WS-SAGAS: Transaction Model for Reliable Web-Services-Composition Specification and Execution

Neila BEN LAKHAL[♠] Takashi KOBAYASHI[♦]
Haruo YOKOTA[♥]

Recently, building reliable web services compositions has triggered extensive research efforts. Considering that web services tend to be frequently updated or even to disappear unexpectedly, composition may fail easily causing reliability decrease. To challenge reliability, in this paper we propose WS-SAGAS, a transaction model for a reliable specification of web services composition. A composition is modeled as a hierarchy of arbitrary nested transactions, executed in a distributed architecture, with proper failure detection and recovery mechanisms.

1. Introduction

Nowadays business processes are typically running within a collection of largely distributed and loosely coupled computing environments. Generally, such business processes need to be continually reconsidered to fit to process changes. To cope with such environment requirements, wide range of solutions were proposed. Unfortunately, many of them have several limitations. They mainly lack appropriate supports for correctness and reliability enhancement in the presence of failure. Addressing those issues is difficult because of the absence of standards application.

Later on, with the emergence of XML-based standards, followed by web services, researchers realized that those standards are strong enough to support such requirements. As a consequence, the trend is towards deploying business processes by connecting elementary web services. As an example of such trends, Open Grid Services Architecture (OGSA) is proposed [5]. Despite these standards help considerably to enhance interoperability, reliability is not yet well addressed. In fact, with web services environment volatility and dynamism, it is most likely to happen that a component service is updated or moreover it disappears suddenly. In such situation, it is necessary to provide proper failures detection and recovery mechanisms. This is fundamental to avoid overall composition consistency review.

To achieve these requirements, augmenting web services composition specification with the transaction concept, as already revealed in other areas, seems to be adequate. Nevertheless, considering that web services are naturally distributed i.e. hosted by different web services providers, relying on these providers to support transactions is not feasible. Thus, it becomes obvious that defining an accurate transaction model valid for the whole composition is essential.

However there are many advanced transaction models [2]. Applying directly already proposed models is not acceptable because of web services particularity, compared with usual software components.

Motivated with these concerns, in this paper we propose WS-SAGAS, a new transaction model. Specifically WS-SAGAS extends nested-sagas model [1] and enriches it with "State" feature. State capturing allows coordinating primarily autonomous web services in a composition and helps to inform about web services composition potential execution progress. Moreover, in case of potential failure occurrence, it will allow to detect it and indeed to recover. Besides, in WS-SAGAS, we inherit also the "vitality degree" from other models such as ConTract and Open Nested [2]. We expect the vitality degree to reduce considerably failure possibilities and indeed increase composition availability. We justify this as follow. Since originally a transaction succeeds only if all its components are successful, with vitality degree introduction, only vital components success is required.

The remainder of the paper is organized as follows: Section 2 gives an overview of WS-SAGAS transaction model. Section 3 discusses the execution model of WS-SAGAS in a distributed architecture, and explains it in an illustrative example. Finally, Section 4 concludes the paper and provides some remarks concerning future works.

2. WS-Sagas Transaction Model

2.1 Transaction Paradigm Applicability

Considering that web services tend to be frequently updated or even to disappear unexpectedly, composition may fail easily, causing reliability significant decrease. This makes web services rather different from usual software components. Consequently, web services warrant a particular transaction support especially shaped for them. We discuss in what follow what kind of properties a transaction needs to satisfy to fit to web services context.

A traditional transaction is supposed to support fully ACID properties, which is not acceptable for web services. Our justifications are as follow. First, "Atomicity" property full support is not required. Instead of deducing a whole composition failure when one of its components fails, it is more profitable to soften "Atomicity", in other words, to take advantage of the transaction support that services might formerly encompass e.g. compensation. Moreover, it would be more efficient to choose another service, which supports the same semantic. Actually, there is a wide range of semantically equivalent web services enabled to provide same functionalities in different ways. Second, similarly "Isolation" and "Consistency" properties should be relaxed because none of them is relevant. Since compositions might be long running, enforcing "Isolation" affects negatively execution progress. This is because parallel transactions communication is required in web services context, since cooperation among transactions is an essential feature. Besides, ensuring "Consistency" enforcement means first, monitoring each web service invocation and later, identifying the service(s), component

---

[♠] Student Member Graduate School of Information Science and Engineering, Tokyo Institute of Technology
neila@de.cs.titech.ac.jp

[♦] Regular Member Global Scientific Information and Computing Center tkobaya@cs.titech.ac.jp

[♥] Regular Member Tokyo Institute of Technology, Global Scientific Information and Computing Center
yokota@cs.titech.ac.jp

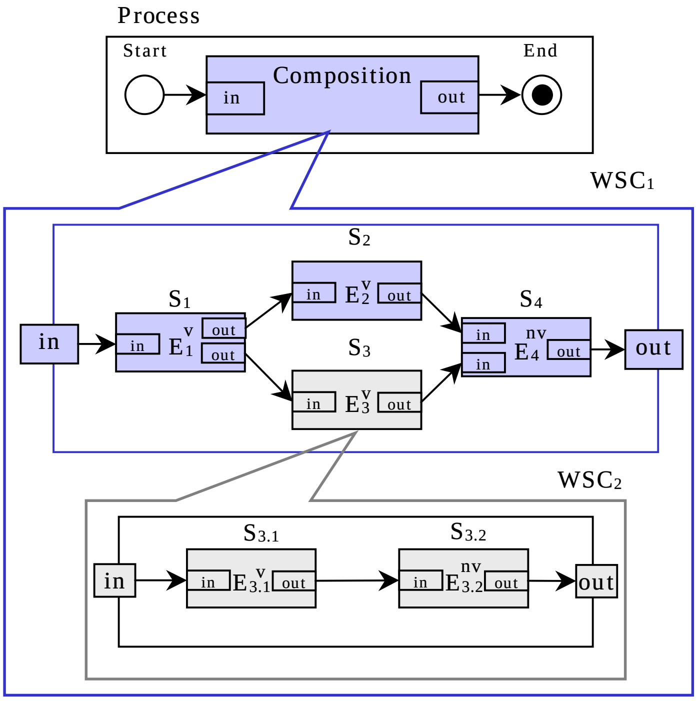

Figure1. WS-SAGAS Description

from a composition, that might violate the whole composition consistency. This is rather tedious and not acceptable. Finally "Durability" property should be kept because once a composition completes, its execution results must be made persistent. As a result, the traditional transaction model, with ACID properties full-support, could not straightforwardly apply to web services context but it needs to be extended.

2.2 Nested-Sagas Transaction Model

There are several advanced transaction models proposed [2]. Seeing web services context requirements, we investigated the applicability of the nested-transaction model [3], sagas model and finally nested-sagas model [1]. In what follow a comparison between those three models:

1. Nested-sagas transactions can be recursively defined. Thus they support an arbitrary level of nesting contrary to the original sagas transactions, where nesting is limited to two levels;
2. The original nested transaction model of [3] ensures atomicity and isolation of the whole transaction. A sub-transaction failure is reflected on its parent. This is not conceivable because considerable amount of already executed works would be lost;
3. Nested-sagas transaction model considers about communication mechanisms, an essential feature in web service context. Each saga specifies input and output ports bound at run time to mailboxes i.e., messages queue.
4. Nested-sagas with already specified input and output ports can be more practically mapped to web services since structures are somehow identical;
5. Finally, compensating transaction provides much more flexibility. Since it was primarily proposed in sagas transaction model, it is worthy to build on nested-sagas instead of the nested-transaction model.

Guided by this comparison, we propose to inherit features of interest from the nested-sagas model. Specifically, arbitrary nesting level, relaxed ACID properties and transaction compensation. We also inherit the vitality degree feature, proposed in several advanced transaction models [2]. Moreover, in order to satisfy properly the transaction support described in 2.1, we also propose to enrich it with state capturing feature that we will describe in the following subsection.

2.3 WS-Saga Description

A ws-composition WSC is a collection of n elements from a composition { E_1^v, E_2^v,…E_n^v }. As depicted in Figure 1., a composition is specified as an orchestration of elements. Depending on the considered nesting level, the same element E_i^v is either assimilated to an atomic element or to a ws-composition e.g. E_3^v is assimilated to an atomic element in WSC₁ specification while in WSC₂ specification, it is composed of two elements $E_{3.1}^v$ and $E_{3.2}^{nv}$.

An element has a state Sᵢ and a vitality degree, where "v" in E_i^v stands for Vital element and "nv" in E_i^{nv} stands for Not Vital element.

Definition I Element state Sᵢ

An atomic element is exclusively in one of the following states:

1. *Waiting*: If element E_i^v is not yet submitted for execution and is waiting to;
2. *Executing*: If element E_i^v is executing;
3. Failed: If element E_i^v encounters a failure;
4. *Aborted*: If element E_i^v receives a request to abort itself;
5. *Committed*: If element E_i^v has successfully terminated and was committed and
6. *Compensated*: If element E_i^v has been compensated.

An element execution is actually the execution of a web service providing functionalities of interest. This service execution control is delegated to an engine eᵢ, already allocated to the considered element. State change, as described in Figure 2., is performed by that engine eᵢ. The state concept introduction is motivated with the following concerns:

1. Since web services are originally without state, when they are executing as component of the same composition, without the state concept introduction, it will not be possible to know the execution progress.
2. In order to decide how to go forward in a WSC execution i.e. decide to which element(s) to delegate the execution control or whether to resume the execution, it is essential to know the execution progress of elements being executed.

Definition II Element Vitality Degree

We introduce the vitality degree of an element E_i^v in order to add flexibility in the way ws-composition failure is cascaded. We distinguish a vital element E_i^v from a

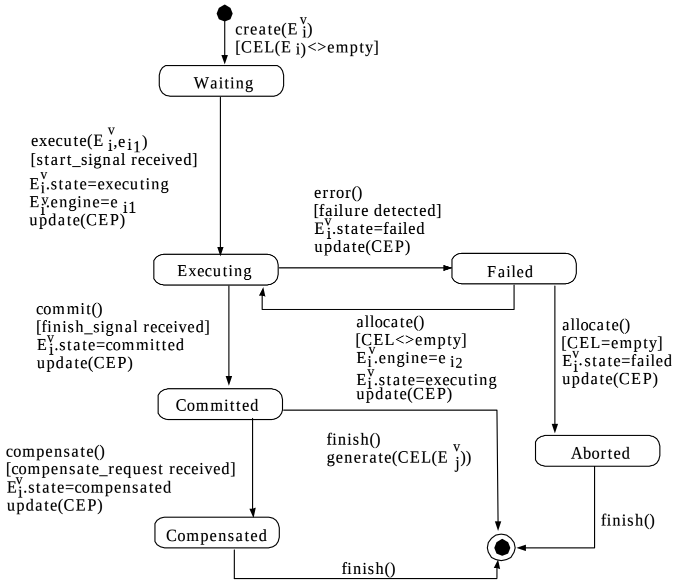

Figure 2. Element State Transition Diagram

not-vital element E_i^{nv} as follow:

1. Aborting a vital element component of a ws-composition will induce aborting the whole ws-composition, if there is no alternative web service to execute the same element.
2. Aborting a not-vital element will not be reflected on its parent, thus a ws-composition can complete successfully even though not all its components elements were committed.
3. Initially the vitality degree of all elements is by default set to vital.

3. WS-SAGAS Transactions Execution in THROWS Distributed Architecture

3.1 THROWS Distributed Architecture Overview

In this section we describe how web services composition, specified as WS-SAGAS transactions, will be executed in a distributed architecture that we already proposed and named THROWS. More details about THROWS architecture are available in [4].

THROWS stands for “a Transaction Hierarchy for Route Organization of Web Services”. Specifically THROWS is a distributed architecture for web services composition reliable execution where the control is hierarchically delegated among distributed engines dynamically discovered, during the composition execution. These distributed engines interact in a peer-to-peer way.

Each engine e_i is responsible of the successful execution of an element E_i^v that is, the execution of an available web service WS_i. WS_i is offering the same functionalities, as element E_i^v requires.

THROWS achieves failure capturing and recovery from failure by the “Candidate Engines List (CEL)” concept and the “Current Execution Progress (CEP)” concept. CEP and CEL are to be available on each engine side. CEL is relative to an element from a composition. It is the list of candidate engines enabled to execute it i.e. they control the execution of web services providing semantics desired by that element.

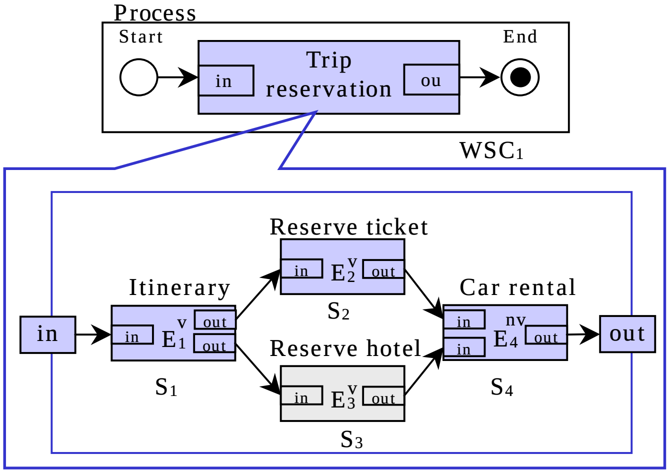

Figure 3. Trip Reservation Business Process

CEL concept reinforces the composition reliability by providing alternative execution routes for a same WS-SAGAS transaction. Besides CEP keeps track of a composition execution progress and allows execution control delegation among engines. An engine, while executing an element, every change in the element, as described in Figure 2., has to be also reflected on CEP.

3.2 Illustrative Example

To ensure a better understanding of how web services compositions are depicted as WS-SAGAS transactions, we describe in what follow the case of a trip reservation business process. To reserve a trip, a customer needs to submit an itinerary that indicates desired destination, departure and arrival time and date. Airplane ticket and hotel reservation are crucial for the whole process success. Car rental is considered as optional. By using WS-SAGAS transaction model description, the trip reservation business process will be specified as in Figure.3. Initially, all the elements state is set to “waiting”, with no engine allocated. The CEP of WSC_1 is indicated as follow:

- $CEP\ (WSC_1)_{initial} = \{(E_1^v, waiting, null), [(E_2^v, waiting, null); (E_3^v, waiting, null)], (E_4^{nv}, waiting, null)\}$

The order of execution is $E_1^v<[E_2^v; E_3^v]<E_4^{nv}$, i.e. when element E_1^v is committed successfully, elements E_2^v and E_3^v executions are concurrently launched. To start executing element E_4^{nv}, E_2^v and E_3^v successful completions are necessary because both of them are vital. Contrary to E_4^{nv}, considered as not vital i.e. if airplane ticket and hotel reservation were successful but car rental failed, the whole composition success can be deduced.

Suppose e_{11} is allocated to execute E_1^v, i.e. to execute a web service WS_{11} providing the same semantic required by E_1^v. If engine e_{11} executed successfully WS_{11}, E_1^v state is updated to “committed”, e_{11} will check in CEP whether there is following elements, it will generate CEL of E_2^v and E_3^v, as successors that need to be executed in parallel, and allocate e_{21} and e_{31} to execute respectively web services WS_{21} and WS_{31}. Finally it will update CEP as follows, and communicate it to e_{21} and e_{31}.

- CEP $(WSC_1)_{11\rightarrow 21,31}$={($E_1^v$, committed, e_{11}), [(E_2^v, waiting, e_{21}); (E_3^v, waiting, e_{31})], (E_4^{nv}, waiting, null)}

Suppose that, while engine e_{21} was executing WS_{21}, a failure occurs for any unpredicted reason such as WS_{21} unavailability e.g. this may occur since web services are hosted on the web services providers side. Failure recovery is performed as follows. e_{21} will inform its predecessor engine(s), here e_{11}, that it was unable to terminate successfully E_2 execution i.e. will communicate to engine e_{11} the CEP updated as follows:

- CEP $(WSC_1)_{21\rightarrow 11}$ = {(E_1^v, committed, e_{11}), [(E_2^v, failed, e_{21}); (E_3^v, executing, e_{31})], (E_4^{nv}, waiting, null)}

To avoid the whole composition failure, since element E_2^v execution is crucial for its success, i.e. E_2^v is a vital element; e_{11} checks the content of CEL (E_2^v) that it has already generated. Two cases are conceivable:

- *Case 1:* CEL (E_2^v)={e_{22}, e_{23}, e_{24}}

Engine e_{11} allocates e_{22} to execute E_2^v. It also updates CEP as follows and communicates it to engines e_{22} and e_{31}. As result, the execution retrial is enabled and failure is avoided.

- CEP$(WSC_1)_{11\rightarrow 22,31}$= {($E_1^v$, committed, e_{11}), [(E_2^v, waiting, e_{22}); (E_3^v, executing, e_{31})], (E_4^{nv}, waiting, null)}
- *Case 2:* CEL (E_2^v)=empty

Engine e_1 while checking CEL (E_2^v) content, it finds out that there are no other candidate engines, enabled to execute element E_2^v. As result, WSC_1 execution will be resumed. Executing elements will be requested to abort and already committed elements will be compensated for i.e. E_1^v will be compensated for by engine e_{11}, element E_3^v will be aborted by engine e_{31} and the executed part from E_2^v before failure, will be compensated for.

4. Conclusion

In this paper, we proposed WS-SAGAS, a transaction model for a reliable specification of web services composition. In actual fact, concerned with web services environment volatility and dynamism, an also with the fact that web services are in essence naturally hosted by independent web services providers, we realized that, to enhance reliability, applying straightforwardly already proposed transaction models is not possible. A comparative study of several advanced transactions models guided us to the nested-sagas transaction model, from which we inherited several interesting features. Specifically, arbitrary nesting level, relaxed ACID properties and transaction compensation.

Besides, we inherited also the vitality degree notion. It allowed us increasing the availability of composition and indeed to reduce failures possibilities. Moreover, we enriched it with state capturing. This allowed us to enhance considerably composition reliability. In fact state capturing enabled us to know a composition potential execution progress and to detect failures.

In addition, we also described how web services composition, modelled as WS-SAGAS transactions, can be executed in a distributed architecture THROWS. We have taken a trip reservation business process as an illustrative example. In THROWS, WS-SAGAS transactions reliable execution is ensured using CEL and CEP concepts. Transactions are executed in a distributed way; each transaction element is mapped to a web service, allocated already to a distributed engine. Such execution model allowed us to improve noticeably the system performance. Moreover, WS-SAGAS transaction model is rich enough to be applied in other kind of architecture i.e. with centralized control.

As future work, we are currently investigating implementation issues and specifically the feasibility of web services composition implementation using WS-SAGAS transaction model.

[References]

[1] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem, Modeling long-running activities as nested sagas. *Data Engineering Bulletin, vol.14, no.1,* pp. 14-18. March.1991.

[2] A.k.Elmagarmid, ed., Database transaction models for advanced applications, Morgan Kaufmann Publishers, San Mateo, California, 1992.

[3] J.E.B. Moss, Nested transactions: an approach to reliable distributed computing, " *MIT Press, Cambridge, Massachusetts,* 1985.

[4] Neila BEN LAKHAL, Takashi KOBAYASHI and Haruo YOKOTA, Distributed architecture for reliable execution of web services, *Technical Report of IEICE,* DBWS2003 2B, DE2003-24, pp.97-102, 2003-DBS-131 (17), pp.129-136, 2003.7.

[5] OGSA home page, http://www.globus.org/ogsa/

Neila BEN LAKHAL

She received a BS in computer science applied to management from High Institute of Management of Tunis Tunisia in 2000. She is currently a Master course student in the Graduate School of Information Science and Engineering of Tokyo Institute of Technology. She has been working on transaction processing and distributed workflow systems failure recovery.

Takashi KOBAYASHI

He received a B.E. and M.E. in computer science from Tokyo Institute of Technology in 1997 and 1999, respectively. He is a research associate of Global Scientific Information and Computing Center, Tokyo Institute of Technology. His research interests include software patterns, software architecture and information retrieval. He is a member of IPSJ and JSSST.

Haruo YOKOTA

He received the B.E., M.E., and Dr.Eng. degrees from Tokyo Institute of Technology in 1980, 1982, and 1991, respectively. He joined Fujitsu Ltd. in 1982, and was a researcher at ICOT for the Japanese 5th Generation Computer Project from 1982 to 1986, and at Fujitsu Laboratories Ltd. from 1986 to 1992. From 1992 to 1998, he was an Associate Professor in Japan Advanced Institute of Science and Technology (JAIST). He is currently a Professor at Global Scientific Information and Computing Center in Tokyo Institute of Technology. His research interests include general research area of data engineering, information storage systems, and dependable computing. He is a chair of Technical Group of Data Engineering in IEICE and a member of IPSJ, JSAI, IEEE, IEEE-CS, ACM and ACM-SIGMOD.