Analysis of Non-Functional Service Properties for Transactional Workflow Management

Katharina Hahn, Heinz Schweppe

Institute for Mathematics and Computer Science, Freie Universität Berlin
{khahn|schweppe}@mi.fu-berlin.de

Abstract. With the encapsulation of functionality in services, many applications are nowadays built on composite Web-Services. Those are specified using workflow execution languages, such as BPEL, which represent the structure of the composition. However, they do not integrate transactional guarantees such as failure-atomicity. It is up to the application designer to define appropriate failure handling mechanisms. Transactional coordination specified for Web-Services lacks the possibility to map structural patterns of the execution semantics. In this paper, we analyze workflow patterns in the presence of non-functional (especially transactional) properties of services in order to provide appropriate forward- and backward recovery mechanisms. We identify workflow adaptations to support transactional execution in the presence of dynamic service binding. This is done by adapting the structure of the workflow and identifying non-functional preference relations for service alternatives.

1 Introduction

Web-Services (WS) allow for applications to be built upon heterogeneous and possibly distributed components connected through standard protocols. Encapsulation of functionality whose implementation is transparent through a standardized interface supports reusability and interoperability between different soft- and hardware platforms. Assembling different services allows for new value-added *composite services*. Their composition and thus their structure is defined through the arrangement of services in workflows. Especially through the possibly physical distribution of several services, it is indispensable to be able to cope with failures of different kinds to guarantee *correct execution*. This implies the assurance, that the workflow will not result in any inconsistent state, which cannot be healed. Failure handling is either done through *forward-recovery* by still allowing the workflow to successfully complete or *backward-recovery* by resetting the system to a previously consistent state.
Workflow execution and transactional coordination for WS have often been two separate concerns. The execution of workflows is usually defined in BPEL, and controlled by workflow engines (e.g. [1]). Those provide support for the design, execution, visualization and analysis of workflows but do not support transactional guarantees. Transactional coordination has been specified by the WS-Transaction Framework WS-Tx [13] which offers means for coordination

of different services. It specifies two different coordination types for short- and long-running activities. As blocking situations are to be avoided especially for long-running business activities, relaxed atomicity requirements and convenient backward-recovery mechanisms are employed to guarantee consistent execution. As with most advanced transaction models (ATM) proposed for asynchronous and decentralized transaction processing, WS-Tx lacks the flexibility to map different structural patterns to different transaction semantics.
By considering *non-functional* (i.e., in this context *transactional*) properties of WS, transactional coordination of services can be reasonably integrated with workflow management. Gaaloul et al. [9] employ an event-calculus to verify transactional behavior of composite WS. Transactional properties of services, such as compensatability and redoability, are used to formally model their behavior. *Normal execution dependencies* between services which represent their correlation as well as *transactional dependencies* to model situations in which failures occur are used to validate the execution of the workflow. If the validation fails, the designer is asked to alter or add constraints until consistent execution can be guaranteed. While this verification is indispensable for correct workflow execution, it is statically done before or after the execution. However, this limits the visions and possibilities of workflow engines to dynamically bind services at runtime. Dynamical changes of the workflow during runtime are not captured.
Our contribution is the analysis of transactional workflow patterns along with transactional service properties. We are thereby able to provide means for transactional workflow execution: By adapting the structural design of the workflow according to the deployed scenario, we ensure correct execution in the presence of dynamic service binding. Thus, we integrate forward-recovery by substitution of services with different transactional properties at runtime. This increases the chances of successful execution in case of failure. Additionally, we illustrate the influence of transactional properties on the preference relation of services.

We will recurrently refer to the following **application scenario** of a travel agency. As a special offer, the agency promotes a trip to Berlin, including transportation to Berlin, accommodations and ticket reservation for the Blue Man Group BMG (www.blueman.com).

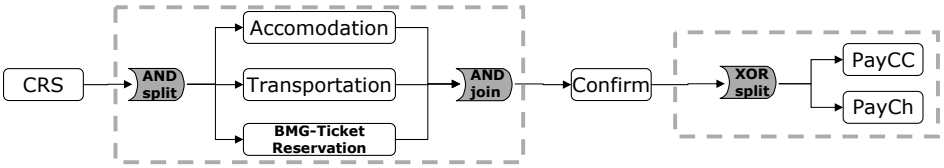

Fig. 1. Workflow of the Travel Agency.

The workflow of the agency, as created by the designer, is shown in Figure 1. At first, a customer is asked to specify its requests (*CRS*), i.e. date of the trip, number of travelling persons and other personal preferences. The workflow then parallely (intended by the AND-split) books *Accommodation*(s), *Transportation*

to Berlin and back, and reserves tickets for the BMG show (*BMG-Ticket Reservation*). The threads are then synchronized, and if all are completed, the customer agrees on the legal terms and the booked tickets are printed (*Confirm*). Finally, the payment is performed, either by credit card (*PayCC*) or by cash (*PayCh*). 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.

The rest of the paper is structured as follows: In Section 2 related work is presented. Section 3 contains the transactional model of service composition and Section 4 the transactional guarantees. In Section 5 and 6 we present the results of our analysis and their application. We conclude in Section 7.

2 Related Work

Many advanced transaction models (ATM) have been proposed which support transactional processing in distributed and heterogeneous databases [14]. These use less strict notions of atomicity and isolation in order to avoid blocking situations. Although they are very powerful, they are not capable to integrate structural requirements of complex applications in one transaction. Mobile transaction models (such as [11, 16]), which are to be deployed in more volatile environments, are able to cope with failures due to frequent disconnections. However, they do not consider different structural patterns of cooperation of services as well. Failure handling through forward-recovery and thus dynamic service binding at runtime has been proposed, e.g. by [17]. But this concept focuses on one specific failure situation to replace one service with another rather than ensuring transactional execution of the whole workflow. Transactional service properties are not considered.

Fauvet et. al [8] 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 differs inherent in the examination of transactional properties. Our work mainly differs in the approach, as we integrate transactional composition in existing standards, such as workflow patterns.

In order to verify the execution of Web-Service workflows, several formalisms have been used, such as petri nets [12] or finite-state-machines [3]. These introduce powerful means to formally verify the execution of composite Web-Services but do not consider the possibility to dynamically adjust workflows at runtime. Gaaloul et al. [9] use an event based-approach to model transactional composite services. As it provides suitable means to specify transactional behavior, we use this formalism in our work. However, this work does not explore dynamical adaptations of the workflow.

Dynamic service binding can only be performed if services can be discovered and matched at runtime. We refer to existing approaches, such as UDDI or

group-based service discovery (GSD) [4] (an approach for discovering services in mobile environments) for discovery and description languages such as OWL-S [6] or Diane Service Description DSD [15] for matching.

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 [2, 5]. Other approaches focus on minimizing communication costs or ensuring prearranged QoS obligations defined in service level agreements [7]. As opposed to those, our work focuses on the analysis in order to reschedule activities to guarantee transactional correctness.

3 Transactional Composition of Services

In this section, we specify the components of a transactional composite service model. The model is used to illustrate the transactional behavior of services and composite services to define the attempted transactional guarantees (Section 4). It is based on the event-algebra presented in [9].

3.1 Service Model

In order to focus on its transactional behavior, a service is modeled as a state-machine. Each service has at least the following states: *initial*, *active*, *completed*, *cancelled* and *failed*. If a service can be compensated for, it also has a *compensated* state, as for example in Figure 2. If a service is completed, then it is successfully executed and its changes are visible. The states cancelled, failed and compensated indicate, that service execution has not successfully completed, thus no changes are made persistent.

Transitions between these states are either *internally* or *externally* triggered. Internal transitions (indicated by solid lines in Figure 2) are triggered through the execution of the service itself, e.g. complete or fail. External transitions (represented by dashed lines), e.g. activate, are triggered by an external entity, such as another service, the workflow management system or the application.

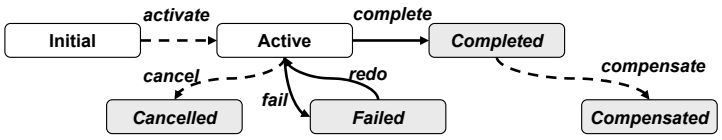

Fig. 2. State machine of a compensatable and redoable service.

3.2 Transactional Composite Service

Composite services (e.g., our example in Figure 1) consist of a set of component services and a set of axioms which defines their correlation. It has to be stated

which event triggers the activation of each service but additionally what happens in case services fail. Regular execution is defined by *normal execution dependencies*, i.e. the completion of a service triggers the activation of another. In Figure 1, normal execution dependencies between CRS and all services within the AND-pattern (Transportation, Accommodation and BMG-Ticket Reservation) exist, as the completion of CRS triggers the activation of those.

Additionally, failure handling mechanisms, such as executing an *alternative*, *cancelling* or *compensating* for a service have to be explicitly specified. Those are referred to as *transactional execution dependencies*. In our example, the failure of any service within the AND-pattern triggers the cancellation of the other services within the pattern. As those should either all be completed or none. Additionally, PayCh is an alternative for PayCC, if PayCC fails. So far, it is up to the designer, to explicitly define failure handling for each situation. Through the analysis of workflow patterns in the presence of transactional service properties, we want to automate this process.

3.3 Workflow Patterns

Formally, a *workflow pattern* is a function that given a set of transactional services returns a control flow [18]. In the following, we exemplary present three common workflow patterns and their characteristics.[1]

SEQUENCE Two services arranged in sequence state that one service is enabled after the completion of its preceding one. The invocation of both services is done in a single 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.

AND Using an AND-split, the designer parallelizes the control flow. Contained services are executed inependently from each other. The control flow is synchronized at the join-point and the subsequent workflow is activated as soon as *all* services have completed. In our example, Accomodation, Transportation and BMG-Ticket Reservation are executed in parallel.

XOR Based on any control data, one branch out of many is chosen to continue the execution of the workflow. As these branches are never executed in parallel, the workflow is continued as soon as *one* branch completes. Referring to our example, PayCC and PayCh in the XOR pattern are alternatives. We will show in Section 5, that transactional properties of services also influences the choice of which one to prefer.

The definition of these patterns does not initially specify any failure handling yet. *Transactional workflow patterns* are workflow patterns which are augmented with *transactional dependencies* as introduced in Section 3.2. We state, that if transactional execution of the workflow is desired, that dependencies are to be automatically added to workflow patterns. So far, in our example, cancellation dependencies are to be added among the services in the AND-pattern (*Accommodation*, *Transportation* and *Ticket Reservation*), as either all of them are to

[1] Due to space limitations, we restrict the presentation to three patterns.

be completed or none. Considering the XOR-pattern, *PayCh* is to be specified as an alternative for *PayCC*, so in case *PayCC* fails, *PayCh* is executed. However, these transactional execution dependencies are also influenced by the transactional properties of services which are specified in the following section.

3.4 Transactional Service Properties

In order to be able to decouple the completion of services in time, we regard the following non-functional properties. We examine the transactional properties that have been considered for flexible transactions. Additionally, we introduce the property of *consistent completion*, that to the best of our knowledge, has not been considered for transactional service composition so far.

Compensatability A service S is considered to be compensatable (denoted as $S.comp = 1$ and $S.comp = 0$ accordingly for non-compensatable services which are sometimes referred to as *pivot* services) if there exists a service C which semantically undoes the effects of S. A cancellation of a booked tour is the compensating service for the booking itself. In the employed model, compensatability is expressed through a *compensate*-transition and a *compensated*-state (see Figure 2). Compensatability indicates, whether the effects of a service *can* be undone after completion.

Redoability A redoable service S (denoted as $S.redo = 1$) will definitely complete if its activation is repeated a positive number of n times. This is e.g. important, when considering the compensating service: Assuming the compensatability of a service, it is also assumed, that the compensating action will complete (i.e. not fail). Redoability of a service is modeled through a *redo*-transition (see Figure 2). Once invoked the completion of the service can be guaranteed.

Through the inclusion of a service in the workflow, a designer states, whether the completion of the service is inevitable for the completion of the workflow. It is vice versa assumed, that a service is only allowed to be completed if the workflow is completed. E.g., no hotel room is allowed to be booked, if the whole trip is not booked. However, some services may allow for *inconsistent completion*, i.e. they complete although the workflow may be cancelled. This is usually given by the consistency demands or the semantics of the service. Consider e.g. a printing service or a registration services at a conference: If the payment will not be pursued two days after registration the latest, then the registration will be deleted. We therefore introduce the following transactional property of services:

Consistent Completion A service S demanding consistent completion (denoted as $S.consCompl = 1$) needs recovery in case the workflow is rolled back. Thus, its completion infers the completion of the workflow.[2] 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 after completion in case of backward-recovery.

[2] The completion of the workflow does not infer completion of the service, as alternatives might exist.

Referring to our example of the travel agency, *BMG-ticket reservation* reserves tickets for the show at a certain date. The tickets have to be picked up one hour before the show starts at the latest. Otherwise, the reservation is deleted. As it demands interaction outside the workflow, the completion of this service does not necessarily need to be recovered in case the booking of the whole trip fails. An arbitrary combination of these properties is possible, although the compensatability of a service is disregarded in case of inconsistent completion. We denote the transactional properties of a service as a triple P_S defined as follows:

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

Accordingly, a service S with $P_S = (0, 1, 1)$ is a service which is not compensatable, demands consistent completion and is redoable.

These non-functional properties *also* influence the transactional dependencies that are added to the workflow: A compensation dependency may only exist, if the service is compensatable. Additionally, if a service does not need consistent completion, a compensation depency is not needed.

Based on the this model, we will now introduce the transactional guarantees that we support when dealing with transactional workflows.

4 Transactional Model

Blocking of resources is contra productive in the environment of loosely coupled services. In order to avoid blocking situations, different notions of relaxed atomicity e.g., *semantic atomicity* [10] and *semi-atomicity* [19], 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 [19], 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 non-compensatable 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. Through specifying the workflow with accepted termination states (ATS), the designer implicitly defines representational sets of services whose completion represents the successful execution of the workflow. Note that multiple sets exists, as alternatives or multiple ATS may exist, e.g. *PayCC* or *PayCh* in Figure 1. We aim at semi-atomic execution of the workflow, which is according to the presented transactional properties in Section 3.4 defined as follows.

Semi-Atomicity Semi-Atomicity of a composite service whose execution is represented by a workflow with defined ATS is ensured if

- either all services belonging to one valid execution path to an ATS are completed and all services which are not included on that path and demand consistent completion are not completed

– or no service demanding consistent completion is completed.

This relaxes semi-atomicity as defined for flexible transactions as it disregards backward-recovery for services which may complete inconsistently.

5 Effects of Transactional Properties

In this section, we present the general effects, that transactional service properties take on the workflow execution.

Execution Order and Type Initially, the **order** of the execution is given through the workflow pattern itself. However, depending on the services involved, this given order endangers the semi-atomic execution as non-curable failures might occur. Depending on the non-functional properties of the concrete services, the order within a pattern can be *sequential* (i.e. S_i before S_j), *parallel* or *in any order* (i.e. S_i before S_j or S_j before S_i but not parallel). The execution **type** is either *independent* thus, no coordination has to take place, or *coordinated*, indicating, that their execution has to be coordinated in a sub-transaction (e.g., using 2PC) to guarantee that either all or none of them are completed. Generally, we aim at independent execution trying to avoid blocking situations.

Recovery Mechanism When backward-recovery is pursued, the workflow management system is in charge to take the appropriate measures. Those measures are determined through transactional properties of the executed services which indicate which services have to be compensated.

Transactional Pattern Property In order to analyze the *whole* workflow, we group services being included in one pattern (indicated by dashed lines in Figure 1) and regard the transactional properties of the pattern. Its properties are derived by the properties of the included services.

Let $WP(S)$ be a workflow pattern, including the set of elements S.[3] The transactional properties of the pattern are denoted just as the transactional service properties: $WP(S).comp$, $WP(S).consCompl$ and $WP(S).redo$. According to the transactional properties of included services, we introduce the derived property of *c-compensatability*:

A pattern is regarded to be *c-compensatable* (denoted as $WP(S).cComp = 1$), if all of the executed services which demand consistent completion are compensatable and all contained patterns are c-compensatable. This states, whether a pattern is backward-recoverable.

6 Analysis of Workflow Patterns

In this section present our results of the analysis of transactional service properties within the AND-pattern and the XOR-pattern.[4]

[3] Elements of S are either services or contained patterns.

[4] Due to space limitations, we omit the analysis results for other common patterns, such as Sequence, OR and N-OUT-OF-M.

6.1 Effects on the AND-pattern

All services aggregated in an AND-pattern have to be completed in order activate the subsequent workflow. Thus, the properties of *all* services are important.

Initially, the execution of the included services is decoupled. If services registered at runtime expose the following transactional properties, than their execution has (as opposed to originally intended) be *ordered*[5] to preserve semi-atomic execution. Recall, that the properties of a service S, are denoted as:

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

If within the AND-pattern, there exists

1. at least one non-redoable service S_i with $P_{Si} = (*, *, 0)$ and one service S_j with $P_{Sj} = (0, 1, 1)$,
2. or if there is at least one service S_i with $P_{Si} = (*, 0, 0)$ and one non-compensatable service S_j with $P_{Sj} = (0, 1, *)$,
3. or if there is at least one compensatable, non-redoable service S_i with $P_{Si} = (1, *, 0)$ and one non-compensatable service with $P_{Sj} = (0, 1, *)$

then, the execution of S_i has to precede the execution of S_j ($S_i \rightarrow S_j$). 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 included services S_i and S_j are non-compensatable, need consistent completion and are not redoable: $P_{Si} = P_{Sj} = (0, 1, 0)$ then their execution has to be *coordinated* within a subtransaction. Otherwise, in case of failure of one and completion of the other, the workflow is in an inconsistent execution state which cannot be recovered.

The *transactional property* of the AND-pattern is determined through the transactional properties of all included services. The AND-pattern WP_{AND}

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

If the pattern in c-compensatable, then all services which demand consistent completion are compensatable.

Consider again our example (Figure 1). Assume CRS and $Confirm$ are local services with $P_{CRS} = P_{Confirm} = (1, 1, 1)$. Additionally, a transportation service T with $P_T = (0, 1, 0)$ and ticket reservation R with $P_R = (0, 0, 0)$ are bound.

[5] Ordering prevents coordination in a sub-transaction (and thus blocking).

According to the previous analysis (case 2), the completion of R has to precede T to ensure semi-atomicity. Considering $P_{PayCC} = (1, 1, 0)$ and $P_{PayCh} = (1, 1, 1)$, the XOR-pattern is redoable, although $PayCC$ is not redoable.

The accommodation-service A is discovered at runtime. Consider the following alternatives: A_1 with $P_{A1} = (0, 1, 1)$, A_2 with $P_{A2} = (1, 1, 0)$ and A_3 with $P_{A3} = (0, 1, 0)$. Considering those, the following modifications of the workflow are performed: If A_1 is included, T and A_1 are aligned in sequence due to the first case of the ordering constraints of the AND-pattern (Figure 3.a). If A_2 registeres at runtime, then the execution of T and A_2 is parallelized, as shown in Figure 3.b. If service A_3 is included, then a sub-transaction coordinating T and A_3 will be necessary (as stated in the analysis) to preserve semi-atomicity .

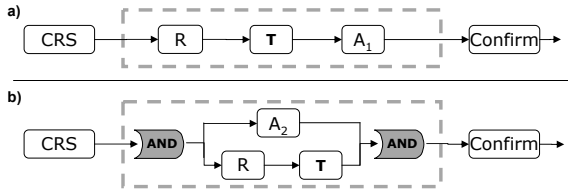

Fig. 3. Dynamic alteration of the workflow at runtime.

6.2 Effects on the XOR-pattern

As opposed to the AND-pattern, one and only one service of the XOR-pattern is completed. For the following analysis, we consider alternatives of services. I.e., we disregard situations, in which the choice is not dependent on transactional properties rather than other criteria.

The *type* and *order* of the XOR-pattern is not changed through dynamically bound services as only one service is executed at time. The *transactional pattern properties* are determined differently than for an 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.
  $WP_{XOR}(S).comp = 1 \Leftarrow \forall S_i \in S : S_i.comp = 1$
  $WP_{XOR}(S).comp = 0 \Leftarrow \forall S_i \in S : S_i.comp = 0$
- needs consistent completion, if all included services demand consistent completion. If none of the included services demands consistent completion, the pattern allows for inconsistent completion.
  $WP_{XOR}(S).consCompl = 1 \Leftarrow \forall S_i \in S : S_i.consCompl = 1$
  $WP_{XOR}(S).consCompl = 0 \Leftarrow \forall S_i \in S : S_i.consCompl = 0$
- is redoable, if at least one service is redoable. Else, it is non-redoable.
  $WP_{XOR}(S).redo = 1 \Leftrightarrow \exists S_i \in S : S_i.redo = 1$

– is c-compensatable, if all included services are either compensatable or allow for inconsistent completion.
$WP_{XOR}(S).cComp = 1 \Leftarrow \forall S_i \in S : S_i.Comp = 1 \vee S_i.consCompl = 0$
$WP_{XOR}(S).cComp = 0 \Leftarrow \forall S_i \in S : S_i.comp = 0 \wedge S_i.consCompl = 1$

The *recovery mechanisms* to be taken are determined through the executed service. If the pattern is c-compensatable, then it is backward-recoverable.

According to the workflow, transactional properties of services can be used to determine a preference relation on which service to include in the XOR-pattern. Consider for example Figure 4: At runtime, either branch S_i or S_j are to be taken within the XOR-pattern, before the subsequent workflow S_{subseq} is invoked. Let the transactional properties be $P_{Si} = (1, 1, 0)$ (compensatable, consistent completion, non-redoable) and $P_{Sj} = (0, 1, 1)$ (non-compensatable, consistent completion, redoable). According to the previous analysis, the XOR-pattern is thus redoable, as S_j is redoable. If $S_{prev}.comp = 1$ 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 $S_{prev}.comp = 0$ 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 assured through the presence of S_j (as $S_j.redo = 1$). Thus, in this case the choice between those two services does not rely on the transactional properties.

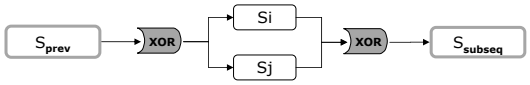

Fig. 4. Preference of S_i and S_j according to transactional properties.

7 Summary

In this paper, we presented the analysis of non-functional service properties in the presence of transactional workflow patterns in order to guarantee semi-atomic execution of workflows. We introduced the property of consistent completion which has not been considered for transactional workflow execution so far and adjusted the notion of semi-atomicity to support transactional workflow execution. By identifying structural adaptations of the workflow and illustrating the influence of transactional properties on the preference relation of services, we are able to preserve semi-atomicity if services with different transactional properties are discovered at runtime. As part of future work, we want to use this as a foundation for the design and verification of an adaptive workflow engine, which exploits transactional properties of services to automate transactional execution, including dynamical changes at runtime, automated failure handling mechanisms and transactional preference relations.

References

1. ActiveBPEL Engine. http://www.activebpel.org.
2. P. C. Attie, M. P. Singh, A. Sheth, and M. Rusinkiewicz. Specifying and Enforcing Intertask Dependencies. In *Proceedings of the 19th VLDB Conference*, 1993.
3. T. Bultan, X. Fu, et al.. 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*, New York, NY, USA, 2003. ACM.
4. D. Chakraborty, A. Joshi, et al.. Toward Distributed Service Discovery in Pervasive Computing Environments. *IEEE Transactions on Mobile Computing*, 2006.
5. H. Davulcu, M. Kifer, et al.. Logic Based Modeling and Analysis of Workflows (Extended Abstract). In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ACM Press, 1996.
6. M. Dean, D. Connolly. et al.. Web Ontology Language (OWL) Reference Version 1.0, 2002. http://www.w3.org/TR/2002/WD-owl-ref-20021112.
7. D. Dyachuk and R. Deters. Service Level Agreement Aware Workflow Scheduling. In *Proceedings of International Conference on Services Computing (SCC)*, 2007.
8. M. Fauvet, H. Duarte, M. Dumas, and B. Benatallah. Handling transactional properties in web service composition. In *WISE*, pages 273–289, 2005.
9. W. Gaaloul, M. Rouached, C. Godart, and M. Hauswirth. Verifying composite service transactional behavior using event calculus. In *Proceedings of the 14th International Conference on Cooperative Information Systems (COOPIS'2007)*, 2007.
10. H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions Database Syst.*, 8(2):186–213, 1983.
11. A. Gupta, N. Gupta, K. Ghosh, and M. M. Gore. Team Transaction: A New Transaction Model for Mobile Ad Hoc Networks. In *ICDCIT*, 2004.
12. Hamadi and Benatallah. A petri net-based model for web service composition. In *Proceedings of the 14th Australasian Database Conference (ADC'03)*, 2003.
13. WebServices AtomicTransaction, 2005. http://www.ibm.com/developerworks/library/specification/ws-tx/.
14. S. Jajodia and L. Kerschberg, editors. *Advanced Transaction Models and Architectures*. Kluwer, 1997.
15. 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.
16. 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 (VLDB 00)*, San Francisco, CA, USA, 2000.
17. M. Schäfer, P. Dolog, and W. Nejdl. Engineering Compensations in Web Service Environment. In *ICWE*, pages 32–46, 2007.
18. W. van der Aalst, P. Barthelmess, C. 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.
19. A. Zhang, M. Nodine, B. Bhargava, and O. Bukhreset. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. *SIGMOD Rec.*, 23(2), 1994.