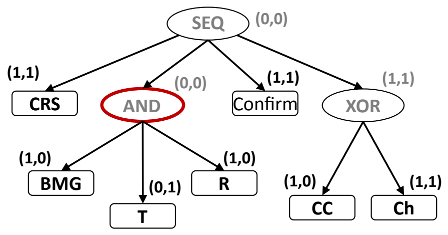

Fig. 5. Tree representation of the running example.

Intuitively, a workflow is guaranteed to semi-atomically terminate, if in case of failure of any service,

- the execution of the workflow can be either backward-recovered (thus all previously completed services are recovered)
- or there exists at least one alternate execution path to an ATS which is guaranteed to complete i.e., it is redoable.

To formalize this, we define the property *semi-atomicity preservation* (SAP) of workflow elements. It denotes, whether the execution of an element will in any case result in semi-atomic commit or abort. A workflow is correct, i.e., SAP, if *all* possible executions result in semi-atomic termination. In the following, we define elements, whose execution may hinder semi-atomic termination of a workflow (*conflict elements*). We will then use these definitions to formalize the verification criterion SAP for workflow elements (in Section IV-A2).

1) Conflict Elements: In this section, we examine pairs of elements, which hinder the correct execution of a worklow ω. Intuitively, those are pairs of elements, which may cause failure situations, which cannot be healed by backward- or forward-recovery. The semi-atomicity of an executed workflow is not preserved, if an element e which is not recoverable (i.e., $p_T(e) = (0, *)$) is completed, while another element e', which is not retrieable (i.e., $p_T(e') = (*, 0)$) failed (and no alternative exists). Those are defined as *transactional conflict elements*.

Definition 3 (Transactional Conflict $\{e_i, e_j\}_C$): A *transactional conflict* between workflow elements e_i and e_j (denoted as $\{e_i, e_j\}_C$) exists, if e_i is not recoverable and e_j is not redoable, i.e.:

$$p_T(e_i) = (0, *) \text{ and}$$
$$p_T(e_j) = (*, 0).$$

If such a pair of elements exists within a workflow, they cannot be executed independently of each other. Recall for example, the transportation service T (Figure 3) which is not recoverable but redoable i.e., $p_T(T) = (0, 1)$. *BMG* on the other hand is recoverable but not redoable i.e., $p_T(BMG) = (1, 0)$. $\{T, BMG\}_C$ is then a conflicting pair of elements. If they are executed concurrently, T might complete while BMG fails. This termination is not semi-atomic i.e., not correct. Rearranging them to $WP_{SEQ}(BMG, T)$ however, will in any case result in semi-atomic termination. Rearranging elements in any order is only possible, if no data-dependency exists. We therefore additionally regard *directed transactional conflicts*.

Definition 4 (Directed Transactional Conflict $(e_i, e_j)_C$): A *directed transactional conflict* between workflow elements e_i and e_j (denoted as $(e_i, e_j)_C$) exists, if the transactional conflict $\{e_i, e_j\}_C$ and a (direct or indirect) data dependency e_i to e_j exist. I.e.:
$\{e_i, e_j\}_C$ and
$e_i \rightarrow e_j$ exist.

Directed transactional conflicts between pairs of elements cannot be solved by rearranging them. Their order remains fixed according to the data-dependency. In this case, semi-atomicity can only be preserved by utilizing a subtransaction pattern WP_{subTA}, altering the workflow to be $WP_{subTA}(WP_{SEQ}(e_i, e_j))$. This results in decreased autonomy of these elements.

Considering the correlation between directed transactional conflicts, one easily identifies, that they are *transitive* in the following way: If directed transactional conflicts $(e_1, e_2)_C$ and $(e_2, e_3)_C$ exist, it concludes from Definition 4 that $p_T(e_2) = (0, 0)$ and $(e_1, e_3)_C$ exists as well (see Figure 6, left side).

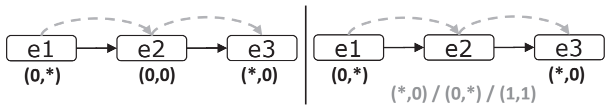

Fig. 6. Transitivity of conflicts (left) and enclosed conflict elements (right).

On the other hand, if the conflict $(e_1, e_3)_C$ exists and data-dependencies $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ (see Figure 6, right side), it again follows from Definitions 3 and 4, that

- e_2 conflicts with e_1 (i.e., $(e_1, e_2)_C$ exists), if e_2 is non-redoable,
- e_2 conflicts with e_3, if it is non-recoverable (i.e., $(e_2, e_3)_C$ exists) or
- if e_2 is recoverable and redoable (i.e., $p_T(e_2) = (1, 1)$) it neither conflicts with e_1 nor e_3.

In the last case, we refer to e_2 as an *indirect conflict element*. For the sake of simplicity, we forego an explicit notation of indirect conflict elements: Thus, as those are surrounded by at least one directed transactional conflict, they are included in subtransactions as well. However, as they offer full flexibility, they do not need to be coordinated.

2) Correctness of Workflow Elements: We will now employ these definitions and observations to derive conditions under which workflow elements are correct, i.e. SAP. Recall that a workflow element is SAP, if *all* possible executions result in semi-atomic termination. The execution of one *single service* is thus always SAP as all introduced termination states are semi-atomic. When regarding *workflow patterns*, it depends on the type of the pattern and the included elements, whether it is SAP. In the following, regarding the tree representation, we consider solely direct child nodes of pattern nodes.

*Proposition 1 (*SAP *of a Sequence WP_{SEQ}):* A sequence pattern $WP_{SEQ}(E)$ is SAP
$\iff$
a. all of its elements $e \in E$ are SAP and
b. no transactional conflicts $\{e_i, e_j\}_C$, with $e_i, e_j \in E$ and $i < j$ exist

Proof:

- '$\Longrightarrow$': Assume $WP_{SEQ}(E)$ to be SAP. Claim: All $e \in E$ are SAP and no transactional conflict $\{e_i, e_j\}_C$, with $e_i, e_j \in E$ and $i < j$ exists. Proof by contradiction:
  - Assume it exists $e \in E$ which is not SAP. Thus, execution of e can result in non semi-atomic termination of e and thus of $WP_{SEQ}(E)$. This contradicts the assumption.
  - Assume it exists $\{e_k, e_l\}_C$ with $e_k, e_l \in E$ and $k < l$. If e_k ($p_T(e_k) = (0, *)$) completes and e_l ($p_T(e_l) = (*, 0)$) fails, the execution is not semi-atomic, thus $WP_{SEQ}(E)$ is not SAP. This contradicts the assumption.
- '$\Longleftarrow$': Assume all elements $e \in E$ to be SAP and no transactional conflict $\{e_i, e_j\}_C$, with $e_i, e_j \in E$ and $i < j$ to exist. Claim: $WP_{SEQ}(E)$ is SAP. Proof by contradiction:
  Assume $WP_{SEQ}(E)$ is not SAP. Then,
  - it either exists an element $e \in E$ which is not SAP and whose execution may cause $WP_{SEQ}(E)$ not to be SAP,
  - or it exists $e_l \in E$ which is not retrieable ($p_T(e_l) = (*, 0)$) and $e_k \in E$ which is not recoverable ($p_T(e_k) = (0, *)$), such that in case of failure of e_l the previous completion of e_k prevents SAP termination. Thus, $l < k$. Then $\{e_k, e_l\}_C$ is transactional conflict.

  Both cases contradict the assumption.

■

Recall that elements of a WP_{SEQ} are ordered in increasing order of their index. The propositions uses transactional conflicts (i.e., not directed) as services which are aligned in sequence do not necessarily need to be data-dependent on their predecessors. As opposed to sequences, no order of elements of a WP_{AND} is given. Thus, a WP_{AND} is SAP in the following case:

*Proposition 2 (*SAP *of an WP_{AND}):* An AND pattern $WP_{AND}(E)$ is SAP

$\Longleftrightarrow$

a. all of its elements $e \in E$ are SAP and
b. no transactional conflicts $\{e_i, e_j\}_C$, with $e_i, e_j \in E$ exist

Proof: Analogous to proof of Proposition 1.

- '$\Longrightarrow$': Assume $WP_{AND}(E)$ to be SAP. Claim: All $e \in E$ are SAP and no transactional conflict $\{e_i, e_j\}_C$, with $e_i, e_j \in E$ exists. Proof by contradiction:
  - Assume it exists $e \in E$ which is not SAP. Thus, execution of e can result in non semi-atomic termination of e and thus of $WP_{AND}(E)$. This contradicts the assumption.
  - Assume it exists $\{e_k, e_l\}_C$ with $e_k, e_l \in E$. If e_k ($p_T(e_k) = (0, *)$) completes and e_l ($p_T(e_l) = (*, 0)$) fails, the execution is not semi-atomic, thus $WP_{AND}(E)$ is not SAP. This contradicts the assumption.
- '$\Longleftarrow$': Assume all elements $e \in E$ to be SAP and no transactional conflict $\{e_i, e_j\}_C$, with $e_i, e_j \in E$ to exist. Claim: $WP_{AND}(E)$ is SAP. Proof by contradiction: Assume $WP_{AND}(E)$ is not SAP. Then,
  - it either exists an element $e \in E$ which is not SAP and whose execution may cause $WP_{AND}(E)$ not to be SAP,
  - or it exists $e_l \in E$ which is not retrieable ($p_T(e_l) = (*, 0)$) and $e_k \in E$ which is not recoverable ($p_T(e_k) = (0, *)$), such that in case of failure of e_l the parallel completion of e_k prevents SAP termination. Then $\{e_k, e_l\}_C$ is transactional conflict.

  Both cases contradict the assumption.

■

As no data-dependencies between elements of a WP_{AND} exist, rearranging recoverable elements before non-recoverable conflicting elements and redoable elements after those is used to avoid conflicts.

According to its definition, only one branch of an WP_{XOR} pattern is executed. Therefore it follows, that an WP_{XOR} pattern is SAP if and only if *all* of its elements are SAP.

3) Verification Algorithm: Utilizing the just stated propositions, we are now able to define our verification algorithm of a workflow. It takes the tree representation T_ω of a workflow ω as input and returns *true*, if it is correct, i.e. SAP and *false*, if it is not SAP. T_ω is traversed bottom-up:

```
//traverse T_w bottom-up
for (every node n in T_w) {

   //verify the node:
   if (node n is !SAP)
      return false;
   else
      if (node n is a pattern)
            derive p_T(n);
}
return true;
```

The algorithm inspects whether all children of ω are SAP, and if so, whether the enclosing pattern is SAP. Since a workflow ω is always composed by workflow patterns, Propositions 1 and 2 guarantee the correctness of the algorithm.

Recall our running example of MoP: Its tree representation is depicted in Figure 5. The verification algorithm traverses the tree bottom up. As all leaf nodes are services, all of them are per definition SAP. After that, the WP_{AND} node (labeled *AND*) is inspected. Proposition 2 states, that it is SAP, if no transactional conflict exists. However, T transactionally conflicts with both *BMG* and R, thus transactional conflicts $\{T, BMG\}_C$ and $\{T, R\}_C$ exist. Therefore, this node and thus the whole workflow is not SAP.

B. Adaptation

If the verification fails, we adapt the structure of a workflow to ensure correct execution (recall Figure 4). The verification is performed for every non-verified WP_{XOR} node and for the whole workflow, if it could not be validated. In the following, for the sake of simplicity, we refer to the algorithm being executed on a workflow ω, returning an adapted workflow ω'.

For adaptation purposes, we consider the *data-dependency graph* $D_\omega(V, E)$ of a workflow ω. $D_\omega(V, E)$ is an acyclic directed graph. Its set of vertices V contains all mandatory elements of ω, thus all WP_{XOR} nodes of T_ω and all services s of ω which are not element of an WP_{XOR}. A directed edge (v_i, v_j) between nodes v_i and v_j exists, if there exists a direct data-dependency in the form $v_i \to v_j$. In Figure 7, the $D_\omega(V, E)$ of the running example is depicted.

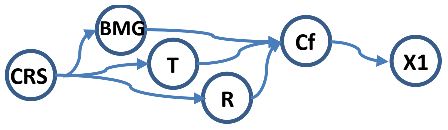

Fig. 7. $D_\omega(V, E)$ of the running example.

1) Minimal Set of Coordinated Elements: Using SAP as the correctness criterion, a workflow ω is correct, if no failure situation may occur, which cannot be recovered. However, those may be avoided by including the whole workflow ω in a blocking subtransaction pattern WP_{subTA}. As mentioned before, this limits the autonomy (and *loosely* coupling) of involved entities. Thus, we spare the usage of WP_{subTA} patterns if possible. The following theorem states, which elements of a workflow *have* to be coordinated in order to ensure SAP.

Theorem 1 (Minimal Set M *of Coordinated Elements):* Let E_{CP} be the set of directed transactionally conflict elements of the workflow ω i.e.,

$$E_{CP} := \{e \mid \exists\, e' \text{ such that } (e, e')_C \text{ or } (e', e)_C\}.$$

Let further be E_Z the set of elements of ω, which are not recoverable and not redoable, i.e.

$$E_Z := \{e \mid p_T(e) = (0,0)\}.$$

M is defined as follows:

$$M := \begin{cases} \emptyset & \text{if } E_{CP} = \emptyset \text{ and } |E_Z| \leq 1 \\ E_{CP} \cup E_Z & \text{otherwise} \end{cases}$$

Then, the following holds:

a. If M is coordinated using a WP_{subTA} pattern i.e., WP_{subTA} $(WP(M))$, SAP of the workflow can be ensured.
b. If SAP of the workflow is ensured by coordinating a set of elements M' i.e., WP_{subTA} $(WP(M'))$, then M' is a superset of M, i.e. $M' \supseteq M$.

Proof: is straightforward however tedious using Propositions 1 and 2.

- a. Assume: M is coordinated. Claim: SAP of the workflow can be ensured. Proof by contradiction:
  Assume, there exists an element $e_i \notin M$, such that execution of the tuple of elements e_i, e_j hinders SAP. e_i, e_j are then either aligned in sequence or in parallel. Thus, one of the following is true:
  - $\{e_i, e_j\}_C \in WP_{SEQ}$ (e_i, e_j), with $i < j$. Since $e_i \notin M$: No data dependency $e_i \rightarrow e_j$ exists and at most one of them exposes $p_T(e) = (0,0)$. Otherwise, e_i were element of M. Thus, they can be rearranged in WP_{SEQ} (e_j, e_i) which is then SAP. This contradicts the assumption. (Existence of conflict $\{e_j, e_i\}_C \in WP_{SEQ}$ (e_j, e_i), with $j < i$ results in alignment WP_{SEQ} (e_j, e_i) accordingly.)
  - $\{e_i, e_j\}_C \in WP_{AND}$ (e_i, e_j). Since $e_i \notin M$: No data dependency $e_i \rightarrow e_j$ exists and at most one of them exposes $p_T(e) = (0,0)$. Otherwise, e_i were element of M. Rearranging the pattern to be WP_{SEQ} (e_j, e_i) (or WP_{SEQ} (e_i, e_i)) ensures SAP. This contradicts the assumption.
- b. Assume: SAP of ω is ensured by coordinating M' (i.e., WP_{subTA} $(WP(M'))$). Claim: M' is a superset of M (i.e., $M' \supseteq M$). Proof by contradiction:
  Assume, $\exists\ e \in M$, which is not coordinated (i.e., $e \notin M'$). Thus, one of the following holds:
  - If $e \in E_{CP} \Rightarrow \exists\ e' \in M$ with $(e, e')_C$ (or $(e', e)_C$). As a data dependency between those two exists, they have to be aligned in sequence WP_{SEQ} (e, e') (or WP_{SEQ} (e', e)). According to Proposition 1, this sequence is not SAP. Rearranging is not possible due to the data dependency. Thus, ω is not SAP. This contradicts the assumption.
  - Else if, $e \in E_Z$, then (according to the definition of M),
    * another element $e' \in E_Z$ exists. e and e' then form transactional conflicts $\{e, e'\}_C$ and $\{e', e\}_C$. Therefore, neither aligning them in sequence (see Proposition 1), nor in parallel (see Proposition 2) ensures SAP.
    * or a directed transactional conflict $(e_i, e_j)_C \in E_{CP}$ exists (recall $p_T(e_i) = (0,*)$ and $p_T(e_j) = (*,0)$). $\{e_i, e\}_C$, $\{e, e_j\}_C$ (and $\{e_i, e_j\}_C$) then form transactional conflicts. No sequential alignment of e, e_i and e_j ensures SAP according to Proposition 1, as any alignment regarding the data dependency $e_i \rightarrow e_j$, still contains transactional conflicts. Proposition 2 states, that no parallel alignment of e, e_i and e_j ensures SAP either.

  This contradicts the assumption.

■

Theorem 1 illustrates, that M is the minimal set of elements which needs to be coordinated in order to ensure SAP. Therefore, if this set of elements is enclosed by a WP_{subTA} pattern, an alignment of all other elements can be found, such that the workflow is correct. Again, please note, that for simplicity reasons *indirect conflict elements* are not explicitly labeled and thus not excluded from the WP_{subTA} pattern. However, as they offer full flexibility, they do not have to be coordinated in the subtransaction.

Using this theorem, we are able to show that our algorithm produces optimal results with respect to the size of WP_{subTA} patterns.

2) Adaptation Algorithm: We now define our algorithm which adapts a given input workflow ω to an output workflow ω' which ensures SAP. Our algorithm proceeds by traversing the data dependency graph $G_\omega(V, E)$. Since edges represent the existing data dependencies between elements, those are topologically processed by passing through $G_\omega(V, E)$. While traversing G_ω, we *append* elements to our output data structure ω'. Intuitively, recoverable (non-conflicting) elements are aligned before conflicting elements. Those are only followed by (non-conflicting) redoable elements. The algorithm operates in 4 steps, we start off by *initializing* the following variables:

- V_{CP} is the set of all directed transactional conflict elements:
  $V_{CP} := \{e \mid \exists\ e'$, with $e, e' \in V$ such that $(e, e')_C$ or $(e', e)_C\}$,
- V_Z is the set of all non recoverable, non redoable elements:
  $V_Z := \{e \mid e \in V,\ p_T(e) = (0,0)\}$,
- V_I is the set of all indirect conflict elements,
- V_M is the union of the previous sets as follows:
  $$V_M := \begin{cases} \emptyset & \text{if } V_{CP} = \emptyset \text{ and } |V_Z| \leq 1 \\ V_{CP} \cup V_Z \cup V_I & \text{otherwise} \end{cases},$$
- C denotes the set of all *current nodes* of V i.e., all $v \in V$ which do not have an incoming edge.
- Initialize output workflow ω'.

We start traversing $G_\omega(V, E)$ by processing non-conflicting recoverable elements. Propositions 1 and 2 both state, that a pattern is correct, if no transactional conflicts exist. Therefore, recoverable nodes may be aligned in sequence or parallel without causing transactional conflicts among them. This ensures that the output of this step is SAP. If more that one current, recoverable node exists, they are aligned in parallel as no data-dependency exists among them. Otherwise, they are aligned in sequence. This is formalized as follows:

```
while (C contains elements {r}
              with p_T(r) = (1,*)) {
   if (|{r}| > 1) {
      append WP_AND({r}) to w';
   } else {
      append r to w';
   }
   update G_w, update C;
}
```

By updating $G_\omega(V, E)$, already processed nodes and their respective outgoing edges are deleted. Thereby, the set of current nodes C is also altered.

This loop is executed until the set of current nodes does not contain any recoverable nodes. Therefore, if the set of conflict nodes V_M is not empty ($V_M \neq \emptyset$), the set of current nodes C *then* at least contains one element m, which is in the set of nodes that needs to be coordinated i.e., $m \in V_M$.[8] As only recoverable elements are appended to the output workflow ω' so far, ω' is recoverable i.e., $p_T(\omega') = (1, ?)$.

In the next step, elements which need to be coordinated, are processed. As stated above, if V_M is non-empty, there exists at least one element $m \in V_M$ which is also a current node, that is $m \in C$. In Theorem 1, we proved that all elements in V_M need to be coordinated in order to ensure SAP of the workflow. Thus, in this step, all elements $v \in V_M$ are appended to ω'. Again, if no data dependency exists, they are aligned in parallel, otherwise, they are appended according to the data-dependencies. If only one element v with $p_T(v) = (0, 0)$ exists (thus $v \in V_Z$) and no other conflicting elements exist i.e., $V_M = \emptyset$, v does not need to be coordinated. It is aligned behind recoverable and before retrieable elements. This is formalized as follows:

```
if (V_M != {}){
    M := {}
    while (C contains elements {m},
                    with {m} elem V_M) {
        if (|{m}| > 1) {
            append WP_AND({m}) to M;
        } else {
            append m to M;
        }
        update G_w, update C;
    }
    append WP_subTA(M) to w';
} else if (|V_Z| == 1) {
    append v elem V_Z to w';
    update G_w, update C;
}
```

Due to the transitivity of conflicts and the properties of enclosed conflict elements (recall Figure 6), all elements which lie on a path in G_ω from one conflict node m_i to any another m_j are in V_M. Either, they are conflicting elements themselves, or indirect conflict elements. Therefore, as soon as the set of current nodes does not contain any $m \in V_M$, V_M is completely processed. Thus, the subtransaction is closed.

In this step, we appended a WP_{subTA} pattern (or only one element $v \in V_Z$) to a recoverable workflow. According to Proposition 1, the resulting workflow ω' is still SAP.

By now – if existent – all conflict elements are processed and appended to ω'. Therefore, only retrieable elements are left to process. If there were a non-retrieable element $v \in C$, v would transactionally conflict with the WP_{subTA} pattern, thus v were a conflicting element and would have been included in V_M. Retrieable elements are appended in the same manner as recoverable elements at the beginning:

```
while (C contains elements {r}) {
    if (|{r}| > 1) {
        append WP_AND({r}) to w';
    } else {
        append r to w';
    }
    update G_w, update C;
}
return w';
```

Just as with recoverable elements, retrieable are arranged in parallel, if no data dependencies exist. Otherwise, they are topologically sorted regarding their data-dependencies. As no transactional conflicts exists among retrieable elements, the alignment of those at the end of ω' - and thus the output of the algorithm ω' - is SAP. The algorithm terminates, if the set of current nodes is empty: All nodes of $G_\omega(V, E)$ have been processed and added to ω'.

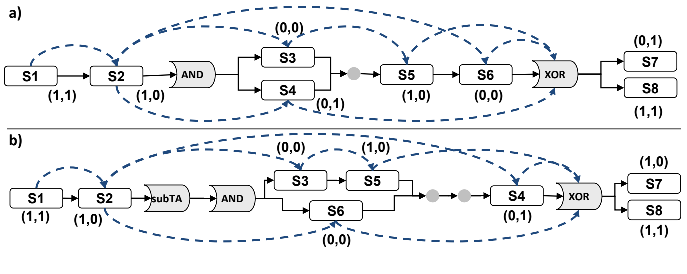

Fig. 8. Workflow as initially designed ω (a) and adapted ω' (b).

Consider for example the abstract workflow ω depicted in Figure 8.a. It consists of eight services (S_1 through S_8) with the given transactional properties and a set of seven data-dependencies (depicted as dashed lines). The verification for ω fails: Among others, the transactional conflict $\{S_3, S_4\}_C$ exists and according to Proposition 2, ω is thus not SAP.

The according data-dependency graph of ω is shown in Figure 9. Note that, since $WP_{XOR}(S7, S8)$ is correct, the adaptation is performed for the whole workflow and $WP_{XOR}(S7, S8)$ is generalized as node X_1 in the graph.

By analyzing the workflow, the conflict elements are detected: $(S_3, S_5)_C$ directly transactionally conflict, additionally $S_6 \in V_Z$. Thus $V_M = \{S_3, S_5, S_6\}$. Conflicting nodes

[8]This holds, as otherwise there would exist at least one non-recoverable, non-conflicting node n in C, from which a path (that is a data-dependency) to an element $m \in V_M$ existed. As n is non-recoverable it would then transactionally conflict with $m \in V_M$.

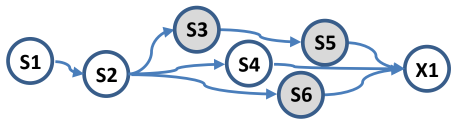

Fig. 9. Data-dependency graph of the example in Figure 8.

$(v \in V_M)$ are shaded in gray in Figure 9. The algorithm starts by traversing recoverable elements and appending those to the output workflow ω' (ω' is depicted in Figure 8.b). The set of current nodes C initially contains S_1. Thus, regarding the data-dependency, $WP_{SEQ}(S_1, S_2)$ is appended to ω' in the first step of the algorithm. After that, the set of current nodes C contains S_3, S_4 and S_6, none of which is recoverable. Therefore, all conflict nodes $v \in V_M$ are then enclosed in a subtransaction $(WP_{subTA}(WP_{AND}(WP_{SEQ}(S_3, S_5), S_6)))$ and appended to ω'. The set of current nodes C then contains S_4, which is redoable. Thus, $WP_{SEQ}(S_4, X_1)$ is appended to ω'.

Taking a closer look at the proposed adaptation algorithm, it is easy to see that, if existent, all $v \in V_M$ are coordinated using a WP_{subTA} pattern. As stated before, we abandon to coordinate indirect conflict elements $v' \in V_I$. Due to their full flexibility, they do not endanger correct execution, even if not coordinated in a subtransaction. As $V_M \setminus V_I$ corresponds to the minimal set M of coordinated elements (see Theorem 1), our algorithm produces optimal results with respect to the number of coordinated elements.

V. Discussion

In this section, we discuss our approach to support transactional service composition. We implemented the Adaptive Workflow Management framework (AWM) as a proxy of the Apache ODE engine. The implementation employs Java Web-Services; transactional properties are assembled via WS-Policies.

We want to briefly present comparisons of AWM to existing techniques which guarantee transactional correctness of composite services. Comparing AWM to solely coordinating all services, e.g. using WS-AT, our approach reduces the number of blocked resources if possible. It only blocks as many resources as WS-AT in the worst case (i.e., if all services are conflicting elements $\in V_M$). Considering Theorem 1 (minimal set of coordinated elements), our algorithm outputs optimal results with respect to the number of coordinated elements.

Transactional support of long-running activities is conventionally implemented using business activities (e.g., WS-BA). Those assume the compensatability of *all* involved entities. However, if this assumption does not hold for one (or more) services, one risks incorrect (non-recoverable) system states. Incorrect termination is e.g. inevitable if a non compensatable service commits while another one fails.

Figure 10 shows the probability that a workflow will semi-atomically terminate (*SA-probability* p_{SA}) when being executed using WS-BA and AWM. The higher p_{SA}, the more likely the workflow will correctly terminate. The results in

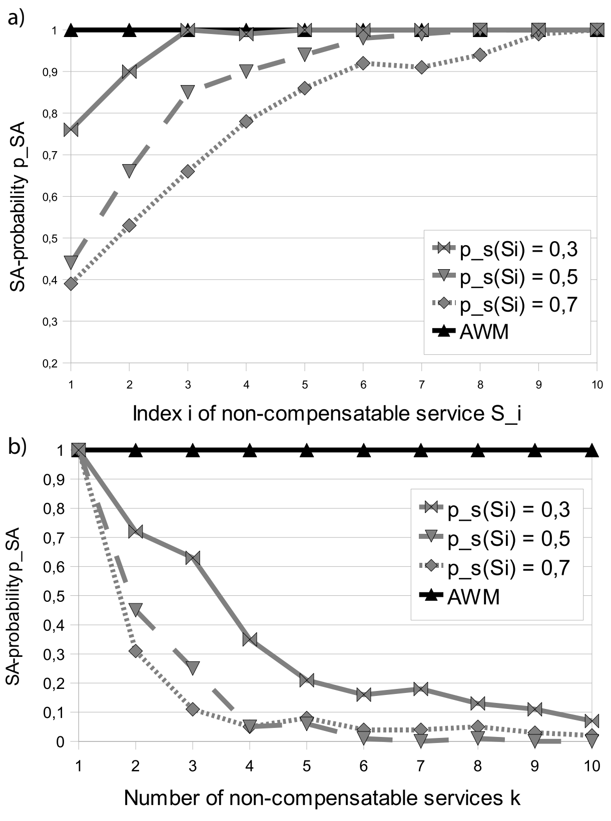

Fig. 10. SA-probability of $WP_{SEQ}(S_1, \ldots, S_{10})$ (a) $[WP_{AND}(S_1, \ldots, S_{10})$(b)] varying the index i of one [number k of] non-compensatable service[s].

Figure 10 both depict average values of 100 executions. In Figure 10.a we included *one* non-compensatable service in a sequence $WP_{SEQ}(S_1, \ldots, S_{10})$. We varied the index i of the non compensatable service (on the x-axis). Additionally, we varied the probability, that individual services will successfully complete (which we refer to as *success probability* denoted as p_s). Those were normally distributed around $0,3$ (gray solid line), $0,5$ (gray dashed line) and $0,7$ (gray fine dashed line) with a standard deviation of 5%. As our algorithm guarantees semi-atomic termination, the *SA-probability* p_{SA} for our approach AWM (black solid line) is 1 for all indices i independent of the success probability. For the rerefence approach WS-BA, it can be seen, if only one non-compensatable services is included, p_{SA} is remarkably decreased (down to 40% for $p_s(Si) = 0,7$ and $i = 1$). The experiment shows, the earlier the non-compensatable service is aligned (thus, the lower i), the lower is the resulting *SA-probability*, thus the higher the risk of inconsistent system states. Additionally, in this setup, the higher the individual success probability of services p_s, the lower is the resulting *SA-probability* p_{SA}. That is, if the success probability of single services is lower, the higher is the probability for a failure before the i-th service is executed (resulting in a semi-atomic abort of the workflow). Therefore, in this scenario with only one non-compensatable service, the optimization potential - which AWM exploits -

ranges from roughly 10% ($p_s = 0,7$ and $i = 7$) to over 60% ($p_s = 0,7$ and $i = 1$).

In Figure 10.b, the results for a parallel alignment of services $S_1, \ldots, S_{10}$ are depicted. This time, we varied the number of non compensatable services k from $k = 1, \ldots, 10$. Again, AWM ensures correct execution, thus the resulting *SA-probability* p_{SA} for AWM (black solid line) is 1. For WS-BA it can be seen, the more non compensatable services are present, the lower the probability that the execution will result in correct termination: The greater k, the greater the possibility of successful completion of one of these k non compensatable services in the presence of the failure of another one. Again, as in Figure 10.a, the probability for the workflow to result in a correct state (i.e., *SA-probability*) is lower for higher success probability $p_s = 0,7$ than for $p_s = 0,3$. The optimization potential that AWM exploits in this case ranges from roughly 30% ($p_s = 0,3$, $k = 2$) to over 90% (for $k = 10$).

We additionally evaluated AWM according to the number of coordinated elements m in the resulting workflow ω'. m is dependent on the transactional conflict elements, thus on the properties of services and the data-dependencies present. Crucial parameters are the ratio of recoverable elements p_{REC} and redoable elements p_{RED} respectively. If no data dependencies exist within ω, m converges to the number of non recoverable and non redoable elements $(1 - p_{REC}) * (1 - p_{REC}) * n$. If the ratio of recoverable services is $p_{REC} = 0,5$ as is the ratio for redoable services $p_{RED} = 0,5$, m converges to $(1 - 0,5) * (1 - 0,5) * n = 0,25 * n$. We also performed experiments varying the number and types of data-dependencies in a workflow. Even if almost all elements are data-dependent on another one, employing AWM reduced m to rougly $0,75 * n$. Thus, AWM reduced m as opposed to WS-AT by about 25%. This is due to the explicit handling of indirect conflict elements, which AWM spares to coordinate.

These basic evaluation results already confirm, that AWM increases the autonomy of participants (as opposed to approaches employing blocking of resources) by still guaranteeing correct execution. Especially in mobile environments, the autonomy of services on the one hand and correct execution of composite services despite their transactional properties on the other hand, are especially important objectives to achieve. In this paper, we proposed an approach to transactionally support mobile service composition: By exploring transactional properties, a composition is validated, if *all* executions are correct, i.e., SAP. If the verification fails, our adaptation algorithm alters the workflow ω to ω' at runtime to ensure correctness in the notion of semi-atomicity.

REFERENCES

[1] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation Specification: A New Approach to Design and Analysis of E-service Composition. In *WWW '03: Proceedings of the 12th Intl Conference on World Wide Web*, New York, NY, USA, 2003. ACM.

[2] D. Chakraborty, A. Joshi, T. Finin, and Y. Yesha. Toward Distributed Service Discovery in Pervasive Computing Environments. *IEEE Transactions on Mobile Computing*, February 2006.

[3] M.-C. Fauvet, H. Duarte, M. Dumas, and B. Benatallah. Handling transactional properties in web service composition. In *Proceedings of WISE*, pages 273–289, 2005.

[4] W. Gaaloul, M. Rouached, C. Godart, and M. Hauswirth. Verifying Composite Service Transactional Behavior Using Event Calculus. In *OTM Conferences (1)*, pages 353–370, 2007.

[5] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, 1983.

[6] H. Garcia-Molina and K. Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, 1987.

[7] A. Gupta, N. Gupta, R. K. Ghosh, and M. M. Gore. Team Transaction: A New Transaction Model for Mobile Ad Hoc Networks. In *ICDCIT*, pages 127–134, 2004.

[8] K. Hahn and H. Schweppe. Exploring Transactional Service Properties for Mobile Service Composition. In *Proceedings of Conference Mobilität und mobile Informationssysteme (MMS)*, 2009.

[9] R. Hamadi and B. Benatallah. A petri net-based model for web service composition. In *Proceedings of the 14th Australasian Database Conference (ADC'03)*, 2003.

[10] S. Jajodia and L. Kerschberg, editors. *Advanced Transaction Models and Architectures*. Kluwer, 1997.

[11] U. Küster and B. König-Ries. Semantic Service Discovery with DIANE Service Descriptions. In *Proceedings of the International Workshop on Service Composition*, Silicon Valley, USA, November 2007.

[12] F.V. Harmelen J.Hendler I.Horrocks D.L. McGuinness P.F. Patel-Schneider M. Dean, D. Connolly and L.A. Stein. Web Ontology Language (OWL) Reference Version 1.0, 2002. http://www.w3.org/TR/2002/WD-owl-ref-20021112.

[13] G. Pardon and G. Alonso. CheeTah: a Lightweight Transaction Server for Plug-and-Play Internet Data Management. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 210–219, San Francisco, CA, USA, 2000.

[14] A. Popovici and G. Alonso. Ad-Hoc Transactions for Mobile Sevices. In *Proceedings of the 3rd VLDB Workshop on Transactions and Electronic Services (TES '02)*, 2002.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proc. of ACM SIGCOMM*, 2001.

[16] O. Ratsimor, D. Chakraborty, A. Joshi, and T. Finin. Allia: Alliance-based Service Discovery for ad-hoc Environments. In *Proceedings of the 2nd international workshop on Mobile commerce*, pages 1–9, New York, NY, USA, 2002. ACM.

[17] M. Schäfer, P. Dolog, and W. Nejdl. Engineering Compensations in Web Service Environment. In *Proceedings of 7th Intl. Conference on Web Engineering (ICWE)*, pages 32–46, Como, Italy, 2007.

[18] S. Stein, T.R. Payne, and N.R. Jennings. Flexible provisioning of web service workflows. *ACM Trans. Internet Technol.*, 9(1):1–45, 2009.

[19] W. v. d. Aalst, P. Barthelmess, C.A. Ellis, and J. Wainer. Workflow Modeling Using Proclets. In *Proceedings of the 7th International Conference on Cooperative Information Systems (COOPIS'2000)*, pages 198–209, 2000.

[20] T. v. Lessen, F. Leymann, R. Mietzner, J. Nitzsche, and D. Schleicher. A Management Framework for WS-BPEL. In *ECOWS '08: Proceedings of the 6th European Conference on Web Services*, Dublin, Ireland, 2008.

[21] K. Vidyasankar and G. Vossen. A multi-level model for web service composition. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services*, page 462, Washington, DC, USA, 2004. IEEE Computer Society.

[22] G. Weikum and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In *Database Transaction Models for Advanced Applications*, pages 515–553. 1992.

[23] Web Service Semantics WSDL-S, 2005. http://www.w3.org/Submission/WSDL-S/.

[24] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. *SIGMOD Rec.*, 23(2), 1994.