Exploring Transactional Service Properties for Mobile Service Composition

Katharina Hahn, Heinz F. Schweppe
Institute for Computer Science, Freie Universität Berlin
Takustr. 9, 14195 Berlin
{khahn,schweppe}@mi.fu-berlin.de

Abstract: Service oriented computing provides suitable means to technically support distributed collaboration of heterogeneous devices such as in mobile environments. However, wireless communication links are unstable. When supporting collaboration in such environments, failures have to be optimistically coped with in order to still provide suitable correctness guarantees and avoid inconsistent system states. In this paper, we explore transactional properties of services in order to reasonably integrate transactional coordination with composition of services in mobile networks and provide suitable correctness guarantees.

1 Introduction

When deploying applications in mobile networks, one has to provide suitable means to cope with the characteristics of such environments. Due to the mobility of participants and the wireless networking technologies, mobile networks are *more dynamic* than fixed networks. This leads to *less stability* of communication links. On account of the inherent network dynamics, the *execution environment of an application is not known* at designtime and might differ from execution to execution. This also holds for the *heterogeneity* of portable devices which can be integrated in collaborative wireless computing.

Service oriented architectures are a powerful approach to support collaboration in such heterogeneous environments as they allow for loosely coupling of components thus respecting to their autonomy. Services can be dynamically discovered and composed into new value-added, so called *composed services*. Service discovery and binding at runtime provides suitable means to deal with composition of services in mobile environments in which the execution context, i.e., available services at runtime, is not previously known.

Especially due to the possibly physical distribution of several services and less stability of network links, it is indispensable to be able to cope with failures of different kinds to guarantee *correct execution*. This might come at the cost of relaxing correctness criteria such as the strict atomicity and isolation of all components which are guaranteed in databases. As strict guarantees rely on blocking of resources, services might be blocked for an arbitrary long time in case of disconnections of participants. In order to avoid blocking, "correctness" is relaxed and e.g. assured by avoiding any inconsistent and non-recoverable system states. This is achieved either through *forward-recovery* by still allowing the workflow to successfully complete or *backward-recovery* by resetting the system to a previously consistent state.

In this paper, we develop a relaxed atomicity criterion for composed services which explores the non-functional (i.e. in this context transactional) properties of services. While avoiding blocking if possible to respect the autonomy of mobile participants, transactional coordination of services is reasonably integrated in workflow management. We adapt the composition of services at runtime in order to support correct execution, which we define by the notion of a relaxed atomicity criterion (see Section 5).

Consider for example a Tourist Ticketing System which supports users in finding points of interest and booking activities according to their previously defined preferences. An example of a mobile ticket vendor, whose business model is to sell tickets for touristic events, such as an exhibition, is shown in Figure 1. To make his offer more attracting to the user, he additionally offers to organize and book according transportation facilities to the exhibition venue and reserve a table at a nearby restaurant.

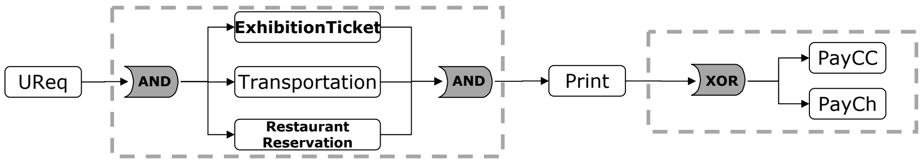

Figure 1: Combined ticketing for tourists.

After specifying the user's request (UReq), including e.g. number of persons, the workflow is split to parallely buy tickets for the *exhibition*, book *transportation* facilities and *reserve* a restaurant table. If all three components successfully finish, the tickets are *printed*, and the user is then asked to either pay by *credit card* or by *cash*. In this workflow, no failure handling has been specified yet. We argue, that it can be automated by exploring the workflow and participating services.

The rest of the paper is structured as follows: In Section 2 related work is presented. Section 3 introduces the formal model of transactional composition of services. In Section 4, transactional properties of components are explored, while Section 5 defines the atomicity guarantee. In Section 6, we outline alterations of the composition in order to support transactional execution. We conclude in Section 7.

2 Related Work

Many advanced transaction models (ATM), e.g. [GMS87, WS92] have been proposed which support transactional processing in distributed and heterogeneous databases [JK97]. 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 (such as [GGGG04, PA00, PA02]) 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.

Workflow execution and transactional coordination standards for Web-Services (WS) are two separate concerns. The execution of workflows is usually defined in BPEL (Business Process Execution Language), and controlled by workflow engines (e.g. [Apa]). Those provide support for the design, execution, visualization and analysis of workflows but do not integrate transactional guarantees. Transactional coordination has been specified by the WS-Transaction Framework (WS-Tx) [IBM05] which offers means for coordination of different services. It specifies different coordination types for short- and long-running activities and employs convenient backward recovery mechanisms to guarantee correct execution. As with most advanced transaction models proposed for asynchronous and decentralized transaction processing, WS-Tx lacks the flexibility to map different structural patterns to different transaction semantics.

Forward-recovery for composite services as e.g., proposed by [SDN07], is a promising approach to deal with the unstable availability of single participants in service oriented computing. In [SDN07], the authors propose the use of an abstract service provider. Its responsibility is to dynamically replace a failing service at runtime with a semantically equivalent service. Thus, specific failure situations are covered. But transactional execution of the whole workflow or subparts of it, is not considered.

Fauvet et. al [FDDB05] 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 interesting and powerful it uses a proprietary operator. We rather try to integrate transactional composition in existing standards by exploring non-functional properties.

In order to verify the execution of Web-Service workflows, several formalisms have been used, such as petri nets [HB03] or finite-state-machines [BFHS03]. These introduce powerful means to formally verify the execution of composite Web-Services but do not focus on transactional verification. Gaaloul et al. [GRGH07] use an event based-approach to model transactional composite services. As it provides suitable means to specify transactional behavior, we base our work on this formalism. It captures static verification however adaptation of the workflow in order to guarantee correct execution is left to the designer.

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 [RFH+01, CJFY06, RCJF02], as well as semantic description languages, e.g., OWL-S, WSDL-S or DSD [MDS02, WSD05, Kle04] to support matching of service offers and requests.

We also want to relate to the work done in the area of workflow scheduling, which identifies the problem of finding correct execution sequences for workflow activities, obeying inherent constraints, e.g. temporal or causality constraints [ASSR93, DKRR96]. Other approaches focus on minimizing communication costs or ensuring prearranged QoS obligations defined in service level agreements [DD07]. As opposed to those, our work focuses on the analysis in order to reschedule activities to guarantee transactional correctness.

3 Formally Modeling Transactional Composition of Services

In the following, we introduce the model used to focus on the transactional behavior of services and composite services. The model is based on the event-algebra presented by [GRGH07]. It is also used to specify the relaxed transactional guarantees to be given for composite services, which we define in Section 5.

3.1 Service Model

The behavior of a service is modeled using a state-machine. Regardless its properties, a service has at least 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), either due to an internal error or externally triggered. If the service completed successfully, its state is *completed*. Additionally, if the service is compensatable, such as the example in Figure 2, it also has a *compensated* state.

The transitions between these states are either *internally* triggered (indicated by solid lines), i.e., by the service itself, or *externally* triggered (indicated by dashed lines), i.e., by another entity, such as the workflow engine, another service or a person.

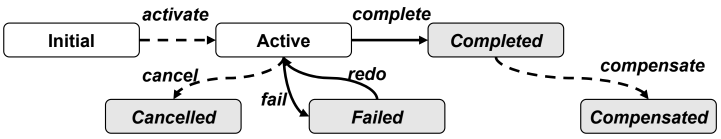

Figure 2: State machine of service.

3.2 Transactional Composition of Services

A composite service consists of a set of components and a set of axioms which define the correlation between the components. The normal execution defines the execution flow in case no failure occurs. *Transactional* composition of services additionally defines the relation of the components in case of failure in order to avoid inconsistent states, e.g. by compensation. So far, failure handling is left to the designer. By defining a workflow, the designer defines the regular execution; using compensation handler, he is able to specify behavior in case of *specific* failure situations. Thus, transactional guarantees for the whole workflow are not generally given.

The regular execution of services is modeled using *normal execution dependencies*. A

normal execution dependency between services S_i and S_j states that the completion of S_i triggers the activation of S_j:
Normal execution dependency: $depNrm(S_i, S_j) : S_i.complete() \Rightarrow S_j.activate()$

The standard failure handling mechanisms are cancellation of active services, compensation for completed services or activation of alternatives. According to these mechanisms and the possible failure situations, the following dependencies are defined:[1]

- Alternative dependency: $depAlt(S_i, S_j) := S_i.fail() \Rightarrow S_j.activate()$
- Fail-Cancel dependency: $depFlCln(S_i, S_j) := S_i.fail() \Rightarrow S_j.cancel()$
- Fail-Compensate dependency: $depFlCps(S_i, S_j) := S_i.fail() \Rightarrow S_j.compensate()$
- Cancel-Cancel dependency: $depClnCln(S_i, S_j) := S_i.cancel() \Rightarrow S_j.cancel()$
- Cancel-Compensate dependency:
  $depClnCps(S_i, S_j) := S_i.cancel() \Rightarrow S_j.compensate()$
- Compensate-Compensate dependency:
  $depCpsCps(S_i, S_j) := S_i.compensate() \Rightarrow S_j.compensate()$

In order to guarantee transactional execution, we propose to automatically add these transactional dependencies and thus provide transactional guarantees. In the stated example (recall Figure 1), an *alternative* dependency has to be added between *PayCC* and *PayCh*, since the later is an alternative for the former one. Additionally *fail-cancel* dependencies are to be added between the *ticketing*, *transportation* and *reservation* services, as those should either all complete or none. Which dependencies are to be added, depends on the transactional properties of a service as well as its structural context. This is examined using workflow patterns.

3.3 Workflow Patterns

The structure of a composite service is represented by workflow patterns. Formally, a *workflow pattern* is a function that given a set of services returns the control flow [ABEW00]. Components of patterns are either services or contained patterns. By $WP(S)$, we denote a workflow pattern with a set of elements S. In the following, we exemplary present three common workflow patterns and their characteristics for transactional workflow management.

SEQUENCE-Pattern This is a basic pattern which is used by the designer to specify, that one service is activated after the completion of the previous one. It is also known as *sequential* or *serial routing*. The invocation of the services is done in the same control thread. Arranging services S_i and S_j in sequence always infers a normal execution dependency (see Section 3.2) between S_i and S_j.

[1]Note, that besides the alternative dependency, their denotation encodes cause and effect.

AND-Pattern Services, which are arranged in an AND-pattern are executed in parallel. Thus, the control flow is split up in parallel threads which are executed independently of each other. Therefore, we assume that not data dependencies between components of one AND-pattern exist. The control flow is synchronized at the join point and the subsequent workflow is activated as soon as *all* elements of the AND-pattern are completed. E.g., in our example *print* is invoked if *ticketing*, *transportation* and *reservation* are completed. The AND-pattern is also known as *parallel split* or *parallel routing*.

XOR-Pattern Based on any control data, one branch out of many is chosen. As these branches are never executed in parallel, the workflow is continued as soon as *one* branch completes. The XOR-pattern is also known as *exclusive choice*, *switch* or *conditional routing*. We focus on situations in which the choice relies on non-functional properties of services. Considering the formal model, alternative execution dependencies between elements of an XOR-pattern exist, as the failure of one can be recovered by the execution of another one. Thus, the XOR-pattern can be used to integrate forward-recovery, as the failure of one participant might be compensated by the execution of another one (e.g. *PayCC* and *PayCh*).

These patterns specify the control flow of the composite service and additionally the execution semantics. In an AND-pattern all included elements have to be completed while an XOR-pattern specifies, that one and only one element is to be completed. This already indicates the transactional dependencies to be added to avoid inconsistent states: E.g., the semantics of the AND-pattern infer, that in case that one element fails, all of the others have to be canceled. Thus *fail-cancel* dependencies between all elements are added. By contrast, if an element within an XOR-pattern fails, an alternative can be executed, thus *alternative* dependencies are added between elements of the XOR-pattern. However, the transactional dependencies are also influenced by the transactional properties of services, which are studied in the next section.

4 Exploring Non-Functional Service Properties

Within this section, we introduce non-functional properties of services which are relevant to define appropriate failure handling mechanisms. Additionally, we derive properties for workflow patterns according to the contained elements.

4.1 Transactional Service Properties

In order to ensure correct execution of composite services, we examine the transactional behavior of services: I.e., whether services can fail, whether completed services can be compensated for and whether they need to be recovered in case of failure. The first two properties (redoability and compensatability) have already been considered in the context of flexible transactions [ZNBB94]. We additionally classify services according to their need for compensation in case of failure.

Compensatability

The compensatability of a service indicates whether its effects can be undone after completion. Thus a service S is compensatable (denoted as $S.comp = 1$ and $S.comp = 0$ accordingly for non-compensatable services) if there exists a service C which semantically undoes the effects of S.[2] This is formally specified by the *compensate*-transition and a *compensated*-state as shown in Figure 2.

Redoability

The redoability of services specifies whether the service's execution can fail, i.e. a redoable service S (denoted as $S.redo = 1$) will definitely complete after being activated. This is important for example for compensating services, as it is assumed that their execution will not fail. Redoability of a service is modeled by an internal *redo*-transition (see Figure 2). Thus, once invoked the completion of the service can be guaranteed.

Consistent Closure

By including a service in a workflow, the designer states whether its completion is inevitable for the success of the workflow (e.g., if no alternatives exist). So far, it is vice versa assumed, that a service is only allowed to be completed if the workflow is completed. However, some services may allow for *inconsistent closure*, i.e. they complete although the workflow may be canceled. This is usually given by semantics of the service or the consistency demands of the data it operates on. Examples are a log-in service which authenticates the user to the provider or a printing service which prints a confirmation. We therefore additionally consider the following transactional property of services:

A service S demanding consistent closure[3] (denoted as $S.consCompl = 1$) needs recovery in case the workflow is rolled back. Thus, its completion infers the completion of the workflow. A service, that is allowed to complete *in*consistently ($S.consCompl = 0$) does not need recovery, in case the workflow execution fails. Thus it states, whether the effects of a service *have to be* undone in case of recovery. Within our model, {*fail, cancel, compensate*}-compensate dependencies (*-compensate dependencies for short) to a service demanding consistent completion have to be added.

Derived Property: Recoverability

For validation and defining the appropriate backward recovery mechanisms, i.e. adding the transactional dependencies, it is of interest to know, whether a service (or a pattern) can be backward-recovered in case of failure. We denote this by *recoverability* of an element. A service S is *recoverable* ($S.recover = 1$), if and only if it is compensatable or does not need consistent completion. This property is especially interesting for patterns: A pattern is regarded to be *recoverable* (denoted as $WP(S).recover = 1$), if all of the executed services which demand consistent completion are compensatable and all contained patterns are recoverable.

[2]Non-compensatable services are sometimes also referred to as *pivot* services.

[3]We use the term *consistent completion* interchangeably.

4.2 Inferring Pattern Properties

As stated before, we want to analyze the workflow with the bound services at runtime. We thus analyze the composite service bottom-up and derive the transactional properties for patterns according to the contained elements, as indicated by the dashed lines in Figure 1. The transactional properties of a pattern $WP(S)$ are denoted just as the transactional service properties: $WP(S).comp$, $WP(S).consCompl$, $WP(S).redo$ and $WP(S).recover$. They are determined according the properties of in included elements as well as the pattern (i.e., its execution semantics). As the SEQUENCE and the AND-pattern both state that *all* included elements have to be completed, their transactional properties are determined in the same way.

Transactional Properties of the SEQUENCE and AND-Pattern

A pattern $WP(S)$ which is a SEQUENCE-pattern or an AND-pattern

- is compensatable, if and only if all included services are compensatable: $WP(S).comp = 1 \Leftrightarrow \forall S_i \in S : S_i.comp = 1$
- needs consistent completion, as soon as one service needs consistent completion: $WP(S).consCompl = 1 \Leftrightarrow \exists S_i \in S : S_i.consCompl = 1$
- is redoable, if all included services are redoable: $WP(S).redo = 1 \Leftrightarrow \forall S_i \in S : S_i.redo = 1$
- is recoverable, if all included services are either compensatable or allow for inconsistent completion: $WP(S_i).recover = 1 \Leftrightarrow \forall S_i \in S : S_i.comp = 1 \vee S_i.consCompl = 0$

Transactional Properties of the XOR-pattern

Due to the different execution semantics (one and only one included element is to be executed), the transactional pattern properties for an XOR-pattern are determined differently than for the SEQUENCE- and the AND-pattern. As it cannot be previously to execution stated which service will complete, the pattern properties can only be previously determined for the following (not all) cases. Otherwise they cannot be determined until execution. The XOR-pattern

- is compensatable, if all included services are compensatable. It is non-compensatable, if all included services are non-compensatable. Otherwise, it is not known previous to execution.
  $WP_{XOR}(S).comp = 1 \Leftarrow \forall S_i \in S : S_i.comp = 1$
  $WP_{XOR}(S).comp = 0 \Leftarrow \forall S_i \in S : S_i.comp = 0$
- needs consistent completion, if all included services demand consistent completion. If none of the included services demands consistent closure, the pattern allows for inconsistent closure. Otherwise, it is not known until execution.
  $WP_{XOR}(S).consCompl = 1 \Leftarrow \forall S_i \in S : S_i.consCompl = 1$
  $WP_{XOR}(S).consCompl = 0 \Leftarrow \forall S_i \in S : S_i.consCompl = 0$

- is redoable, if at least one service is redoable. This service can be invoked in case of failure of any other included service. Else, the pattern is non-redoable.
  $WP_{XOR}(S).redo = 1 \Leftrightarrow \exists S_i \in S : S_i.redo = 1$
- is recoverable, if all included services are either compensatable or allow for inconsistent completion.
  $WP_{XOR}(S).recover = 1 \Leftarrow \forall S_i \in S : S_i.Comp = 1 \vee S_i.consCompl = 0$
  $WP_{XOR}(S).recover = 0 \Leftarrow \forall S_i \in S : S_i.comp = 0 \wedge S_i.consCompl = 1$

An arbitrary combination of the properties *compensatability*, *consistent closure* and *redoability* is possible. The *recoverability* is derived by the former properties for services and by the properties of the included components for patterns. Note, that for verification purposes, the compensatability of a service is disregarded in case of inconsistent completion. However, it is important when adding transactional dependencies to a composite service. We denote the transactional properties of a service or a pattern S as P_S defined as follows:

$$P_S = (S.comp, S.consCompl, S.redo, S.recover)$$

Accordingly, a service S with $P_S = (0, 1, 1, 0)$ is a service which is not compensatable, demands consistent completion, is redoable and thus not recoverable. Based on the this model, we will now introduce the transactional guarantees that we support when dealing with transactional workflows.

5 Relaxing Atomicity for Transactional Composite Services

Blocking of resources is counterproductive in the environment of loosely coupled and especially mobile services. Due to the autonomy of mobile services and the fluctuant availability of those, relaxed atomicity guarantees have to be defined. These specify the criteria for *correct execution*. In order to avoid blocking situations, different notions of relaxed atomicity e.g., *semantic atomicity* [GM83] and *semi-atomicity* [ZNBB94], which allow the commitment of subtransactions at different times have been proposed for database transactions. Convenient backward-recovery mechanisms ensure that already committed subtransactions are recovered in case of failure. In the model of flexible transactions [ZNBB94], semi-atomicity is validated by reviewing the order of subtransactions according to their non-functional properties: The commitment of compensatable subtransactions *precedes* the commitment of pivot subtransactions. As their commitment infers the commitment of the whole transaction, it is only *followed* by redoable subtransactions.

We adapt the model of semi-atomicity defined for flexible transactions and extend it to comprise transactional workflow management in mobile environments. By defining accepted termination states (ATS), the designer defines representational sets of services whose completion reflect the successful execution of the composite service. Multiple sets exists, if alternatives in the workflow exist (or even multiple ATS exist).

We define *semi-atomic* execution of the workflow with respect to the transactional properties of components (as presented in Section 4) as follows:

Semi-Atomicity of Composite Services: Semi-atomic execution of a composite services with defined ATS is ensured if

- either all services belonging to one valid execution path to an ATS are completed and all other services which demand consistent completion are not completed
- or no service demanding consistent completion is completed.

As recovery for services which do not demand consistent completion is disregarded, this relaxes the semi-atomicity as defined for flexible transactions.

6 Ensuring Semi-Atomicity for Composite Services

In this section, we present how semi-atomicity of a composite service as defined in Section 5 is supported. Due the increased autonomy of mobile devices as opposed to fixed wired networks, we try to avoid the use of blocking protocols such as the 2PC. By exploring the transactional service properties in the context of workflow patterns at runtime, the structure of the composite service is altered and choice on alternatives is influenced to support semi-atomic execution.

6.1 Alter Execution Type and Order

At first, we consider the execution order of components. It is initially given by the workflow pattern, they are arranged in (either parallel or sequential). In case of failure, it has to be ensured, that all already completed services are recoverable or forward-recovery for the failing services exists. This is identified by considering the transactional properties of components. Recall, that the transactional properties of a component S are denoted as:

$$P_S = (S.comp, S.consCompl, S.redo, S.recover)$$

Assuming no data dependency between two services S_i and S_j, then their execution is aligned (i.e., S_i precedes S_j) in the following cases.

1. $P_{Si} = (*, *, 0, 1)$ and $P_{Sj} = (0, 1, *, 0)$, or
2. $P_{Si} = (0, 1, 0, 0)$ and $P_{Sj} = (0, 1, 1, 0)$.

Otherwise, in case S_j completes but S_i fails, the semi-atomicity of the workflow is harmed as in all stated cases, S_j cannot be recovered. If at least two services S_i and S_j both are not recoverable and not redoable, i.e. $P_{Si} = P_{Sj} = (0, 1, 0, 0)$, then semi-atomic execution is only assured by *coordinating* them within a subtransaction, e.g. using WS-Tx. Otherwise, in case of failure of one of them and the completion of the other, the composite service is in an inconsistent state which cannot be recovered.

Any other combination of elements can be executed *independently* of each other without harming the semi-atomicity of the execution.

Recall the example of the combined ticketing from Figure 1. The designer indicated to execute the ticketing *Ti*, transportation *Ta* and reservation *R* services in parallel. The Ti be a local service with $P_{Ti} = (1, 1, 0, 1)$. Providers for *Ta* and *R* are discovered at runtime. Assume that services with the properties $P_{Ta} = (0, 1, 1, 0)$ and $P_R = (0, 0, 1, 1)$ are discovered. The workflow is then altered as shown in Figure 3a. *Ti* precedes the execution of *Ta*, *R* is executed parallely.

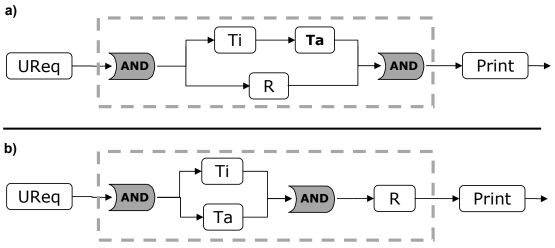

Figure 3: Altering execution order of components.

Assume that executed again different services with the following properties $P_{Ta} = (1, 1, 0, 1)$ (i.e., compensatable) and and $P_R = (0, 1, 0, 0)$ (i.e., non-compensatable and non-redoable), then the composition is altered as shown in Figure 3b. The execution of *Ta* and *Ti* is parallelized, while *R* is executed after them. Otherwise, if the workflow was executed as originally intended, then failure of either *Ta* or *Ti* with concurrent completion of *R* would lead to a non-recoverable system state. In both cases the non-functional properties of the pattern WP included in the dashed lines are $P_{WP} = (0, 1, 0, 0)$.

6.2 Service Selection According to Transactional Properties

According to their context, transactional properties of services are used to determine a preference relation on which service to include in an XOR-pattern. Consider for example Figure 4: At runtime, either branch S_i or S_j is to be taken within the XOR-pattern, before the subsequent workflow S_{subseq} is invoked. Let the transactional properties be $P_{Si} = (1, 1, 0, 1)$ and $P_{Sj} = (0, 1, 1, 0)$. According to Section 4.2, the XOR-pattern is thus redoable, as S_j is redoable. If $P_{Sprev} = (1, 1, *, 1)$ and $S_{subseq}.redo = 0$ then, S_i must be chosen in order to guarantee semi-atomicity. Otherwise, in case of failure of S_{subseq}, the XOR-pattern cannot be recovered. If in contrast $P_{Sprev} = (0, 1, *, 0)$ and $S_{subseq}.redo = 1$, then the XOR-pattern has to complete to ensure semi-atomicity. This is given trough the redoability of pattern which is already assured through the presence of S_j (as $S_j.redo = 1$). Thus, in this case, the choice between those two services can be done according to other non-functional properties.

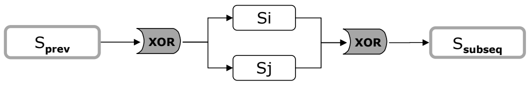

Figure 4: Choosing S_i or S_j according to transactional properties.

6.3 Automation of Recovery: Adding Transactional Dependencies

After dynamically adjusting the workflow, transactional dependencies have to be added at runtime in order to define the appropriate forward- and backward-recovery mechanisms. We will add dependencies according to the workflow patterns and alter those according to the non-functional properties of services.

1. All components S_i and S_j for which a normal execution dependency in the form $depNrm(S_i, S_j)$ exists, the appropriate backward-recovery in case of failure, cancellation or compensation are added. Thus dependencies of the form $depFlCps(S_j, S_i)$, $depClnCps(S_j, S_i)$ and $depCpsCps(S_j, S_i)$ are added ($dep * Cps(S_j, S_i)$ for short).

2. All components which do not need consistent completion, are omitted in backward recovery. Thus if S_j does not need consistent closure and any transactional dependencies in the form $depXCps(S_k, S_j)$ and $depXCps(S_j, S_i)$ exist, for $X \in \{Fl, Cln, Cps\}$, then those are consolidated to an appropriate dependency of the form $depXCps(S_k, S_i)$. Thereby, S_j is skipped in case of backward recovery.

3. All compensatable components in an AND-pattern $WP_{AND}(S)$ are to be compensated in case of failure. Thus, the following dependencies are added to the compensatable components $S_c \in S$ from the subsequent workflow S_{sub}: $dep * Cps(S_{sub}, S_c)$. Additionally, all parallely arranged components are to be cancelled, if failure or cancellation of one of them occurs. Thus, for all $S_j \in S$, $depFlCln(S_j, S)$ and $depClnCln(S_j, S)$ are added.

4. Components aligned in an XOR-pattern $WP_{XOR}(S)$ are considered to be alternatives. After analyzing their preference relation, alternative dependencies of the form $depAlt(S_i, S_j)$, regarding the ranking order are added (i.e., S_j is an alternative for S_i). If S_j is redoable, the alternative dependency in the form of $depAlt(S_j, S_k)$ is deleted, as failure of S_j will not occur.

5. Finally, for any S_j in the composite service, which is not compensatable, any dependency of the form $dep * Cps(S_k, S_j)$ is deleted (as it is not compensatable). On the other hand, if S_j is redoable, then it cannot fail, thus any failure dependency in the form $depFl * (S_j, S_k)$ is deleted.

The approaches introduced in Section 6.1 to 6.3 enable adaptations of the composite service at runtime according to transactional properties of all components. These adaptations and the appropriate recovery mechanisms support semi-atomic execution of the workflow.

7 Conclusion

In this paper, we have introduced adapted notion of semi-atomicity which explores transactional properties of services to define correct execution for composition of mobile services. This is especially interesting for mobile networks, as those are more dynamic and error-prone. We considered the properties whether services can fail, can be compensated and need to be compensated in the context of workflow patterns. As the execution context is not previously known in mobile environments, we outlined dynamical adaptations of the composition at runtime in order to support semi-atomic execution in the current execution context. Appropriate backward- and forward recovery mechanisms are integrated by transactional dependencies.

By adapting the workflow, i.e. changing order and type of invocation, and influencing the preference relation for alternatives, it is possible to support semi-atomic execution. Ordering services avoids coordination through blocking and still provide transactional execution guarantees. As opposed to existing approaches, we explore *existing* standards for service composition to *automate* transactional execution of composite services. As part of future work, we want to design an adaptive, mobile workflow engine which exploits transactional service properties to automate transactional execution at runtime.

References

[ABEW00] 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.

[Apa] Apache Orchestration Director Engine. http://ode.apache.org/.

[ASSR93] P. C. Attie, M. P. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. In *Proceedings of the 19th VLDB Conference*, 1993.

[BFHS03] Tevfik Bultan, Xiang Fu, Richard Hull, and Jianwen Su. Conversation Specification: A New Approach to Design and Analysis of E-service Composition. In *WWW '03: Proceedings of the 12th International Conference on World Wide Web*, pages 403–410, New York, NY, USA, 2003. ACM.

[CJFY06] Dipanjan Chakraborty, Anupam Joshi, Tim Finin, and Yelena Yesha. Toward Distributed Service Discovery in Pervasive Computing Environments. *IEEE Transactions on Mobile Computing*, February 2006.

[DD07] Dmytro Dyachuk and Ralph Deters. Service Level Agreement Aware Workflow Scheduling. In *Proceedings of International Conference on Services Computing (SCC)*, volume 0, pages 715–716. IEEE Computer Society, 2007.

[DKRR96] H. Davulcu, M. Kifer, C. R. Ramakrishnan, and I. V. Ramakrishnan. Logic Based Modeling and Analysis of Workflows. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–3. ACM Press, 1996.

[FDDB05] M.-C. Fauvet, H. Duarte, M. Dumas, and B. Benatallah. Handling Transactional Properties in Web Service Composition. In *WISE*, pages 273–289, 2005.

[GGGG04] Ankur Gupta, Nitin Gupta, R. K. Ghosh, and M. M. Gore. Team Transaction: A New Transaction Model for Mobile Ad Hoc Networks. In *ICDCIT*, 2004.

[GM83] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, 1983.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, 1987.

[GRGH07] Walid Gaaloul, Mohsen Rouached, Claude Godart, and Manfred Hauswirth. Verifying Composite Service Transactional Behavior Using Event Calculus. In *OTM Conferences (1)*, pages 353–370, 2007.

[HB03] Hamadi and Benatallah. A Petri Net-based Model for Web Service Composition. In *Proceedings of the 14th Australasian Database Conference (ADC'03)*, 2003.

[IBM05] WebServices AtomicTransaction, 2005. http://www.ibm.com/developerworks/library/specification/ws-tx/.

[JK97] Sushil Jajodia and Larry Kerschberg, editors. *Advanced Transaction Models and Architectures*. Kluwer, 1997.

[Kle04] Michael Klein. Handbuch zur DIANE Service Description. Technical Report 2004-17, Universität Karlsruhe, Faculty of Informatics, December 2004.

[MDS02] 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.

[PA00] G. Pardon and G. Alonso. CheeTah: a Lightweight Transaction Server for Plug-and-Play Internet Data Management. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 210–219, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[PA02] 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.

[RCJF02] 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.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proccedings of ACM SIGCOMM*, 2001.

[SDN07] Michael Schäfer, Peter Dolog, and Wolfgang Nejdl. Engineering Compensations in Web Service Environment. In *Proceedings of 7th Intl. Conference on Web Engineering (ICWE)*, pages 32–46, Como, Italy, 2007.

[WS92] Gerhard Weikum and Hans-Jorg Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In *Database Transaction Models for Advanced Applications*, pages 515–553. 1992.

[WSD05] Web Service Semantics WSDL-S, 2005. http://www.w3.org/Submission/WSDL-S/.

[ZNBB94] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. *SIGMOD Rec.*, 23(2), 1994.