T2R2 東京科学大学 リサーチリポジトリ
Science Tokyo Research Repository

論文 / 著書情報
Article / Book Information

Title	FENECIA: failure endurable nested-transaction based execution of compositeWeb services with incorporated state analysis
Author	Neila Ben Lakhal, Takashi Kobayashi, Haruo Yokota
Journal/Book name	The VLDB Journal, Vol. 18, No. 1, pp. 1-56
発行日 / Issue date	2009, 1
URL	http://www.springerlink.com/content/g546851225w35404/
権利情報 / Copyright	The original publication is available at www.springerlink.com.

Powered by T2R2 (Science Tokyo Research Repository)

The VLDB Journal manuscript No.
(will be inserted by the editor)

Neila BEN LAKHAL · Takashi KOBAYASHI · Haruo YOKOTA

FENECIA: Failure Endurable Nested-transaction based Execution of Composite Web Services with Incorporated State Analysis

the date of receipt and acceptance should be inserted later

Abstract Interest in the Web services (WS) composition (WSC) paradigm is increasing tremendously. A real shift in distributed computing history is expected to occur when the dream of implementing Service-Oriented Architecture (SOA) is realized. However, there is a long way to go to achieve such an ambitious goal. In this paper, we support the idea that, when challenging the WSC issue, the earlier that the inevitability of failures is recognized and proper failure-handling mechanisms are defined, from the very early stage of the composite WS (CWS) specification, the greater are the chances of achieving a significant gain in dependability. To formalize this vision, we present the *FENECIA (Failure Endurable Nested-transaction based Execution of Composite Web services with Incorporated state Analysis)* framework. Our framework approaches the WSC issue from different points of view to guarantee a high level of dependability. In particular, it aims at being simultaneously a failure-handling-devoted CWS specification, execution, and quality of service (QoS) assessment approach. In the first section of our framework, we focus on answering the need for a specification model tailored for the WS architecture. To this end, we introduce *WS-SAGAS*, a new transaction model. WS-SAGAS introduces key concepts that are not part of the WS architecture pillars, namely, *arbitrary nesting*, *state*, *vitality degree*, and *compensation*, to specify failure-endurable CWS as a hierarchy of recursively nested transactions. In addition, to define the CWS execution semantics, without suffering from the hindrance of an XML-based notation, we describe a textual notation that describes a WSC in terms of *definition rules*, *composability rules*, and *ordering rules*, and we introduce graphical and formal notations. These rules provide the solid foundation needed to formulate the execution semantics of a CWS in terms of *execution correctness verification dependencies*. To ensure dependable execution of the CWS, we present in the second section of FENECIA our *architecture THROWS*, in which the execution control of the resulting CWS is distributed among engines, discovered dynamically, that communicate in a peer-to-peer fashion. A dependable execution is guaranteed in THROWS by keeping track of the execution progress of a CWS and by enforcing forward and backward recovery. We concentrate in the third section of our approach on showing how the failure consideration is trivial in acquiring more accurate CWS QoS estimations. We propose a model that assesses several QoS properties of CWS, which are specified as WS-SAGAS transactions and executed in THROWS. We validate our proposal and show its feasibility and broad applicability by describing an implemented prototype and a case study.

Keywords Web services · composition · dependability · failure · distributed execution · transaction model · QoS

Neila BEN LAKHAL
Tokyo Institute of Technology, Department of Computer Science
2-12-1 Oh-Okayama, Meguro-ku Tokyo, 152-8552 JAPAN
Tel.: +81-3-5734-3505, Fax: +81-3-5734-3504
E-mail: neila@de.cs.titech.ac.jp

Takashi KOBAYASHI · Haruo YOKOTA
Global Scientific Information and Computing Center
Tokyo Institute of Technology, Department of Computer Science
2-12-1 Oh-Okayama, Meguro-ku Tokyo, 152-8552 JAPAN
Tel.: +81-3-5734-3505, Fax: +81-3-5734-3504
E-mail: {tkobaya,yokota}@cs.titech.ac.jp

1 Introduction

With the current proliferation of Web services (WS), a considerable shift is expected to occur in the way distributed computing systems are integrated. The conventionally integrated systems are foreseen to be gradually replaced in the near future by distributed and loosely coupled services-oriented systems. The key features that allow the WS technology to accomplish such a shift are: *a*) It builds on a set of universally recognized XML standards, especially WSDL (Web Service Description Language) [1], SOAP (Simple Object Access Protocol) [2], and UDDI (Uniform Description Discovery and Integration) [3] to describe, discover, and invoke any type of services in a networked environment. *b*) It has the potential to glue any systems together, no matter how different they are. *c*) It reduces dependency among components to obtain less fragile systems with increased responsiveness and ability to be frequently modified.

One issue that is gaining notable momentum in the research community is WS composition (WSC), which is used to create what is called *value-added services* or *composite Web services (CWS)* by taking a set of preexisting elementary WS, typically owned and managed by diverse entities, and weaving them together to build more powerful and feature-rich business processes. An example of CWS is an application that books a flight, rents a car, and makes a hotel reservation to provide a complete trip reservation process.

There is a myriad of specifications available for composing WS, exemplified by the emerging standards such as BPEL4WS (Business Process Execution Language for Web Services) [4] and industrial solutions such as IBM's Emerging Technologies Toolkit ETTK [5] and Microsoft's .Net [6]. In addition, academic researchers are making substantial research efforts, working on a whole panoply of WSC strategies including dynamic composition (e.g., eFlow [7, 8]), declarative composition (e.g., SELF-SERV [9,10]), and semantic composition (e.g., SHOP2 [11]). A careful investigation of the major part of the available solutions for WSC reveals that only a very few cases are geared toward a distributed environment, such as the SELF-SERV framework. However, all the other approaches, such as BPEL4WS and eFlow, only support the integration of WS into a centralized model consisting of dedicated centralized engine(s). They have totally ignored the nature of the WS environment where interaction follows a peer-to-peer model and where each peer WS owner provides a set of services that can be used to compose a CWS.

The WSC technology is still regarded as immature: it requires considerable development before reaching its apogee [12,13]. In particular, the WSC technology has to overcome a major obstacle—the widely recognized unreliability of the Internet—because all the available WS rely heavily on the Internet to be deployed. Adding to the Internet unreliability is a whole set of characteristics of the modern computing environments (e.g., unpredictability, heterogeneity, autonomy, dynamism, complexity, etc.) in which the WS subsist, making the most unexpected failure a normal part of any WS. Furthermore, with the assembly of several elementary WS into a CWS to create richer functionalities, the failure frequency is more important than ever.

In this paper we advocate that, when challenging the WSC issue, the earlier we accept the inevitability of failures and make available proper failure-handling mechanisms—from the very early stage of the CWS design—the greater are the chances of achieving a significant gain in dependability. To formalize this vision, we propose *FENECIA (Failure Endurable Nested-transaction based Execution of Composite web services with Incorporated state Analysis)*, in which we tackle the WSC issue from different viewpoints to guarantee a higher level of dependability. Our approach aims at being, simultaneously, a failure-handling-devoted CWS specification, execution, and QoS assessment approach. Our framework is depicted in (Figure 7.1) and its contributions are threefold:

WS-SAGAS: a CWS specification model. The first section of our approach tackles the WSC issue from a different viewpoint: instead of trying to avoid failures, we accept their inevitability and we propose a new CWS specification model that builds primarily on the *transaction* concept—widely recognized by the database community as a strong concept for enhancing reliability and availability [14]. Specifically, we present a new *transaction model* that we name *WS-SAGAS* [15–18] to capture the underpinning logic of the CWS in transactions. Our model is specifically tailored to fit the characteristics of the WS architecture, thereby allowing to overcome the constraints imposed by the traditional transaction model [14]. In particular, WS-SAGAS specifies the CWS as a *hierarchy of arbitrary nested transactions* and introduces key features including *state capture*, *vitality degree*, and *compensation* mechanisms. These mechanisms are critical to inform of and recover from any transient failure. We build on these concepts to specify failure-endurable CWS as a hierarchy of recursively nested transactions. In addition, to define the CWS execution semantics without suffering from the hindrance of an XML-based notation, we describe a WSC in terms of *Definition Rules (DR)*, *Composability Rules (CR)*, and *Ordering Rules (OR)*, and we introduce graphical and formal notations. These rules provide the solid foundation required to formulate the execution semantics of a CWS in terms of *execution correctness verification dependencies*.

THROWS: a CWS execution architecture. In the second section of FENECIA, we propose a new architecture, named *THROWS (Transaction Hierarchy for Route Organization of Web Services)*, for a highly dependable distributed execution of CWS. In THROWS [19,20], CWS execution control is hierarchically delegated among distributed *engines*; these engines are *discovered dynamically* throughout the CWS execution progress and they interact in a *peer-to-peer* fashion, thereby avoiding WS execution dependence on a single authority, which can constitute a potential single point of failure. In THROWS, we achieve failure capture and recovery, and control of long-running and parallel transactions by introducing two key concepts: the *Candidate Engines List (CEL)* and the *Current Execution Progress (CEP)*.

QoS estimation and analysis model. In the third section of FENECIA, we focus on another issue related to the qualitative aspect of CWS: we verify to what extent the failure-handling mechanisms we propose are sufficiently strong to achieve a significant gain in dependability. during execution. We present a novel model that *characterizes*, *estimates*, and *analyzes* several QoS properties of dynamically executed CWS [21,17]. In particular, we estimate the *reliability* and the *execution time* of the CWS. We concentrate on one important issue that has received little attention to date, that is, considering the *potential failures repercussions* on the CWS execution performance estimates. We advocate that accounting for failures and their repercussions on the effective performance of the CWS is particularly required in the WS architecture, in view of the WS inherent tendency to fail relatively easily (compared to other computing components).

Contrary to most of the current approaches dealing with QoS estimations in the WS context, which rely on the QoS information advertised by the WS providers, our model computes QoS estimates on the basis of the CWS execution *observation*.

Approach validation. To check the feasibility of our approach, we present a prototype that we implemented [18,22] and that specifies CWS as a hierarchy of recursively nested WS-SAGAS transactions and simulates their execution in THROWS architecture. In addition, we report a case study that demonstrates the applicability of our proposal [21].

By bringing together the sections described above: *i*) We build on the strength of the WS architecture-enabling standards. *ii*) We combine a number of carefully selected features: the transaction-based specification and execution, the state-guided execution failure monitoring, the failure-aware QoS estimation, and the execution observation-driven QoS analysis. *iii*) Finally, we introduce the dedicated failure handling and recovery strategy, and we provide a solid foundation for the FENECIA approach to become a comprehensive methodology for the development of highly dependable CWS .

The remainder of this paper is organized as follows. Section 2 describes the type of failures we consider in this paper. Section 3 describes the key requirements that a transaction model for the WS context must satisfy. Section 4 is an overview of the evolution of the transaction concept. Section 5 describes our WS-SAGAS transaction model. Section 6 describes our architecture, THROWS. Section 7 introduces our QoS model. Section 8 describes our validation and checks the applicability of our proposal. Section 9 describes related work. Finally, Section 10 concludes our paper and gives a few tentative suggestions for future work.

2 Fault Model

The fault model and the failure modes we identified were inspired by a failure taxonomy for the particular case of WS architecture developed in [23], which in turn is based on the seminal work of [24].

A fault model is a model of the types of faults that can occur in a system while it is running. The widely recognized specificities of the modern IT environment in which WS subsist (e.g., heterogeneity, complexity, and autonomy of the participating systems and of their underlying platforms, versatile communications protocols and dynamic management policies, uncertainties about system boundaries, etc.) make the system subject to all the classes of faults categorized in [24]. The classes of faults of interest are physical faults including all fault classes that affect hardware, interaction faults including all external faults, and development faults including all fault classes occurring during development. However, in multitier CWS, which span multiple interacting systems, interaction faults, which occur during use, have the greatest impact. Examples of faults in this class are lost/corrupted messages, process crashes, and faults introduced by updates. The interaction faults can be categorized as transient faults or permanent faults. We consider permanent faults beyond the scope of this paper.

2.1 Failure Modes

A complete understanding of possible failure modes helps determine the mechanisms for fault tolerance. In this paper, we consider failure modes encountered by the system users, specifically timing-related failures where the time of arrival, or the duration of the information delivered, at the service deviates from the expected duration implemented by the system function. These failures are environment-related failures and are associated with WS crashes and timeouts; they are commonly characterized as silent failures because the system service is no longer available to users. At the composition level, special monitoring is required to handle these failures. This class of failure is handled by performing either a forward recovery or a backward recovery without requiring any external intervention.

The other category of failures encountered by system users is content-related failures, such as WS execution exceptions, WS programmed exceptions, exceptions propagated from other participant WS, and fault messages received from SOAP calls to WS. We consider content-related failures to be beyond the scope of this paper; dealing with them is complex because WS providers define WS differently.

3 A Transaction Model for WS Context: Key Requirements

We identify the key requirements that a transaction model for CWS must satisfy. A number of contributions have added a transactional support for CWS such as WS-transaction [25] and WS-CAF [26]. Although available solutions are mostly for statically composed WS, we target a dynamic CWS. After identifying the different requirements that a transaction model for dynamic CWS must satisfy, we provide a state-of-the art summary of the concept of transactions in database technology to identify previously proposed concepts that may help to increase dependability.

REQUIREMENT 1. *A generic model that can combine different transactional semantics:* WS interleaved in a CWS tends to be hosted by different providers. It is most likely that their providers are using noncompliant transaction supports (if they provide any). Moreover, it is not possible to compel the WS providers to make the same transaction model available. To this end, a transaction model for the WS context must be sufficiently generic to accommodate different transactional semantics in the same model. Furthermore, it must add the required transactional semantics to the WS, if they do not exist. BPEL [4] is a typical example of a WSC specification that defines only one type of transactional semantic for all the WS interleaved in the same CWS.

The sagas model [27] was used to define the required transactional support for static CWS in BPEL. In BPEL, the only way to handle a failure is by compensation; the case where it is impossible or unnecessary to define a compensator for a particular saga is not addressed. It is true that failure atomicity is guaranteed because if any activity fails the overall process is compensated. However, we argue that the support of other transactional behaviors in the same model improves the chances of CWS execution completing successfully. Other required transactional behaviors in this situation are: *(i)* envisaging alternative mechanisms to compensation if compensation is not an option; *(ii)* having recourse to compensation only as a last resort, when there is no means of saving some part of the process progress; *(iii)* including idempotent tasks that need no compensation.

REQUIREMENT 2. *A model that can support different interaction patterns:* The logic underpinning business processes tends to be versatile and semantically varying. Consequently, in the same transaction, we may have to orchestrate elementary WS in different ways and in line with different control flow patterns (e.g., join, split, synchronize, etc.). However, a major part of the proposed transaction models only supports a concurrent or sequential interaction within a transaction. To overcome this limitation, the Workflow community contributions are of interest. In particular, well-known Workflow Patterns are those proposed in the seminal work [28]. This collection of patterns has been used to evaluate the functionality of commercial products and standards supporting the development of process-oriented applications (e.g., the METEOR project [29] and BPEL [4]). This work serves as a reliable starting point for defining the required aggregation patterns.

REQUIREMENT 3. *A model that can guarantee the best match between WS and CWS components:* A well-known characteristic of the WS realm is its unpredictability; this characteristic is not part of the equation in either the Workflow area or transaction models. Both are designed for a computing environment where modifications are very rare. Moreover, the different components, for either a transaction (subtransactions) or tasks for Workflows, are predefined, which totally eliminates unpredictability.

Returning to the WS context, unpredictability introduces a high probability of failure when WS are statically orchestrated. To overcome this limitation, it is required to define several alternative WS for the same component so that if the execution using one fails, it can be reattempted using others. Moreover, as WS tends to provide basic functionalities, it is very probable that one transaction as a whole cannot be satisfied by one WS alone. This introduces another requirement for component/transaction semantics, composition/decomposition, to facilitate and ensure that the best match is made. This requirement satisfaction is partially addressed in this paper; for a full description, refer to [18].

REQUIREMENT 4. *A model that can guarantee correct and dependable execution:* Many specified details of the CWS relate to the defined execution correctness restriction methods. In particular, we cite serializability [30], widely accepted as the cornerstone of database correctness, as unsuitable. Our justification is that serializability is very rigid and imposes restrictions that are not required (or feasible) in the WS context. For example, the shared resource condition is not satisfied because we are no longer dealing with transactions to be serializable against only one database; the different processes described as CWS are far more complex than simple write/read operations.

Several proposals, such as quasiserializability for a multidatabase environment [31], have proposed solutions that, although they relax the strict serializability condition, still target concurrency control and database integrity control.

Nevertheless, for CWS, correctness means ensuring that the semantics of the CWS are correct against the process-predefined semantics (i.e., process logic and components orders). Therefore, serializable execution is not required in the same way as in a conventional database [32]. An important approach, which indeed was already used for ensuring correct execution of CWS, is by specifying a set of Acceptable Terminal States (ATS) [33,34].

This approach was initially proposed for transactional Workflow systems and later extended to CWS. In this approach, designers have a crucial role in determining which is the correct execution, in terms of ATS. We argue that ATS is a powerful approach that fits well for CWS with a centralized and static execution, as in [34]. However, for a dynamic and distributed execution, ATS is insufficient as there is no central entity that is responsible for verifying that the execution verifies, or violates, the predefined ATS. Moreover, ATS only verifies termination dependency and, even if the different components of a CWS terminate in states included in their ATS, there is no guarantee, or means of proving, their execution order correct. We require special mechanisms to enforce that the execution order of the components of a CWS does not deviate from the prescribed order.

4 Transaction Concept: State of the Art

We highlight features of several transaction models that are interesting for dependability enhancement. We explain for each feature/model why it can or cannot be integrated in a transaction model tailored for CWS.

4.1 Traditional Transaction Model

This model is undoubtedly the precursor of all the transaction models that have been proposed. It refers to a transaction endowed with the *ACID (Atomicity, Consistency, Isolation, and Durability)* properties [35]. With these properties, each transaction is guaranteed to enforce failure atomicity and serializability as a correctness criterion. Each transaction has a flat structure.

Although the effectiveness of the traditional transaction model in conventional database applications, where transactions are generally simple and of short duration, is irrefutable, the unsuitability of its strict ACID properties for the WS context is clear. Maintaining strict isolation and serializability causes a lack of functionality, flexibility, and performance. This precludes the possibility of intertransaction cooperation and long-running transactions.

4.2 Advanced Transaction Models

Several advanced transaction models have been proposed in response to the inflexibility of the traditional transaction model (refer to [14] for a comprehensive description of some of these). We investigated the applicability of some of these models that inherently allow transaction composition (structuring) —an essential feature for a model for CWS—and encompass concepts with recognized contributions in enhancing dependability but not yet part of the WS architecture.

The nested-transaction model [36], which uses a serializable correctness criterion, made a significant contribution to the database community by: *(i)* extending the flat transaction structure to a multilevel structure; *(ii)* introducing the concept of contingent and nonvital subtransactions; and *(iii)* allowing a higher degree of intratransaction parallelism. All of these concepts are of considerable relevance to WS architecture because, first, the concurrent execution of transaction is an essential feature. Second, contingent subtransactions are easily realizable because WS that share the same functionalities are numerous; considerable research effort is directed toward achieving this issue. Third, definition of nonvital subtransactions is essential to increase availability.

To deal with the problem of long-lived transaction faults, the concept of compensation was first introduced in the sagas model [27]. A saga consists of a set of ACID subtransactions with a predefined order of execution and a set of compensating subtransactions. If a long-lived transaction fails, it can be aborted and rolled back, and then retried. However, if a saga as a whole becomes irrecoverable and has to abort, appropriate compensations are run to compensate for the completed parts of the transaction (backward recovery), that is, semantically undoing the effects of the failed parts. The other possibility in recovery is a forward recovery, that is, the system needs to retry the same failed transaction parts. The compensation ingredient here is of particular interest because it can realize a flexible fault-handling approach—a highly desirable characteristic in the WS context, in view of its high failure tendency. However, the restriction imposed by sagas that each subtransaction must be successfully compensatable cannot always be fulfilled. Therefore, alternative mechanisms for noncompensatable tasks are required.

The nested-sagas transaction model has been proposed as an extension to the sagas model [37]. It treats communication between transaction steps as an essential feature in the WS context. Each saga specifies input and output ports, bound at run time to mailboxes (i.e., queue of messages). Communication is achieved using three different classes of predefined commands: Bind, Send, and Receive.

The Flex transaction model was designed to allow more flexibility in transaction processing [33,38]. A flexible transaction is specified by defining a set of subtransactions, a set of intratransaction execution dependencies, and a set of acceptable terminal states (ATS) defining the conditions for the success of the flexible transaction. The Flex transaction model goals are very similar to our goals because it targets a multidatabase system, which can be assimilated to a special case of the WS environment where the participating systems are predefined and cannot dynamically disappear without prior notice. In particular, the way a flexible transaction is defined makes it the best candidate for CWS, as it allows the designer to specify a set of functionally equivalent subtransactions, each of which, when completed, accomplishes the task. Moreover, the contribution of the state and the intratransaction execution dependencies associated with each transaction can overcome the stateless WS and provide flexible atomicity and isolation, especially if the subtransactions support some form of compensation. It is also suitable for controlling and tracking the execution progress in a distributed environment.

While these advanced models differ in various forms, they all share the same line of thinking: the strict ACID properties support is no longer a viable solution for a nontraditional database environment. In this sense, they exploit application-specific semantics to define nonserializable correctness criteria to specify and constrain the behavior of the transaction components and their interactions. As well as these transaction models, many others were also proposed for databases (e.g., cooperative SEE transactions [14], DOM transactions [14], etc.) or by the Workflow research community. We limited our study to these models because they are at the base of many others that were proposed later.

5 WS-SAGAS Transaction Model

We propose to adopt features of interest from the transaction models described above and to build on them to make our transaction model sufficiently rich to support any CWS underpinning logic and to provide it with the required mechanisms to guarantee a dependable specification of dynamic CWS executed in a peer-to-peer environment.

Specifically, we inherit the *arbitrary nesting of transactions*, the *forward recovery with execution retrial*, the *backward recovery ensured with compensation mechanism*, the *vitality degree*, the *state*, the *Workflow-like aggregation patterns*, and the *intratransaction execution dependencies* to ensure correct execution.

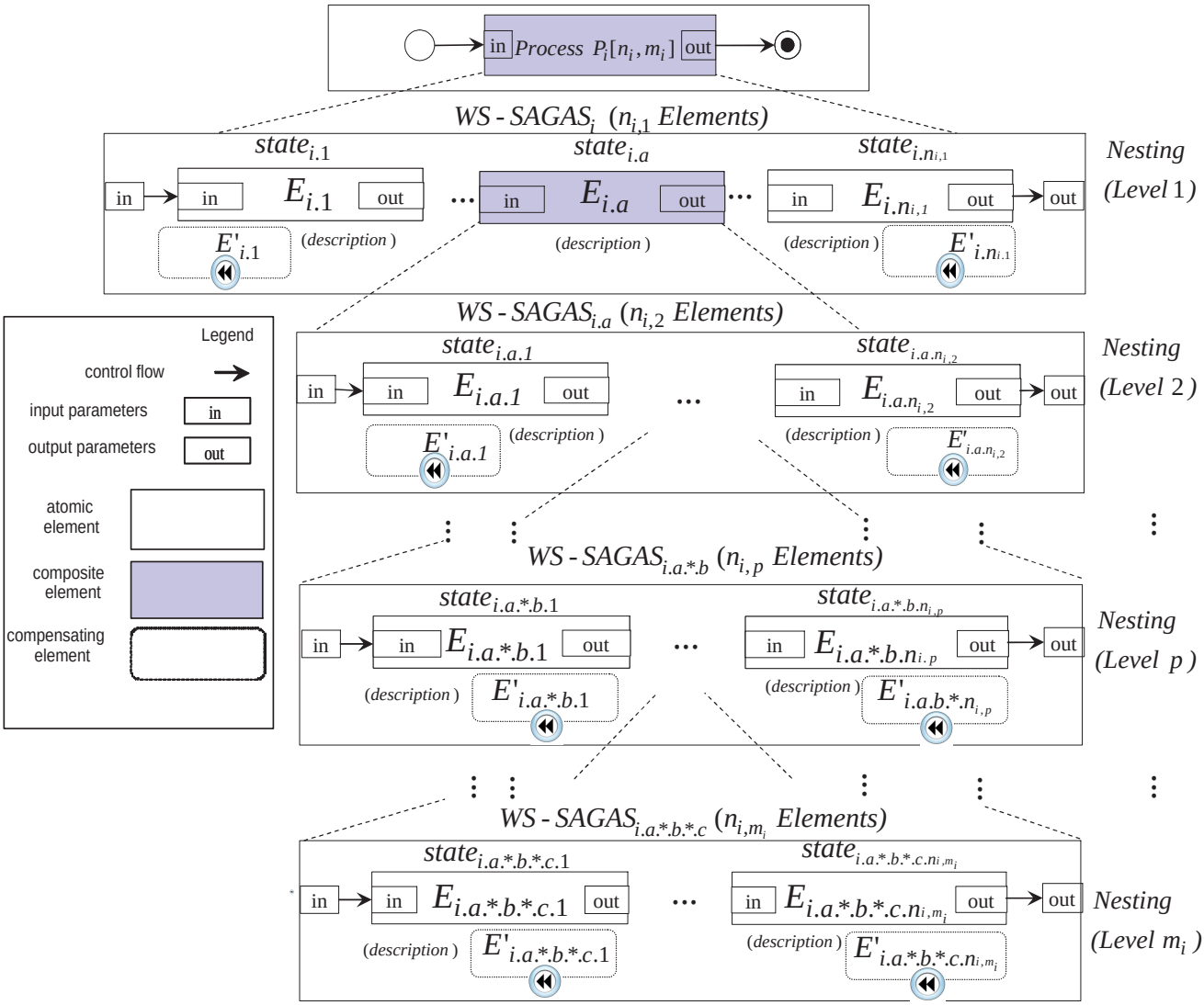


Fig. 4.1 WS-SAGAS transaction model: graphical notation $\mathcal{G}_i$ of a process P_i described as a hierarchy of recursively nested WS-SAGAS

5.1 General Assumptions

ASSUMPTION 1. We assume we are dealing with business processes that may need to combine various transactional behaviors. That is, a process puts together different activities: several are idempotent and need not be undone (e.g., displaying order information), several can be easily undone or compensated for (e.g., adding products to an order), and several others cannot be (automatically) undone because they mark a decision, commonly called noncompensatable (e.g., checking out and ordering).

ASSUMPTION 2. We assume that there is no dependency between successive invocations of the selected WS, if the dynamic WS discovery and selection leads to selecting the same WS,

ASSUMPTION 3. We assume the process of candidate WS discovery, selection, and mapping, and that verification that a certain candidate WS and a certain component from a CWS are semantically equivalent can be performed automatically. A very active area of research is measuring the semantic and syntactic similarity between WS to ensure the best match can be done. We consider this issue beyond the scope of this paper, as we can apply any of the available proposals.

ASSUMPTION 4. We assume the system designers have a comprehensive description of the business rules buried in the process-underpinning logic and they can use these rules without ambiguity to discern the different transactional behaviors and their scope (i.e., a CWS, a component from a CWS, an aggregation of components, etc.).

5.2 Description of WS-SAGAS Model Salient Features

Our model introduces the following features to specify the underpinning logic of a process (e.g., virtual travel agency), as a fault-tolerant and dynamically executed CWS against a peer-to-peer environment:

5.2.1 Process, Transaction, and Element

To allow a dynamic process composition, instead of specifying the underpinning logic of a process using a set of pre-existing WS woven together into a static CWS, we introduce the concept of an *Element*—represented by a rectangle in (Figure 4.1)—and use it as a unit in the composition of a *process* as a hierarchy of recursively nested *WS-SAGAS* transactions.

The same element can be simultaneously a component from a WS-SAGAS and a parent of other elements in another WS-SAGAS. Therefore, it is called a `composite` element and we represent it as a blue rectangle. Alternatively, an `atomic` element is only embedded in a WS-SAGAS and is represented as a white rectangle.

On executing a process, WS are dynamically discovered, and candidates are selected and mapped either to the different elements or to WS-SAGAS, considering the WS availability. In this paper we limit the WS selection and mapping to the *atomic* elements. However, our approach supports the mapping to entire WS-SAGAS. This issue is detailed in [18]. The control flow between the different elements specifies the ordering relation between the different elements and is represented by directed edges. Finally, the data flow specifies how the data produced by an element are transferred to another element and are represented by the mapping between the different input and output boxes; we do not consider this issue in this paper and we will address it in our future work. More precisely, we adopt the following notation of a process, illustrated also by (Figure 4.1):

$$\begin{aligned} P_i[n_i, m_i] : & \textit{WS-SAGAS}_i && (n_{i,1} \text{ elements}) && (\text{nesting level } 1) \\ \vdash & \textit{WS-SAGAS}_{i.a} && (n_{i,2} \text{ elements}) && (\text{nesting level } 2) \\ \vdash & \quad \vdots \\ \vdash & \textit{WS-SAGAS}_{i.a.*.b} && (n_{i,p} \text{ elements}) && (\text{nesting level } p) \\ \vdash & \quad \vdots \\ \vdash & \textit{WS-SAGAS}_{i.a.*.b.*.c} && (n_{i,m_i} \text{ elements}) && (\text{nesting level } m_i) \end{aligned} \tag{5.1}$$

A *process* (denoted P_i) is assumed to have a unique identifier i as a subscript, where i ranges over the set of natural numbers $\mathbb{N}$ to designate different processes. Each process is assumed to have n_i elements distributed over m_i nesting levels. In the hierarchy of WS-SAGAS forming the process, we denote the uppermost WS-SAGAS $\textit{WS-SAGAS}_i$. Note that we keep the same subscript for the corresponding process. Note also that $\textit{WS-SAGAS}_i$ is the only WS-SAGAS in the hierarchy that has no parent.

In ($\textit{WS-SAGAS}_i \vdash \textit{WS-SAGAS}_{i.a}$), the symbol "$\vdash$" indicates that $\textit{WS-SAGAS}_i$ is defined at the top of the subtransaction $\textit{WS-SAGAS}_{i.a}$. That is, the parent element of the subtransaction $\textit{WS-SAGAS}_{i.a}$ must be $E_{i.a}$, one of the elements aggregated in $\textit{WS-SAGAS}_i$.

A hierarchy of WS-SAGAS forming a process contains a parent *WS-SAGAS*, plus zero or more children; the children can be `atomic` elements or `composite` elements, parents of other WS-SAGAS.

We guarantee the uniqueness of an element identifier by keeping the identifier of the subtransaction it appertains to and concatenating it to a unique identifier for the element. More formally, let $E_{i.a}$ be one of the elements from the uppermost transaction $\textit{WS-SAGAS}_i$ and "$i.a$" its identifier; "i" is the index of its parent WS-SAGAS. We emphasize that the number of "$.$" in the identifier indicates the nesting level, and the last digit (i.e., a for $E_{i.a}$) indicates the order. We assume that "a" is defined in $[1..|\textit{WS-SAGAS}_i|]$ where $|\textit{WS-SAGAS}_i|$ is the cardinality (i.e., the number of assembled elements) of the subtransaction $\textit{WS-SAGAS}_i$ and is equal to $n_{i,1}$. Similarly, it is equal to $n_{i,2}$ for $\textit{WS-SAGAS}_{i.a}$ forming the second nesting level, and equal to n_{i,m_i} for the nesting level m_i containing $\textit{WS-SAGAS}_{i.a.*.b.*.c}$. To generalize, we use in $\textit{WS-SAGAS}_{i.a.*.b}$ the symbol "$*$" to indicate that there exists a subtransaction that has as a parent the element $E_{i.a.*.b}$ and that comes in one of the nesting levels after nesting level 1, which contains the element $E_{i.a}$. In "$i.a.*.b$" the symbol "$*$" is replaced to define the WS-SAGAS actual identifier. We assume $\textit{WS-SAGAS}_{i.a.*.b}$ is the nesting level p where $p < m_i$ and m_i corresponds to the last nesting level in P_i. We denote the last subtransaction in the hierarchy, which corresponds to the nesting level m_i by $\textit{WS-SAGAS}_{i.a.*.b.*.c}$; its first element is denoted $E_{i.a.*.b.*.c.1}$ and its last element is denoted $E_{i.a.*.b.*.c.n_{i,m_i}}$.

5.2.2 Vitality Degree

To add flexibility to the way failures cascade through a process, depicted as a hierarchy of WS-SAGAS transactions, we distinguish `vital` from `nonvital` elements. The vitality degree of an element is denoted by a superscript set to "v" for `vital` and to "$\bar{v}$" for `nonvital`. The vitality degree obeys these assumptions:

- A `vital` element (denoted $E^{v}_{i.k}$) must be executed successfully (i.e., it has to commit) for its parent transaction to commit.
- A `nonvital` element (denoted $E^{\bar{v}}_{i.k}$) may abort without preventing its parent transaction from committing.
- Aborting a `vital` element $E^{v}_{i.k}$ induces aborting the whole transaction it appertains to if there is no alternative WS to retry it.
- Aborting a `nonvital` element $E^{\bar{v}}_{i.k}$ does not reflect on the execution of the transaction it appertains to; the process could complete successfully although not all its component elements were `committed`. Doing so is expected to increase availability and to decrease the probability of overall process failure occurring.

We describe below the definition of the vitality degree of a process P_i depicted as a hierarchy of recursively nested WS-SAGAS transactions. In the remainder of this paper, an element's superscript is omitted and the notation ($E_{i.k}$) without specifying the vitality degree is used for an element when not relevant or interesting. The distinction between a `vital` element ($E^{v}_{i.k}$) and a `nonvital` element ($E^{\bar{v}}_{i.k}$) is only given when a special consideration is required.

5.2.3 Transactional Behavior

Every `atomic` element $E_{i.k}$ has a transactional behavior. The transactional behavior of an element is closely related to the nature of its functional semantics and is determined principally by the designers to describe how the element failure can be handled. The transactional behavior of an element can be one of the transactional behaviors described below:

- `Compensatable`: The functional semantics of the element can be undone.
- `Noncompensatable`: The functional semantics of the element cannot be undone (automatically) once done.

Two other transactional behaviors are implicitly supported by our model: `retriable` and `idempotent` elements. We assume all the `vital` elements are `retriable` with different semantically equivalent WS and that the `nonvital` elements are not retriable because their fulfillment is optional. An `idempotent` element is one that has no effect (e.g., read operation); we treat this as a `compensatable` element that is undone by running an empty compensator.

The choice of potential candidate WS for a particular element must consider the required transactional behavior for that element. When an `atomic` element is `compensatable`, we represent its compensating element just below it with a round-cornered rectangle (see Figure 4.1). Assume that the element $E_{i.1}$ is `compensatable`: we denote its compensating element $E'_{i.1}$. Similarly, a `composite` element is compensated by the different elements aggregated in its corresponding WS-SAGAS: in (Equation 5.1), $E_{i.a}$ is a `composite` element represented by the subtransaction $WS\text{-}SAGAS_{i.a}$ and it can be compensated by compensating $WS\text{-}SAGAS_{i.a}$. We describe this in detail below.

5.2.4 State

We attach to each `atomic` element from a WS-SAGAS transaction a *state* for the following reasons.

(i) To decide how to advance a process execution, (i.e., to decide whether to delegate the execution control to other element(s) or to resume it), it is essential to know the execution progress of each element separately.

(ii) At a certain point of the execution of a process, the prespecified objectives may be achieved. In this case, the process is considered to be successfully completed and can be committed. Because we consider a distributed model, where there is no central monitor that has all the required information about the execution progress, we cannot make a decision unless we attach a state to each element. We can then derive the current state of the whole process. More importantly, we can deduce whether the execution progress is correct against the process prespecified semantics and ordering.

At any time, the *state* of every element $E_{i.k}$—denoted $state_{i.k}$ (in Figure 4.1)—keeps the same identifier as the element it is attached to. The state of an element is assumed to be exclusively in one of the six *states* defined below, if the element is `compensatable` (see Figure 5.1 (a) and (b)):

1. `Waiting`: $E_{i.k}$ is not yet submitted for execution and is still waiting for the execution progress to reach its level.
2. `Executing`: $E_{i.k}$ is effectively being executed.
3. `Failed`: $E_{i.k}$ has encountered a failure.
4. `Aborted`: $E_{i.k}$ has received a request to abort itself and has obeyed it.
5. `Committed`: $E_{i.k}$ has successfully terminated and was committed.
6. `Compensated`: $E_{i.k}$ has been compensated for.

If an element `isnoncompensatable`, the set of states that model the element's internal behavior is reduced to five states by eliminating the `compensated` state (see Figure 5.1 (c) and (d)). A `vital` element is assumed to be `retriable`, and there is therefore a directed edge between the `failed` and `executing` states in Figure 5.1, (a) and (c).

For an element to transfer from one state to another, a transition condition has to be evaluated. When it is verified, several actions may be triggered. Of the different actions, one action makes the state of the element change from one state to another.

For a `compensatable` element $E_{i.k}$, to transit from the state `waiting` to the state `executing`, we assume the verifiability of the condition that indicates that at least one WS bearing the same semantic functionalities as the element must be selected. Only when the selected WS is mapped to the element does the element's state becomes `executing`.

Depending on the allocated WS execution progress and the progress of other elements in the same WS-SAGAS, the `executing` state can transit to `aborted`, if the WS execution must be canceled, or it can transit to either the `committed` or the `failed` states; this depends on whether the selected WS achieved the element's objectives or not.

Assume that the state of a `compensatable` element was set to `failed`. Subsequently, depending on the element's vitality degree, the processing differs:

a) When the element is `vital`, another candidate WS that bears the same semantic functionalities is selected and the execution is retried with this new candidate by changing the element state back to `executing`; success of execution of the new WS means success of the element and its state is set to `committed`. However, if an element is retried a number of times with different WS and all the attempts are unsuccessful and it is no longer possible to retry the execution, for any predefined reason, then the element's state remains `failed`, and a backward recovery is triggered.

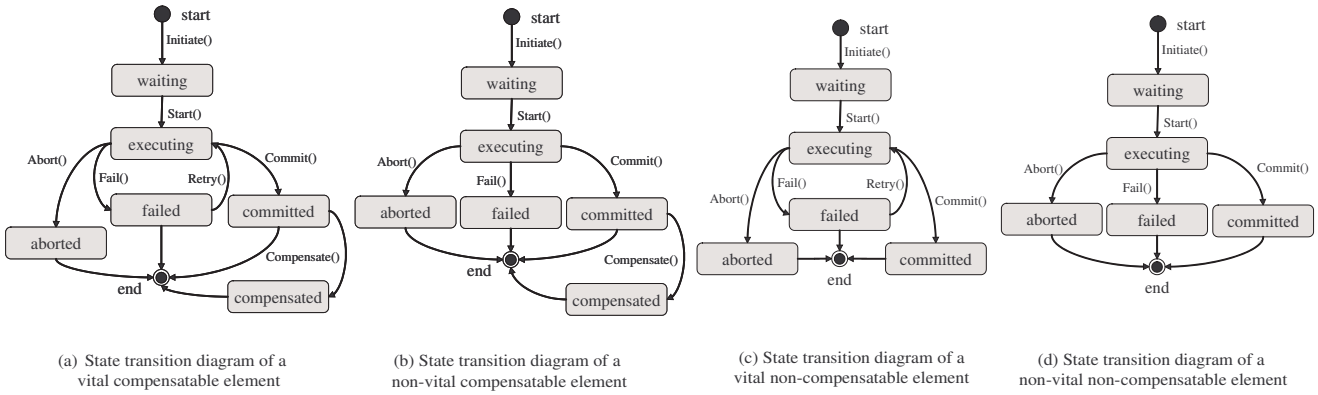


Fig. 5.1 State transition diagram for elements

b) The other possible case is when we have a `nonvital` element. If the first candidate WS execution failed, the element's state is set to `Failed`, no execution retrial is attempted, and the execution of the whole WS-SAGAS is resumed, as if the element was successful.

The main difference in processing a `compensatable` and a `noncompensatable` element becomes clear when an element is in the `committed` state and the execution of another `vital` element from the same WS-SAGAS cannot be retried so a backward recovery is necessary. In such a case, all the `compensatable` elements in the `committed` state are compensated and their state then becomes `compensated`.

The case of `noncompensatable` elements included in a WS-SAGAS requires special consideration because the issue of mixing `compensatable` and `noncompensatable` components in the same transaction is a difficult problem.

In [38], the authors introduced a commit protocol to ensure the `compensatable` components are committed before the commitment of the `noncompensatable` components. The global commit/abort decision is determined by the outcome of the `noncompensatable` components. If they abort, all of the `compensatable` components are compensated. In our model, we extend this protocol and we use mainly the state concept to allow the execution of a hierarchy of recursively nested WS-SAGAS more flexibly. A detailed discussion of this issue is in the following section.

5.2.5 Failure Recovery

The WS-SAGAS defines a *compensating* element for each element, when possible. There are two choices when an element fails to commit (e.g., allocated WS failure): the first is to attempt the element *execution retrial*, which is a variant of the sagas forward recovery. However, the difference is that the same element is reattempted *but* with another WS. If the first choice is not possible, then the second choice is *backward recovery*, in which the WS-SAGAS offers either to compensate or to abort the elements to bring the overall CWS back to a consistent state. We elaborate on this point in greater detail later in this section.

5.2.6 WS-SAGAS Notations

An investigation of most of the current CWS specification languages and approaches showed that there are three main categories of notations adopted to depict a CWS: *(a)* The first category uses an XML-based notation; BPEL [4] and WebTransact [39,40] are typical examples. *(b)* The second category opts for a graphical notation for more expressiveness and to overcome the complexity of an XML-based notation; they typically use a standardized modeling notation to describe CWS. Examples are state charts and UML models—a typical example is SELF-SERV [41]—or they define a proprietary notation, if the standard notations are not sufficiently rich to accommodate all the desired semantics of their approaches, e.g., eFlow [8]. *(c)* The third category prefers formal notations such as π-calculus or other process algebras because of their conciseness and power to analyze the semantics and correctness of the model [42].

In WS-SAGAS, we advocate the use of three notations because we are strongly convinced that one notation alone is inadequate to express all the semantics of an approach and may not be suitable for different users. First, instead of an XML-based notation, we propose a *textual notation* that can be used to generate automatically an XML-based notation of the CWS, when later implementing the system. Our textual notation is much less error prone, less complex, more human readable, and more easily modifiable. The most important feature of our notation is that we exploit it to specify and constrain the behavior of the different elements in a process and the interactions between them. In addition, to complement our textual notation and to define a common solid foundation for comparison with other formal approaches, we propose a formal notation. We also define a proprietary *graphical notation* and we use this to illustrate a running example, because the standardized notations, such as UML diagrams, do not encompass all the semantics we required for our model. Finally, to have a comprehensive notation of a CWS, our three notations can be combined or used separately.

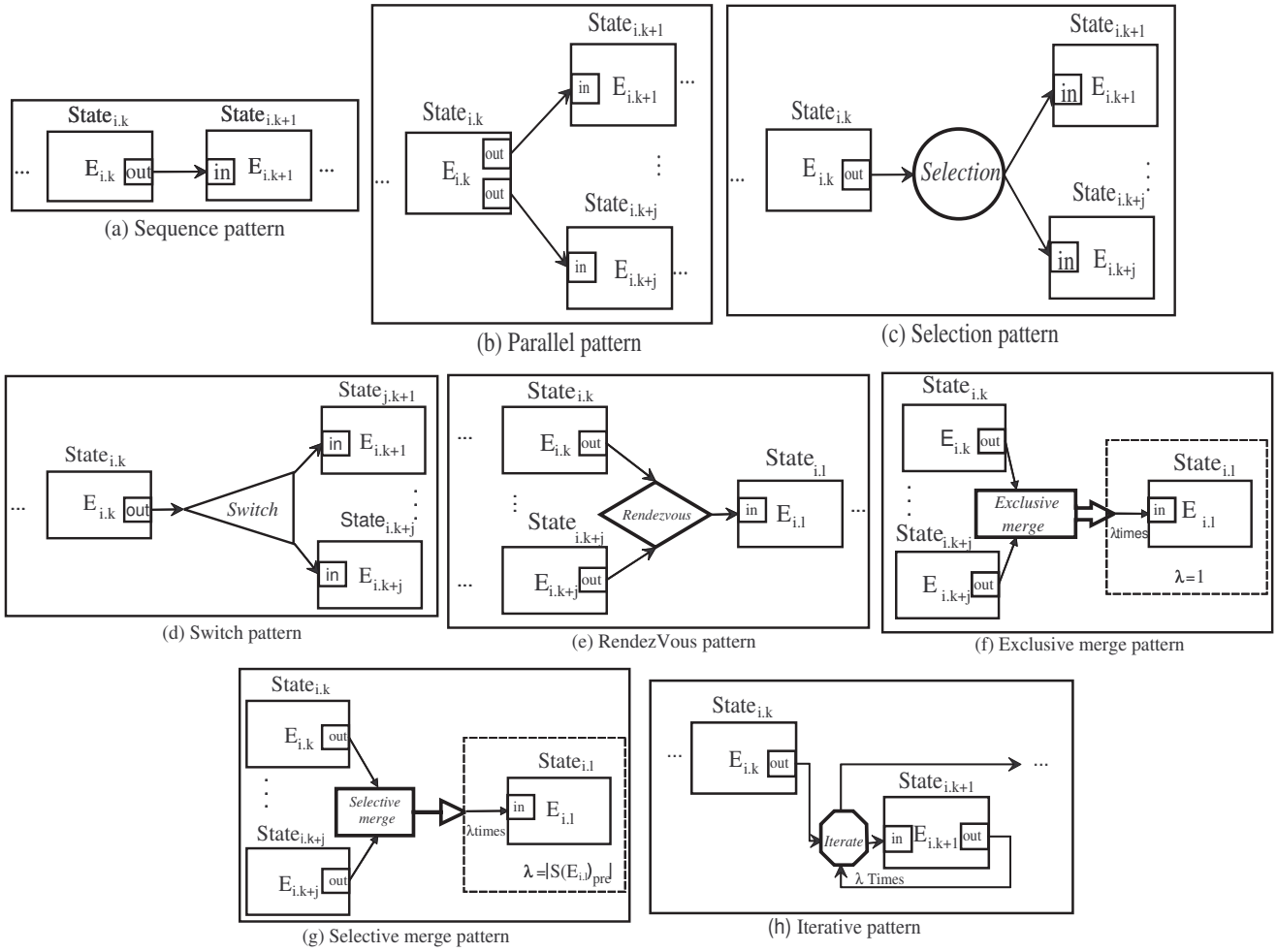


Fig. 5.2 WS-SAGAS aggregation patterns

5.2.7 WS-SAGAS Aggregation Patterns

To define the aggregation patterns, we propose building on existing work on Workflow patterns and on an analysis of existing Workflow languages reported in [28]. The following motivated our choice: *a*) Control flow dependencies encountered in Workflow modeling comply with the WS context, because the situations they capture are also relevant in this domain. *b*) Existing languages for WSC, such as BPEL and BPML, were built on the basis of languages for Workflow modeling [43]; therefore, we have a common basis for comparing our work with these approaches. *c*) It is possible for our model to incorporate different aggregation patterns into the same structure, which was not allowed with advanced transaction models proposed previously. By incorporating the transaction concept with the different aggregation patterns we enable a flexible and dependable WSC [18].

The analysis of existing Workflow languages allowed us to identify the relevant patterns necessary to model the logic of any process, no matter what it is. We identified eight patterns: *sequence, parallel, selection, switch, rendezvous, selective merge, exclusive merge, and iterative* (see Figure 5.2) [17, 18]. In [28], the authors introduced 20 patterns, but we limited our study to eight of these and we deliberately excluded the others (such as the cancellation patterns or the state-based patterns) because those eight patterns, when combined with the *compensation*, the *state*, and the *vitality degree*, are sufficient to express any process that our approach can support. Existing Workflow languages provided either a graphical notation or an XML-like notation of these patterns and, to the best of our knowledge, there is no standard notation for such aggregation patterns. We have already identified the limitations of XML-like notation and described why we prefer to define our textual notation. We continue with the same line of thinking and propose a process algebraic approach to formalize the description of our aggregation patterns. Instead of an informal notation, we propose using process algebras (PAs) because we are dealing with aggregation patterns that have semantics and the correctness of their semantics needs to be verified. PA contributions in this direction make them an interesting candidate.

5.3 WS-SAGAS Notation

The underpinning logic of a process P_i, depicted as a hierarchy of recursively nested WS-SAGAS, is denoted by a 3-tuple $\langle \mathscr{T}_i, \mathscr{G}_i, \mathscr{F}_i \rangle$ formed by a textual notation ($\mathscr{T}_i$), a graphical notation ($\mathscr{G}_i$), and a formal notation ($\mathscr{F}_i$).

5.3.1 Description of Textual Notation ($\mathscr{T}_i$)

A textual notation of a process P_i (denoted $\mathscr{T}_i$) is formed with three different sets of *Rules* identified by the system designers using the process logic description: the set of *Definition Rules (DR)*, the set of *Composability Rules (CR)*, and the set *Ordering Rules (OR)*.

The *DR*,*CR*, and *OR* share the same tuple-like notation but their semantics differ because: *(i)* Each *DR* gives relevant information of an *entity* that either relates to the CWS specification (e.g., a process, an element, a component, etc.) or intervenes in the CWS execution (e.g., a WS, a coordinator, etc.). *(ii)* Each *CR* specifies the relation between the different *entities* defined by the different *DR* (i.e., how the entities are combined, how the entities interact with each other, etc.). *(iii)* Each *OR* defines the condition that the relation between the different entities defined by the *CR* must verify (i.e., under which condition entities are combined, under which condition entities interact with each other).

We propose the following tuple-like generic notations to define a *DR*, a *CR*, and an *OR*:

$$
\begin{aligned}
&DR(Entity_u) : \langle Attribute_1*, \ldots \rangle \\
&DR(Entity_v) : \langle Attribute_2\times, \ldots \rangle \\
&DR(Entity_w) : \langle Attribute_3\bullet, (Attribute_4, Attribute_5), \ldots \rangle \\
&CR(Entity_w) \rightarrow \langle Entity_u, Entity_v \rangle \\
&OR(Entity_w) \rightarrow \langle Entity_u \; op \; Entity_v \rangle \quad ,
\end{aligned}
$$

where:

- *Attribute* is the relevant information about an *Entity*. We define the multiplicity of each attribute to indicate the *Attribute* occurrence number, that is, the number of times we may find the attribute; we define four forms of an attribute's multiplicity: *i*) The notation $Attribute_1*$ indicates that $Entity_u$ may define zero or several values of $Attribute_1$. *ii*) The notation $Attribute_2\times$ indicates that the $Entity_v$ must define at least one value of $Attribute_2$. *iii*) The notation $Attribute_3\bullet$ indicates that this attribute is optional and at most one value can be provided. *iv*) An attribute name with nothing next to it similarly to the attribute $Attribute_4$ indicates that one value only is to be provided. *v*) Finally, $Attribute_4$ and $Attribute_5$ are associated and this is indicated by the parentheses.
- $CR(Entity_w)$ indicates that $Entity_w$ defines a *composability* relation between $Entity_u$ and $Entity_v$.
- *op* is the condition that the composability relation between $Entity_u$ and $Entity_v$ must define.

Specifically, we define *DR* to provide relevant information about three different entities: a *process*, a *WS-SAGAS*, and an *element*. The set of *CR* identifies the relation of composability between the different elements and WS-SAGAS (e.g., which WS-SAGAS is composed of/composing which WS-SAGAS/elements). Finally, the set of *OR* identifies the ordering condition that every relation of composability between the elements and process must verify.

Definition 5.1 (Definition Rule of an Element)
Let $E_{i.k}$ be an `Atomic` element from $WS\text{-}SAGAS_i$. $DR(E_{i.k})$ is an ordered tuple that provides relevant information of an $E_{i.k}$, namely its *name*, *description*, *state*, *vitality degree*, *transactional behavior*, *operation* with its corresponding *input* and *output* parameters, and its *QoS* attributes. We added only those attributes we considered to be fundamental in composing WS; extending the *DR* expression with other attributes is possible. We use the following notation of a *DR* of an element, which is a specialization of the generic *DR* notation for an *entity* described above:

$$
\begin{aligned}
DR(E_{i.k}) : \langle & name, description, behavior, state, \\
& type, vitality, (operation \times (in*, out*)), qos* \rangle \quad ,
\end{aligned}
$$

where:

- *name* is the name of the element.
- *description* is a concise description of the element's main semantic functionality.
- *behavior* is the transactional behavior of the element. This attribute verifies the condition:

$$
\begin{aligned}
DR(E_{i.k}).behavior \in \{ & \texttt{compensatable}, \\
& \texttt{non-compensatable} \} \quad .
\end{aligned}
$$

- *state* describes the execution progress of the element.
- *vitality* is the vitality degree of the element. This attribute verifies the condition:

$$DR(E_{i.k}).vitality \in \{\texttt{vital}, \texttt{nonvital}\} \quad .$$

- *type* is the element granularity: an element that has no children is `atomic`. Otherwise the element is a parent of a WS-SAGAS and its type is `composite`.
- $operation\times$ indicates that an element may define several values of *operation*, but at least one.
- $in*$ and $out*$ are the different input and output parameters of the element.
- The different input and output parameters are associated with their corresponding operation by using parentheses; we may define at least one operation, but input and output parameters are optional. For example, we may have $(operation_1(in_1, in_2, out_1)), (operation_2())$, for an element with two operations, one of which is a void function that takes no argument.
- $qos*$ are the different QoS attributes of the element, which are estimated when the element is executed. We denote this by *qos* tuples of attributes; for example, we may have $qos* =< qos_1, qos_2, qos_3 >$ to describe an element where we are interested in three particular QoS attributes: the execution time, the reliability, and the cost.

For each `atomic compensatable` element $E_{i.k}$ verifying ($DR(E_{i.k}).behavior =$ `Compensatable` and $DR(E_{i.k}).type =$ `Atomic`), we define a *compensating element* (denoted $E'_{i.k}$). This element is invoked if a failure later in the execution of $E_{i.k}$ makes it necessary.
The occurrence of element $E'_{i.k}$ after element $E_{i.k}$ restores the system to a state that is an acceptable approximation of its state before the start of the execution. For every compensating element, we may define the *definition rule of a compensating element (exactly $DR(E'_{i.k})$)*, in the same way we defined it for an element.

Definition 5.2 (Definition Rule of a WS-SAGAS)
Let $WS\text{-}SAGAS_{i.a}$ be a subtransaction formed by $n_{i,2}$ elements and having as a parent the `composite` element $E_{i.a}$ from $WS\text{-}SAGAS_i$ (see Equation 5.1).
$DR(WS\text{-}SAGAS_{i.a})$ is an ordered tuple that provides relevant information on $WS\text{-}SAGAS_i$, specifically its *name*, *description*, *state*, *vitality degree*, *transactional behavior*, and *QoS* attributes. The values of several attributes are deduced from the attributes in the composing elements of the WS-SAGAS.

$$DR(WS\text{-}SAGAS_{i.a}) : \langle name, description, behavior\times, state\times, vitality, qos\times \rangle \ ,$$

where:

- *name* is the name identifier of the WS-SAGAS.
- $description\times$ is a concise description of the WS-SAGAS; it combines the description of the different elements that appertain to this WS-SAGAS and verifies:

$$DR(WS\text{-}SAGAS_{i.a}).\ description \equiv \bigcup_{\ell=1}^{n_{i,2}} (DR(E_{i.a.\ell}).description) \ .$$

- $state\times$ describes the execution progress of the subtransaction $WS\text{-}SAGAS_{i.a}$ It is an $n_{i,2}$-tuple formed by the states of the elements composing $WS\text{-}SAGAS_{i.a}$:

$$DR(WS\text{-}SAGAS_{i.a}).state\times \equiv \bigcup_{\ell=1}^{n_{i,2}} (DR(E_{i.a.\ell}).state) \ .$$

  We assume that only the state of the `vital` elements affects the overall WS-SAGAS commitment's decision. In addition, all the `compensatable` elements must wait for the `noncompensatable` elements from the same subtransaction (i.e., nesting level) to be able to commit their work. By putting together these two assumptions, we can reduce the state of a WS-SAGAS to the set of states of the elements that verify the conditions:

$$DR(WS\text{-}SAGAS_{i.a}).state\times \equiv \bigcup_{\ell=1}^{n_{i,2}} (DR(E_{i.a.\ell}).state) \text{ with: } DR(E_{i.a.\ell}).vitality = \texttt{Vital} \text{ and } DR(E_{i.a.\ell}).behavior = \texttt{Noncompensatable} \ .$$

- *vitality* is a reduction of a tuple formed by $n_{i,2}$ vitality degrees, one for every `atomic` element aggregated in $WS\text{-}SAGAS_{i.a}$. We reduce this $n_{i,2}$-tuple to a one-value tuple. If there is at least one `vital` element in a WS-SAGAS, the overall WS-SAGAS is `vital`. However, a WS-SAGAS is `nonvital` if all of its composing elements are `nonvital`. These requirements are formulated below:
  - $DR(WS\text{-}SAGAS_{i.a}).vitality =$ `nonvital` ***iff*** $\{\forall E_{i.a.\ell} \in WS\text{-}SAGAS_{i.a} | \ell \in [1..n_{i,2}]\}$ we have: $(DR(E_{i.a.\ell}).vitality =$ `nonvital`$)$.
  - $DR(WS\text{-}SAGAS_{i.a}).vitality =$ `vital` ***iff*** $\{\exists E_{i.a.\ell} \in WS\text{-}SAGAS_{i.a} | \ell \in [1..n_{i,2}]\}$ that verifies: $(|DR(E_{i.a.\ell}).vitality =$ `vital`$)$.
- *behavior*$\times$ is a $n_{i,2}$ tuple formed by the transactional behaviors of the `atomic` elements in $WS\text{-}SAGAS_{i.a}$. Its expression is:

$$DR(WS\text{-}SAGAS_{i.a}).behavior\times \equiv \bigcup_{\ell=1}^{n_{i,2}} (DR(E_{i.a.\ell}).behavior) \ .$$

- Finally, $qos\times$ are the different QoS attributes we consider; we derive these on the basis of the *qos* attributes of the elements that appertain to the WS-SAGAS. We describe how below.

Definition 5.3 (Set of Definition Rule of a Process)
The set of *DR* that defines a process P_i is derived below based on the expression of a process in (Equation 5.1):

$$\begin{aligned} DR(P_i)[n_i, m_i] &\equiv \bigcup_{\ell=i}^{i.a.*.b.*.c} DR(WS\text{-}SAGAS_\ell) \bigcup_{\partial=i.1}^{i.a.*.b.*.c.n_{i,m_i}} DR(E_\partial) \\ &\equiv \Bigg(DR(WS\text{-}SAGAS_i) \bigcup_{\partial=i.1}^{i.n_{i,1}} DR(E_\partial)) \\ &\quad \cup (DR(WS\text{-}SAGAS_{i.a}) \bigcup_{\partial=i.a.1}^{i.a.n_{i,2}} DR(E_\partial)) \cdots \\ &\quad \cup (DR(WS\text{-}SAGAS_{i.a.*.b}) \bigcup_{\partial=i.a.*.b.1}^{i.a.*.b.n_{i,p}} DR(E_\partial)) \cdots \\ &\quad \cup (DR(WS\text{-}SAGAS_{i.a.*.b.*.c}) \bigcup_{\partial=i.a.*.b.*.c.1}^{i.a.*.b.*.c.n_{i,m_i}} DR(E_\partial)) \Bigg) \ . \end{aligned}$$

Definition 5.4 (Composability Rule of a WS-SAGAS)
The next step in our modeling approach is defining the *Composability Rules (CR)*, essential in defining the nesting and composition dependency between the different WS-SAGAS. A typical *CR* of a WS-SAGAS is the specialization of the entity *CR* described above. Below we describe the *CR* of $WS\text{-}SAGAS_{i.a}$, the second nesting level composed of $n_{i,2}$ elements:

$$CR(WS\text{-}SAGAS_{i.a}) \rightarrow \langle E_{i.a.1}, \cdots, E_{i.a.n_{i,2}} \rangle \ .$$

Definition 5.5 (Set of Composability Rules for a Process) We define the set of *CR* for the process shown in (Figure 4.1) and (Equation 5.1):

$$\begin{aligned} CR(P_i)[n_i,m_i] &\equiv \bigcup_{\ell=i}^{i.a.*.b.*.c} (CR(WS\text{-}SAGAS_\ell)) \\ &\equiv CR(WS\text{-}SAGAS_i) \\ &\bigcup CR(WS\text{-}SAGAS_{i.a})\cdots \\ &\bigcup CR(WS\text{-}SAGAS_{i.a.*.b})\cdots \\ &\bigcup CR(WS\text{-}SAGAS_{i.a.*.b.*.c}) \ , \end{aligned}$$

where:

$$\begin{aligned} &CR(WS\text{-}SAGAS_i) && \rightarrow \langle E_{i.1},\cdots,E_{i.a},\cdots,E_{i.n_{i,1}}\rangle \\ &CR(WS\text{-}SAGAS_{i.a}) && \rightarrow \langle E_{i.a.1},\cdots,E_{i.a.n_{i,2}}\rangle \\ &\vdots && \vdots \\ &CR(WS\text{-}SAGAS_{i.a.*.b}) && \rightarrow \langle E_{i.a.*.b.1},\cdots,E_{i.a.*.b.n_{i,p}}\rangle \\ &\vdots && \vdots \\ &CR(WS\text{-}SAGAS_{i.a.*.b.*.c}) && \rightarrow \langle E_{i.a.*.b.*.c.1},\cdots,E_{i.a.*.b.*.c.n_{i,m_i}}\rangle \ . \end{aligned}$$

Definition 5.6 (Ordering Rule of a WS-SAGAS) The step that comes after identifying the different *CR* is the *Ordering Rules* (*OR*) definition. The most important feature of this step is that each rule builds on the process's predefined semantics to define and restrict the execution dependencies between the different elements/WS-SAGAS forming a process (i.e., the correct execution orders). For a WS-SAGAS, if no *OR* is explicitly defined, then the order of the different elements order is interchangeable. Below we describe the *OR* of $WS\text{-}SAGAS_{i.a}$ representing the second nesting level and composed of $n_{i,2}$ elements:

$$OR(WS\text{-}SAGAS_{i.a}) \rightarrow \langle E_{i.a.1}\ op\cdots op\ E_{i.a.n_{i,2}}\rangle \ .$$

Definition 5.7 (Set of Ordering Rules of a Process) We define the different *OR* of the WS-SAGAS subtransactions nested in the process depicted in (Figure 4.1) and (Equation 5.1):

$$\begin{aligned} OR(P_i)[n_i,m_i] &\equiv \bigcup_{\ell=i}^{i.a.*.b.*.c} (OR(WS\text{-}SAGAS_\ell)) \\ &\equiv OR(WS\text{-}SAGAS_i) \\ &\bigcup OR(WS\text{-}SAGAS_{i.a})\cdots \\ &\bigcup OR(WS\text{-}SAGAS_{i.a.*.b})\cdots \\ &\bigcup OR(WS\text{-}SAGAS_{i.a.*.b.*.c}) \ , \end{aligned}$$

where:

$$\begin{aligned} &OR(WS\text{-}SAGAS_i) && \rightarrow\langle E_{i.1}\ op\ \cdots op\ E_{i.n_{i,1}}\rangle \\ &OR(WS\text{-}SAGAS_{i.a}) && \rightarrow\langle E_{i.a.1}\ op\cdots op\ E_{i.a.n_{i,2}}\rangle \\ &\vdots && \vdots \\ &OR(WS\text{-}SAGAS_{i.a.*.b}) && \rightarrow\langle E_{i.a.*.b.1}\ op\cdots op\ E_{i.a.*.b.n_{i,p}}\rangle \\ &\vdots && \vdots \\ &OR(WS\text{-}SAGAS_{i.a.*.b.*.c}) && \rightarrow\langle E_{i.a.*.b.*.c.1}\ op\cdots op E_{i.a.*.b.*.c.n_{i,m_i}}\rangle \end{aligned}$$

In the different *OR*, *op* stands for "operator" and it depends on the control flow that describes the process in terms of elements and their execution ordering through different constructors (e.g., sequence, choice, parallelism, and synchronization).

Considering how business process logic tends often to involve complex behaviors and capabilities, which are structured in different ways, we need to enrich WS-SAGAS with a set of constructors that broadens its potential scope and make it sufficiently rich to sustain any business process, no matter how complex; this remains an ongoing problem in the area of transaction models. The different "operators" are the eight different aggregation patterns we defined on the basis of the seminal work in [28]. To fill the gap caused by the absence of a standard textual notation of the different patterns, we build on the formal notations and PAs.

5.3.2 Description of Formal Notation ($\mathscr{F}_i$)

PAs [44] are formal description techniques to specify software systems, particularly those formed from concurrent and communicating components. Numerous PAs have been proposed; well-known PAs are Milner's Calculus for Communicating Systems (CCS) [45], Hoare's Communicating Sequential Processes (CSP) [46], and all their extensions, such as the π-calculus and LOTOS [44]. These PAs define typically simple constructions to describe dynamic behavior, compositional modeling, operational semantics, behavioral reasoning by model checking, and process equivalence.

PAs comply with the WSC issue because they allow description of formally dynamic processes. In addition, their predefined constructs are adequate to specify CWS, due to their inherent composability property [42].

There are a large number of existing PAs; the most adequate formalism can be determined based on the desired expressiveness orientation. The encoding proposed in any of the PAs can be smoothly translated into any other standard PA.

We chose to build on the Compensating CSP [47], a variant of the CSP PA, because it already supports compensation and reasoning for long-running transactions. The atomic events of CSP are used to model the elements of a WS-SAGAS; several atomic elements can be combined using the operators provided by the CSP language to support sequencing, choice, and parallel composition. In addition, to support failed transactions, compensation operators are inherited from the Compensating CSP. Finally, to allow more advanced combinations to support other aggregation patterns that WS-SAGAS requires to formalize the eight aggregation patterns it defined but that CSP does not define, we introduce a set of advanced aggregation operators.

In formalizing WS-SAGAS, we describe a syntax in the spirit of CSP defined by the following grammar in BNF-like notation:

$$
\begin{array}{lll}
P_i ::= WS\text{-}SAGAS_i \vdash WS\text{-}SAGAS_{i.*} & & \textit{(nesting)} \\
WS\text{-}SAGAS_i ::= [E_{i.k}]; [E_{i.k+1}] & & \textit{(sequence)} \\
\quad | \; [E_{i.k}] \;||\; [E_{i.k+1}] & & \textit{(parallel)} \\
\quad | \; [E_{i.k}] \star [E_{i.k+1}] & & \textit{(arbitrary ordering)} \\
\quad | \; [E_{i.k}] \bigcirc ([E_{i.k+1}] \;||\; [E_{i.k+2}]) & & \textit{(selection)} \\
\quad | \; [E_{i.k}] \lhd ([E_{i.k+1}] \;||\; [E_{i.k+1}]) & & \textit{(fork/choice)} \\
\quad | \; ([E_{i.k}] \;||\; [E_{i.k+1}]) \diamond [E_{i.k+2}] & & \textit{(join)} \\
\quad | \; ([E_{i.k}] \star [E_{i.k+1}]) \;\square\!\!\Rightarrow\; [E_{i.k+2}] & & \textit{(selective merge)} \\
\quad | \; ([E_{i.k}] \star [E_{i.k+1}]) \;\square\!\!\rightarrow\; [E_{i.k+2}] & & \textit{(exclusive merge)} \\
\quad | \; \lambda [E_{i.k}] & & \textit{(iteration)} \\
[E_{i.k}] ::= E^{v}_{i.k} \div E'^{v}_{i.k} \mid E^{\bar{v}}_{i.k} \div E'^{\bar{v}}_{i.k} & & \textit{(compensating pair)} \;.
\end{array}
$$

where:

- P_i designates a process and we represent it as a hierarchy of recursively nested WS-SAGAS by adopting the notation ($WS\text{-}SAGAS_i \vdash WS\text{-}SAGAS_{i.*}$);
- $WS\text{-}SAGAS_{i.*}$ is the lowermost nested subtransaction and "$i.*$" is to be replaced by the subtransaction identifier;
- $[E_{i.k}]$, $[E_{i.k+1}]$, and $[E_{i.k+2}]$ are elements from $WS\text{-}SAGAS_i$ where an element enclosed between "[" and "]" can be a compensating pair of a `vital` or a `nonvital` element, if the element is defined as `compensatable`;
- $[E_{i.k}] and [E_{i.k+1}]$ represent the sequential construction that combines two elements: $[E_{i.k}]$ is executed first, and only when $[E_{i.k}]$ terminates successfully can $[E_{i.k}]$ be executed;
- $[E_{i.k}] \;||\; [E_{i.k+1}]$ is a parallel composition of two elements;
- $[E_{i.k}] \star [E_{i.k+1}]$ represents the operator for constructing the execution of elements where the execution order is arbitrary; it can be in parallel, sequentially, or a combination of these two;
- $[E_{i.k}] \bigcirc ([E_{i.k+1}] \;||\; [E_{i.k+2}])$ represents the selective choice of $[E_{i.k}]$, which selects whichever of $[E_{i.k+1}]$ and/or $[E_{i.k+2}]$ is to be enabled;
- $[E_{i.k}] \lhd ([E_{i.k+1}] \;||\; [E_{i.k+2}])$ represents a particular case of the selective choice operator because only one of $[E_{i.k+1}]$ and $[E_{i.k+1}]$ is to be enabled;
- $([E_{i.k}] \;||\; [E_{i.k+1}]) \diamond [E_{i.k+2}]$ represents where the elements $[E_{i.k}]$ and $[E_{i.k+1}]$ are synchronized at a particular rendezvous point and must wait for each other to execute the element that comes directly after them;
- $([E_{i.k}] \star [E_{i.k+1}]) \;\square\!\!\rightarrow\; [E_{i.k+2}]$ represents where $[E_{i.k}]$ and $[E_{i.k+1}]$ converge but without synchronization at a particular rendezvous point; the element that comes directly after them (i.e., $[E_{i.k+2}]$) is activated every time either of these two elements reaches the rendezvous point;
- $([E_{i.k}] \star [E_{i.k+1}]) \;\square\!\!\Rightarrow\; [E_{i.k+2}]$ is a special case of $(([E_{i.k}] \star [E_{i.k+1}]) \;\square\!\!\rightarrow\; [E_{i.k+2}])$; the difference is that the first element that terminates its execution activates the execution of $[E_{i.k+2}]$;
- $\lambda [E_{i.k}]$ is λ iteration of $[E_{i.k}]$.

5.3.3 Description of Graphical Notation ($\mathscr{G}_i$)

Our proposed graphical notation of WS-SAGAS is shown in (Figure 5.2).

5.4 WS-SAGAS Transaction Model: Execution Semantics and Correctness

To eliminate ambiguities, to allow analysis and further reasoning regarding our transaction model, and to facilitate its comparison with other models, it is necessary to define our model operational semantics and correct execution. Because we are considering a peer-to-peer execution model, the use of strict serializability poses severe limitations that are unacceptable. The description of a process in terms of *DR*, *CR*, and in particular *OR*, partly contributes to avoiding inconsistencies because the different *OR* allow definition of the correct control flow in a process.

To ensure the semantics of each element are respected, when each element executed, in particular its nesting, transactional behavior, and vitality degree, we build on the state concept and define several types of dependencies that must hold between the different elements combined in the same pattern; we term these *intrapattern dependencies*. These dependencies formulate the required conditions for a pattern to commit and describe how failure recovery is performed. Because every WS-SAGAS combines elements following different patterned operational semantics, to define a correct WS-SAGAS on the basis of the different intrapattern dependencies, we describe another form of dependencies, called *intra-WS-SAGAS dependencies*, that formulate the required conditions for a WS-SAGAS to commit and describe how failure recovery is performed.

Finally, we formulate the conditions for correct execution of a process in terms of *intraprocess dependencies* by taking as a basis the intra-WS-SAGAS dependencies formulated for each WS-SAGAS appertaining to the hierarchy of WS-SAGAS in the process.

5.4.1 WS-SAGAS Pattern Execution Semantics

Let $WS\text{-}SAGAS_{i.a}$ be a subtransaction from a hierarchy forming a process P_i (Equation 5.1). $WS\text{-}SAGAS_{i.a}$ combines a collection of elements defined in $CR(WS\text{-}SAGAS_{i.a})$. This collection of elements is equal to $\bigcup_{\ell=i.a.1}^{i.a.n_{i,2}} E_\ell$.

We define *WS-pattern* as the set of possible patterns defined by combining the CSP-like notation and the Workflow patterns:

$$
\begin{aligned}
WS\text{-}pattern : \{ & sequence(;), parallel(||), arbitrary(\star), \\
& selection(\bigcirc), switch(\lhd), iterative(\lambda), rendezvous(\diamond), \\
& selectivemerge(\square\!\!\rightarrow), exclusivemerge(\square\!\!\Rightarrow)\} \;.
\end{aligned}
$$

The different patterns, with their defined operators, are used to write the set of *OR*, as described above in this section.

Depending on the pattern's semantics, the operator of a pattern can be prefixed ($\bigcirc$, $\lhd$, λ), postfixed ($\Diamond$, $\Box\!\!\Rightarrow$, and $\Box\!\!\rightarrow$), or infixed (;, $||$, and $\star$). We define for each pattern a scope that delimits the elements within the reach of that pattern and that should verify its semantics.

We assume in what follows that the scope of each pattern includes only *atomic* elements between $E_{i.a.k}$ and $E_{i.a.l}$, where the subscripts of these two elements verify $k < l < n_{i,2}$. The end of one scope and the start of another is decided when a postfixed or a prefixed operator is encountered in an *OR*. Overlapping of elements between consecutive scopes is allowed. The case of `composite` elements is considered below in the description of the nesting semantics.

We assume there is an entity that contains the different *DR*, *CR*, and *OR* of the entire process. On every execution of every element of a process, the element's state in this entity is updated.

The entity that contains all this information is transferred between elements (i.e., an engine or an authority responsible for the execution of the element) as the execution process advances. We also assume that each element keeps a copy of this entity until the end of the process instance execution. Therefore, any element can know the set of elements that come after and before it.

To describe the patterns' semantics, we define for each pattern several types of dependencies that formulate the conditions that the elements in the pattern must satisfy to activate, commit, interrupt, compensate, or abort the pattern execution. The concept of dependencies is strongly related to the concept of state. We define five types of dependency. Each dependency is denoted by $\mathrm{intra}^{\mathrm{superscript}}(pattern)$, where the superscript is replaced with an abbreviation of the type of dependency and the pattern is defined in *WS-pattern*:

- $\mathrm{intra}^{\mathrm{ac}}(pattern)$ is an *intrapattern execution activation dependency* and describes the condition(s) that must be verified for the elements combined in the pattern to start execution.
- $\mathrm{intra}^{\mathrm{c}}(pattern)$ is an *intrapattern execution commitment dependency* that describes the condition(s) required for the pattern to be successfully terminated.
- $\mathrm{intra}^{\mathrm{i}}(pattern)$ is an *intrapattern execution interruption dependency* that describes the condition(s) where, if verified, the execution of the whole pattern is in a situation where forward recovery is insufficient to suppress a failure and a backward recovery is required.
  When $\mathrm{intra}^{\mathrm{i}}(pattern)$ is valid, depending on the pattern execution progress and from its composing elements, an *intrapattern execution compensation dependency* and/or *intrapattern execution aborting dependency* is/are triggered.
- $\mathrm{intra}^{\mathrm{cp}}(pattern)$ is an *intrapattern execution compensation dependency* that formulates the condition(s) that, if verified, ensure the consistency of the execution by triggering a compensation mechanism.
- $\mathrm{intra}^{\mathrm{a}}(pattern)$ is an *intrapattern execution aborting dependency* that formulates the condition(s) where, if verified, the consistency of the execution is ensured by aborting the elements that have to be aborted included in the pattern.

SEQUENCE PATTERN $([E_{i.a.k}];\ldots;[E_{i.a.k+j}])$

By $\bigcup_{\ell=i.a.k}^{i.a.k+j} E_\ell$ (Figure 5.2 (a)) we denote a set of elements aggregated in a sequence pattern. To ensure the correct execution of a sequence, we assume that among $\bigcup_{\ell=i.a.k}^{i.k+j} E_\ell$, there must exist only one `vital noncompensatable` element. Where more than one `vital noncompensatable` element is included in the sequence, splitting the sequence into several sequences is envisaged. Assuming that the `vital noncompensatable` element in question is $E^{v}_{i.a.\partial}$, it must verify the following conditions:

$$\begin{cases} DR(E_{i.a.\partial}).behavior = \texttt{non-compensatable}\ \textit{and} \\ DR(E_{i.a.\partial}).vitality = \texttt{vital}\ . \end{cases}$$

The activation of the execution of each element requires the termination of each direct predecessor. More formally, let E_ℓ be an element verifying $\{\ell \in [i.a.k\,..\,i.a.k+j]\}$; the execution of E_ℓ requires the successful termination of its direct predecessor, if it is vital (i.e., $DR(E^{v}_{\ell-1}).state = \texttt{Committed}$), and the termination of its predecessor, even with a failure (i.e., $DR(E^{\bar{v}}_{\ell-1}).state = \texttt{Failed}$), if it is `nonvital`.

The commitment of the sequence of elements depends on $E^{v}_{i.a.\partial}$; more formally, the intracommitment dependency of this pattern is specified as:

- $\mathrm{intra}^{\mathrm{c}}(sequence)$ verification requires that (CONDITION S1) and (CONDITION S2) are valid:
  (CONDITION S1.) The sequence can attempt to commit ***iff*** $DR(E_{i.a.\partial}).state = \texttt{Committed}$.
  (CONDITION S2.) If the previous condition is valid, then the sequence can be committed ***iff***

$$\begin{cases} \forall E_\ell | \ell \in [i.a.k\,..\,i.a.k+j] \text{ verifying: } DR(E_\ell).vitality = \\ \texttt{vital},\ \text{we have: } \texttt{DR}(\texttt{E}_\ell).\texttt{state} = \texttt{Committed}. \end{cases}$$

If the set of elements combined in the sequence does not encompass any `vital noncompensatable` element, then the intracommitment dependency verification requires satisfaction only of (CONDITION S2).

If (CONDITION S2) is not verified, that is:

$$\begin{cases} \exists E_\ell | \ell \in [i.a.k\,..\,i.a.k+j] \text{ that verifies: } DR(E_\ell).state = \\ \texttt{Failed} \text{ and } DR(E_\ell).vitality = \texttt{vital}. \end{cases}$$

then the two conditions we define below, (CONDITION S3) and (CONDITION S4), are evaluated. Subsequently, a backward recovery is triggered in the same way whether a sequence includes a `vital noncompensatable` element or not.

An extreme situation is when the set of elements combined in the sequence are `compensatable` and `nonvital`; in such a case, even if all the elements fail, the pattern intra-interruption dependency is deduced and it has no effect on the overall WS-SAGAS, because a `nonvital` WS-SAGAS success is not critical for the overall process commitment.

Assume that the element $E^v_{i.a.\partial}$ was attempted a number of times with different WS but none of those attempts was successful; this mechanism is actually a *forward recovery* where an element is reattempted with different WS. In this case, the element is assumed to have failed and a *backward recovery* is triggered, which implies the verification of the *intrainterruption dependency* of this sequence pattern; more formally:

- $\mathrm{intra}^{\mathrm{i}}(sequence)$ verification requires that (CONDITION S3) is valid:
  (CONDITION S3.) The execution of the sequence pattern is interrupted ***iff*** $DR(E_{i.a.\partial}).state = \texttt{Failed}$ is verified.

Depending on the execution progress of all the other elements in the sequence (i.e., $\bigcup_{\ell=i.a.k}^{i.k+j} E_\ell - E_{i.a.\partial}$), the verification of $\mathrm{intra}^{\mathrm{i}}(sequence)$ may trigger an *intrapattern compensation dependency*, an *intrapattern aborting dependency*, or both. More formally:

- $\mathrm{intra}^{\mathrm{a}}(sequence)$ denotes an intrapattern aborting dependency in a pattern; it requires that $\mathrm{intra}^{\mathrm{i}}(sequence)$ was verified and that (CONDITION S4) is valid:
  (CONDITION S4.) The intrapattern aborting dependency holds and there are elements in the sequence that verify:

$$\begin{cases} \exists E_\ell | \ell \in [i.a.k \,..\, i.k+j] \text{ we have : } DR(E_\ell).state = \\ \texttt{Executing} \text{ and } DR(E_\ell).vitality = \texttt{vital}. \end{cases}$$

The verification of (CONDITION S4) implies that the validity of $\mathrm{intra}^{\mathrm{a}}(sequence)$ and that all the elements that verified (CONDITION S4) are aborted.

- $\mathrm{intra}^{\mathrm{cp}}(sequence)$ denotes an intracompensation dependency in a pattern; it requires that $\mathrm{intra}^{\mathrm{i}}(sequence)$ was verified and that (CONDITION S5) is valid:
  (CONDITION S5.) The intrapattern compensation dependency is satisfied and there are elements in the sequence that verify:

$$\begin{cases} E_\ell \in | \ell \in [i.a.k \,..\, i.k+j], \text{ we have: } DR(E_\ell).state = \\ \texttt{Committed} \text{ and } DR(E_\ell).vitality = \texttt{vital}. \end{cases}$$

The verification of (CONDITION S5) implies that the validity of $\mathrm{intra}^{\mathrm{cp}}(sequence)$ is verified and that all the elements that verified (CONDITION S5) are compensated for.

To explain compensation performance, assume that we have the following sequence from $WS\text{-}SAGAS_{i.a}$; we note that an element placed between "[]" is actually a `compensatable` element:

$$sequence : [E^v_{i.a.k}]; [E^v_{i.a.k+1}]; \ldots; E^v_{i.a.\partial}; \ldots; [E^v_{i.a.k+j}]$$

The execution of the different elements verifies:

$$\begin{cases} \forall E_{i.a.\ell} | k \leq \ell < \partial, DR(E_\ell).state = \texttt{Committed} \text{ and} \\ \forall E_{i.a.\ell} | \partial < \ell \leq k+j, DR(E_\ell).state = \texttt{Executing} \end{cases}$$

When $\mathrm{intra}^{\mathrm{cp}}(sequence)$ is verified, the sequence execution is:

$$[E^v_{i.a.k}; E^v_{i.a.k+1}; \ldots; E^v_{i.a.\partial}; \ldots; E^v_{i.a.k+j}; \ldots; E'^v_{i.a.k+1}; E'^v_{i.a.k}]$$

We assume that the execution of every compensating element, such as $E'^v_{i.a.k+1}$, is successful and does not fail. Its execution is performed by executing a previously mapped WS that can reverse the effects of the WS that was mapped to $E^v_{i.a.k+1}$. The failure of compensation is considered beyond the scope of this paper because it remains an unresolved complex issue.

PARALLEL PATTERN $([E_{i.a.k}]; ([E_{i.a.k+1}] || \ldots || [E_{i.a.k+j}]))$ is the notation of this pattern. Let $\mathscr{Q}(E_{i.a.k})_{\mathrm{succ}}$ be the set of all the elements that are directly ordered after $E_{i.a.k}$ and that are presumed to be executed concurrently, and $\mathscr{S}(E_{i.k})_{\mathrm{succ}}$ be a subset that only contains the subset of elements that is executed effectively. The content of $\mathscr{S}(E_{i.k})_{\mathrm{succ}}$ depends on the aggregation pattern semantics (Figure 5.2(b)). For a parallel pattern, all the elements in $\mathscr{Q}(E_{i.a.k})_{\mathrm{succ}}$ must be activated after $E_{i.a.k}$. This means that: $\mathscr{S}(E_{i.a.k})_{\mathrm{succ}} = \mathscr{Q}(E_{i.a.k})_{\mathrm{succ}}$.

The activation of the execution of the elements in the set $\mathscr{S}(E_{i.a.k})_{\mathrm{succ}}$ requires that $E_{i.a.k}$ successfully terminates its execution (i.e., $DR(E^v_{i.a.k}).state = \texttt{Committed}$), where it is `vital`. Otherwise, it may terminate in any other state, without affecting the execution progress.

Assume that the set $\mathscr{S}(E_{i.a.k})_{\mathrm{succ}}$ contains one or several `vital noncompensatable` elements. For this pattern to commit, a special synchronization mechanism needs to be added to inform the different elements of the progress of the other `vital noncompensatable` elements in the same pattern. The synchronization mechanism must guarantee that either all or none of the `vital noncompensatable` elements are committed.

The commitment of this pattern depends on the `vital noncompensatable` elements' execution progress.

More formally, the intrapattern commitment dependency $intra^c(parallel)$ is specified as:

- $\mathrm{intra}^{\mathrm{c}}(parallel)$ verification requires that (CONDITION P1) and (CONDITION P2) are valid:
  (CONDITION P1.) The set of elements can attempt to commit ***iff***

$$\begin{cases} \{\forall E_{i.a.\partial} \in \mathscr{S}(E_{i.a.k})_{\mathrm{succ}}\} \text{ verifying:} \\ DR(E_{i.a.\partial}).behavior = \texttt{non-compensatable} \text{ and} \\ DR(E_{i.a.\ell}).vitality = \texttt{vital}, \text{ we have:} \\ DR(E_{i.a.\partial}).state = \texttt{Committed}. \end{cases}$$

  (CONDITION P2.) If the previous condition is valid, then the parallel pattern can be committed ***iff***

$$\begin{cases} \forall E_{i.a.\ell} \in \mathscr{S}(E_{i.a.k})_{\mathrm{succ}} \text{ and } DR(E_{i.a.\ell}).vitality = \texttt{vital}, \\ \text{we have: } DR(E_{i.a.\ell}).state = \texttt{Committed}. \end{cases}$$

If the set of elements $\mathscr{S}(E_{i.a.k})_{\text{succ}}$ does not encompass any `vital noncompensatable` element, then the intrapattern commitment dependency verification requires only the satisfaction of (CONDITION P2).

If (CONDITION P2) is not verified, that is:

$$\begin{cases} \exists E_{i.a.\ell} \in \mathscr{S}(E_{i.a.k})_{\text{succ}} \text{ verifying:} \\ DR(E^{v}_{i.a.\ell}).state = \texttt{Failed}, \end{cases}$$

then the conditions, (CONDITION P3) and (CONDITION P4), have to be evaluated, and subsequently, a backward recovery is triggered corresponding to the definition below for a parallel pattern that combines `vital noncompensatable` elements.

An extreme situation is when all the elements in the set $\mathscr{S}(E_{i.a.k})_{\text{succ}}$ are `nonvital`.

Even if all the elements fail, the intrapattern commitment dependency is deduced and the failure of this WS-SAGAS has no effect on the overall process, as a `nonvital` WS-SAGAS success is not crucial for the overall process commitment.

Assume that one or more elements from the set of `vital noncompensatable` elements were attempted a number of times with different WS but none of those attempts was successful. Similar to the sequence pattern, these elements are assumed to be failed and a *backward recovery* is triggered, which implies the verification of the intrapattern interruption dependency of this parallel pattern; formally:

- $\text{intra}^{\text{i}}(parallel)$ verification requires that (CONDITION P3) is valid:
  (CONDITION P3.) The execution of the parallel pattern is interrupted ***iff***

$$\begin{cases} \exists E^{v}_{i.a.\partial} \in \mathscr{S}(E_{i.a.k})_{\text{succ}} \text{ that verifies:} \\ DR(E_{i.a.\partial}).behavior = \texttt{non-compensatable}, \\ \text{we have: } DR(E_{i.a.\partial}).state = \texttt{Failed}. \end{cases}$$

Depending on the execution progress of all the other concurrent elements combined in the same pattern, the verification of $\text{intra}^{\text{i}}(parallel)$ may trigger an intrapattern compensation dependency, an intrapattern aborting dependency, or both. Formally:

- $\text{intra}^{\text{a}}(parallel)$ denotes an intrapattern aborting dependency; it requires that $\text{intra}^{\text{i}}(parallel)$ was verified and that (CONDITION P4) is valid:
  (CONDITION P4.) The intrapattern aborting dependency holds and there are elements in $\mathscr{S}(E_{i.a.k})_{\text{succ}}$ that verify:

$$\begin{cases} \exists E_{i.a.\ell}| \in \mathscr{S}(E_{i.a.k})_{\text{succ}} \text{ we have: } DR(E_{i.a.\ell}).state = \\ \texttt{Executing} \text{ and } DR(E_{\ell}).vitality = \texttt{vital}. \end{cases}$$

The verification of (CONDITION P4) implies the validity of $\text{intra}^{\text{a}}(parallel)$. It entails that all the elements that verified (CONDITION P4) are aborted.

- $\text{intra}^{\text{cp}}(parallel)$ denotes an intrapattern compensation dependency; it requires that $\text{intra}^{\text{i}}(parallel)$ was verified and that (CONDITION P5) is valid:
  (Condition P5.) The intrapattern compensation dependency holds and we have:

$$\begin{cases} \exists E_{i.a.\ell} \in \mathscr{S}(E_{i.a.k})_{\text{succ}} \text{ that verify:} \\ DR(E_{i.a.\ell}).behavior = \texttt{compensatable}, \\ DR(E_{i.a.\ell}).state = \texttt{Committed}, \text{ and} \\ DR(E_{i.a.\ell}).vitality = \texttt{vital}. \end{cases}$$

The verification of (CONDITION P5) implies that the validity of $\text{intra}^{\text{cp}}(parallel)$ is verified and that all the elements that verified (CONDITION P5) are compensated for. We describe below how compensation is performed: Assume that the only `vital noncompensatable` element is $E^{v}_{i.a.\partial}$ and it has failed. Assume also that:

$$\begin{cases} \forall E_{i.a.\ell} \in \mathscr{S}(E_{i.a.k})_{\text{succ}} - E_{i.a.\partial}, \text{ we have: } DR(E_{i.a.\ell}).vitality \\ = \texttt{vital} \text{ and } DR(E_{i.a.\ell}).behavior = \texttt{compensatable}. \end{cases}$$

If the execution progress of these elements verifies:

$$\begin{cases} \forall E_{i.a.\ell} \in \mathscr{S}(E_{i.a.k})_{\text{succ}} - E_{i.a.\partial}, \text{ we have:} \\ DR(E_{i.a.\ell}).state = \texttt{Committed}. \end{cases}$$

then, when $\text{intra}^{\text{cp}}(parallel)$ is verified, the compensation order is:

$$[E^{v}_{i.a.k}; ((E^{v}_{i.a.k+1}||...E^{v}_{i.a.\partial}...||E^{v}_{i.a.k+j}); (E'^{v}_{i.a.k+j}||...||E'^{v}_{i.a.k+1}))$$

We assumed that $E^{v}_{i.a.k}$ is `vital` and `compensatable`. We emphasize that this element can be `nonvital`; however, it must be `compensatable`. Otherwise, a backward recovery would not be possible, because a noncompensatable element's effects, once committed, cannot be undone. If the underpinning process logic requires a parallel pattern with $E_{i.a.k}$ noncompensatable, a plausible solution is to insert an idempotent between $E_{i.a.k}$ and the set of elements to be executed concurrently.

SELECTION PATTERN $([E_{i.a.k}] \circ ([E_{i.a.k+1}]||...||[E_{i.a.k+j}]))$. This pattern is a special case of the parallel pattern, where at *least one* and *at most all* the elements from $\mathscr{Q}(E_{i.a.k})_{\text{succ}}$ could be selected. After executing the element $E_{i.a.k}$, a selection condition is evaluated to choose from the set of its direct successors $\mathscr{Q}(E_{i.a.k})_{\text{succ}}$. Building on the assumption of equal probabilities for the different choices, $E_{i.a.k}$ can choose from $\mathscr{P}(\mathscr{Q}(E_{i.a.k})_{\text{succ}})$, the power set of $\mathscr{Q}(E_{i.a.k})_{\text{succ}}$, which is the set of all subsets of $\mathscr{Q}(E_{i.a.k})_{\text{succ}}$ (Figure 5.2(c)).

The execution commitment, interruption, aborting, and compensation obey the same dependencies defined for the parallel pattern. The only difference is that the chosen set of elements from $\mathscr{P}(\mathscr{Q}(E_{i.a.k})_{\text{succ}})$ must verify the conditions:

$$\{ \mathscr{S}(E_{i.a.k})_{\text{succ}} \subseteq \mathscr{Q}(E_{i.a.k})_{\text{succ}} \quad \text{and} \quad \mathscr{S}(E_{i.a.k})_{\text{succ}} \neq \emptyset \ .$$

SWITCH PATTERN $([E_{i.a.k}] \lhd ([E_{i.a.k+1}]||\ldots||[E_{i.a.k+j}]))$. Similarly, this pattern is a specialization of the *Selection pattern* (see Figure 5.2(d)). It differs in that *only one* element can be chosen from $\mathscr{Q}(E_{i.a.k})_{\text{succ}}$, the set of elements that comes directly after $E_{i.a.k}$. Similarly, by defining $\mathscr{S}(E_{i.a.k})_{\text{succ}}$ as the subset chosen from $\mathscr{P}(\mathscr{Q}(E_{i.k})_{\text{succ}})$, it must verify the following conditions:

$$\begin{cases} \mathscr{S}(E_{i.a.k})_{\text{succ}} \subseteq \mathscr{Q}(E_{i.a.k})_{\text{succ}},\ \mathscr{S}(E_{i.a.k})_{\text{succ}} \neq \emptyset \text{ and} \\ |\mathscr{S}(E_{i.a.k})_{\text{succ}}| = 1 \ . \end{cases}$$

The execution commitment, interruption, abortion, and compensation obey the same dependencies defined for the parallel pattern.

RENDEZVOUS PATTERN $(([E_{i.a.k}]||\ldots||[E_{i.a.k+j}])\diamond[E_{i.a.l}])$. Assume that $\mathscr{Q}(E_{i.a.l})_{\text{pre}}$ is the set of elements that are the direct predecessors of the element $E_{i.a.l}$. This pattern restricts the commitment of a set of elements executed in parallel as follows: all the `vital` elements in $\mathscr{Q}(E_{i.a.l})_{\text{pre}}$ must be committed for $E_{i.a.l}$ to start execution. Therefore, the element $E_{i.a.l}$ activation requires that the $\text{intra}^{c}(parallel)$ dependency of the elements in $\mathscr{Q}(E_{i.a.l})_{\text{pre}}$ is verified. When $\text{intra}^{c}(parallel)$ is verified, $E_{i.a.l}$ execution starts.

The commitment of this pattern depends on $E_{i.a.l}$ progress; the intrapattern commitment dependency is formulated as follows:

- $\text{intra}^{c}(rendezvous)$ verification requires that (CONDITION R1) is valid:
  (CONDITION R1.) The execution commitment can be deduced ***iff*** :

$$\begin{cases} DR(E^{v}_{i.a.l}).state = \texttt{committed} \text{ or} \\ DR(E_{i.a.l})^{\bar{v}}.state = \texttt{failed}; \end{cases}$$

Assume that $E_{i.a.l}$ is `compensatable` and `vital`, and a failure that could not be resolved by a forward recovery occurred. In such a case, a backward recovery must be triggered. A backward recovery mechanism requires undoing the effects of all the elements in $\mathscr{Q}(E_{i.a.l})_{\text{pre}}$. However, this may not be possible in the case where we have $\mathscr{Q}(E_{i.a.l})_{\text{pre}}$ verifying the following condition:

$$\begin{cases} \exists E_{\partial} \in \mathscr{Q}(E_{i.a.l})_{\text{pre}} \text{ verifying:} \\ DR(E_{\partial}).behavior = \texttt{non-compensatable} \end{cases}$$

In this paper, we assume that in this pattern, $\mathscr{Q}(E_{i.a.l})_{\text{pre}}$ includes only `compensatable` elements.

SELECTIVE MERGE PATTERN $(([E_{i.a.k}]\star\ldots\star[E_{i.a.k+j}])\square\!\!\rightarrow [E_{i.a.l}])$. Consider the case where *two or more* elements come together *but* without synchronization. Assume no elements are ever executed in parallel (Figure 5.2(g)).

Let $\mathscr{Q}(E_{i.a.l})_{\text{pre}} = \{E_{i.a.k},\ldots,E_{i.a.k+j}\}$ be the set of all the elements that are the direct predecessors of the element $E_{i.a.l}$. The execution of $E_{i.a.l}$ cannot be activated unless either of the elements appertaining to $\mathscr{Q}(E_{i.a.l})_{\text{pre}}$ has terminated.

On every element termination, $E_{i.a.l}$ is activated again. Let $\mathscr{S}(E_{i..a.l})_{\text{pre}}$ be the subset chosen from $\mathscr{P}(\mathscr{Q}(E_{i.a.l})_{\text{pre}})$ and let $\lambda = |\mathscr{S}(E_{i.a.l})_{\text{pre}}|$. In this case, λ represents the upper bound of the interval of time the element $E_{i.a.l}$ can be activated.

As $\mathscr{S}(E_{i.a.l})_{\text{pre}}$ can combine `vital` and `nonvital` elements, the condition for the activation of the execution of $E_{i.a.l}$ varies because a successful termination for a `vital` element is equivalent to a commitment; nevertheless, for a `nonvital` element it can be any other state. More formally:

Assume that at least one of the elements in $\mathscr{S}(E_{i.a.l})_{\text{pre}}$ is `vital`, that is, the following condition is verified:

$$\{\exists E_{\partial} \in \mathscr{S}(E_{i.a.l})_{\text{pre}} | DR(E_{\partial}).vitality = \texttt{vital}\}$$

If the above condition is verified, the activation condition for $E_{i.a.l}$ is verified and its execution is started every time the following condition is valid:

$$\begin{cases} E_{\partial} \in \mathscr{S}(E_{i.a.l})_{\text{pre}} | DR(E_{\partial}).vitality = \texttt{vital} \\ \text{we have: } DR(E_{\partial}).state = \texttt{committed} \ . \end{cases}$$

To ensure consistent execution, $E_{i.a.l}$ needs to know the cardinality λ of $\mathscr{S}(E_{i.a.l})_{\text{pre}}$. This can be deduced by referring to *OR*'s content. An incremental counter needs to be increased on every activation of $E_{i.a.l}$. When this counter reaches λ, the pattern commitment is verified. The intrapattern commitment dependency is formulated as follows:

- $\text{intra}^{c}(selective\ merge)$ verification requires (CONDITION SM1) to be valid:
  (CONDITION SM1.) The pattern commitment can be deduced ***iff*** :

$$\begin{cases} DR(E^{v}_{i.a.l}).state = \texttt{committed} \text{ or} \\ DR(E_{i.a.l})^{\bar{v}}.state = \texttt{failed} \text{ (i.e., because } E_{i.a.l} \text{ can be } vital \\ \text{or not); and the counter reached } \lambda. \end{cases}$$

If (CONDITION SM1) is verified, a backward recovery is triggered. The backward recovery includes all the elements in $\mathscr{S}(E_{i.a.l})_{\text{pre}}$. Depending on their vitality degree and execution progress, an intrapattern compensation dependency and/or abortion dependency may be triggered, the same as that defined for the previous parallel pattern. However, a critical situation may occur when $E_{i.a.l}$ is activated λ times and some of these activations fail and require a backward recovery. In such a situation, inconsistencies occur, especially if $E_{i.a.l}$ is `noncompensatable`. To deal with this situation, we assume that $E_{i.a.l}$ and all the elements in $\mathscr{S}(E_{i.a.l})_{\text{pre}}$ are `noncompensatable` and in the first failure of $E_{i.a.l}$, the whole pattern failure is deduced and a backward recovery is performed. $E_{i.a.l}$ is compensated first, then the different elements in $\mathscr{S}(E_{i.a.l})_{\text{pre}}$ are either compensated or aborted, in view of their execution progress. This is performed under the same conditions as formulated for the parallel pattern.

EXCLUSIVE MERGE PATTERN $(([E_{i.a.k}]\star\ldots\star[E_{i.a.k+j}]) \Box\!\!\Rightarrow [E_{i.a.l}]))$ is a point in a WS-SAGAS where the execution of two or more elements converge *but* without synchronization (see Figure 5.2(f)). In contrast to the *selective merge pattern*, this pattern assumes that only one element is executed and its execution success triggers the direct successor $E_{i.a.l}$ only *once*. $\mathscr{Q}(E_{i.a.l})_{\text{pre}}$ contains the elements within the scope of this pattern and is where any of them may trigger the execution of $E_{i.a.l}$; we assume that all the elements from $\mathscr{Q}(E_{i.a.l})_{\text{pre}}$ have the same probability of triggering the execution of $E_{i.a.l}$.

Activation of the execution of $E_{i.a.l}$ requires the verification of either of the two following conditions:

- Assume that $\mathscr{Q}(E_{i.a.l})_{\text{pre}}$ contains only `vital` elements:
  (CONDITION EM1.) The activation of the execution of $E_{i.a.l}$ requires that only one `vital` element has been committed and it is the first to be committed; more formally:

$$\begin{cases} \exists DR(E^{v}_{i.a.\ell}).state = \texttt{committed} \text{ and} \\ \forall E^{v}_{i.a.\partial} \in \mathscr{Q}(E_{i.a.l})_{\text{pre}} - \{E^{v}_{i.a.\ell}\}, \text{ we have:} \\ DR(E^{v}_{i.a.\ell}).state \neq \texttt{committed}. \end{cases}$$

- Assume that $\mathscr{Q}(E_{i.a.l})_{\text{pre}}$ contains only `nonvital` elements:
  (CONDITION EM2.) The activation of the execution of $E_{i.a.l}$ requires that only one `nonvital` element has terminated and it is the first to do so; more formally:

$$\begin{cases} \exists DR(E^{\bar{v}}_{i.a.\ell}).state \in \{\texttt{committed},\texttt{failed},\texttt{aborted}, \\ \texttt{compensated}\} \text{ and } \forall E^{\bar{v}}_{i.a.\partial} \in \mathscr{Q}(E_{i.a.l})_{\text{pre}} - \{E^{\bar{v}}_{i.a.\ell}\}, \\ \text{ we have: } DR(E^{\bar{v}}_{i.a.\partial}).state \in \{\texttt{executing},\texttt{waiting}\}. \end{cases}$$

Upon satisfaction of either of the above conditions, the execution of $E_{i.a.l}$ can be started. The commitment of this pattern depends on $E_{i.a.l}$ progress.

- $\text{intra}^{c}(\textit{exclusive merge})$ verification requires that the following condition, (CONDITION EM1), is valid:
  (CONDITION EM1) The pattern commitment can be deduced ***iff*** :

$$\begin{cases} DR(E^{v}_{i.a.l}).state = \texttt{committed} \text{ or} \\ DR(E_{i.a.l})^{\bar{v}}.state = \texttt{failed}; \end{cases}$$

If $E_{i.a.l}$ is `compensatable` and `vital` and a failure that cannot be handled using forward recovery occurs, the same assumption made for the rendezvous pattern applies for failure handling.

ITERATIVE PATTERN $([E_{i.a.k}];\lambda[E_{i.a.k+1}])$ is a point in a WS-SAGAS execution where the execution of a particular element $E_{i.k+1}$ must be repeated λ times (Figure 5.2(h)). The number of iterations depends on the process semantics. This pattern is a special case of the selective merge pattern; the only difference is when $\mathscr{S}(E_{i.a.l})_{\text{pre}}$ is a set that contains only one element. It follows that the processing is the same as for the selective merge pattern, if we replace $E_{i.a.k}$ by $\mathscr{S}(E_{i.a.l})_{\text{pre}}$ and $E_{i.a.k+1}$ by $E_{i.a.l}$.

5.4.2 WS-SAGAS Patterns Correct Structuring

We define a set of aggregation patterns that combines a collection of elements in different ways. A WS-SAGAS is defined by connecting a number of patterns in order to satisfy a particular business rule logic. Putting together different patterns permits the definition of a wide range of process-underpinning semantics. However, some of these pattern combinations may lead to inconsistencies in the control flow. To avoid this, we need to differentiate the permissible pattern combinations from the pattern combinations that may cause inconsistencies. Moreover, we need to define the correct order of combination. To this end, because the process logic is encompassed in the different OR it defines, we need to define the permissible combinations that we use to say if an OR is correct or if it has consistency conflicts.

Let $pattern_1$ and $pattern_2$ be two patterns to be defined in *WSpattern* and that have overlapping scopes (i.e., they come one after the other and have overlapping scope of elements); to obtain a correctly structured WS-SAGAS, the designer must observe several restrictions:

- If $pattern_1 = parallel$, then $pattern_2$ can be either a *rendezvous* pattern or a *selective merge* pattern.
- If $pattern_1 = switch$, then $pattern_2$ can be only an *exclusive merge* pattern.
- If $pattern_1 = selection$, then $pattern_2$ can be either a *selective merge* pattern or an *exclusive merge* pattern.

5.4.3 WS-SAGAS Subtransactions Execution Semantics

Every WS-SAGAS forms a collection of elements assembled following different aggregation patterns. Therefore, the execution of the WS-SAGAS depends on the execution of the different patterns it composes. We formulate the execution semantics of a WS-SAGAS in terms of *intra-WS-SAGAS dependencies* and we define four types of dependencies. Let $WS\text{-}SAGAS_{i.a}$ be a subtransaction and $pattern_{i.a}$ its ordered set of patterns described by $OR(WS\text{-}SAGAS_{i.a})$. Let $pattern_{i.a_1}$ be the first pattern in $pattern_{i.a}$, and $pattern_{i.a_l}$ the last pattern.

$\text{intra}^{ac}(WS\ SAGAS_{i.a})$ is an intra-WS-SAGAS activation dependency. It places conditions on the different intrapattern dependencies formulated for the different patterns in $pattern_{i.a}$. Let $pattern_{i.a_1}$ and $pattern_{i.a_2}$ be two consecutive patterns from $WS\text{-}SAGAS_{i.a}$. For $pattern_{i.a_2}$ execution to be activated, $pattern_{i.a_1}$ must have terminated its execution. We do not restrict the termination to a successful commitment because a pattern can be a collection of `nonvital` elements.

$\text{intra}^{c}(WS\text{-}SAGAS_{i.a})$ is an intra-WS-SAGAS commitment dependency. It places conditions on the different intrapattern dependencies formulated for the different patterns in $pattern_{i.a}$. A WS-SAGAS can commit if all the patterns that contain at least one `vital` element are committed. More formally, $\text{intra}^{c}(WS\text{-}SAGAS_{i.a})$ is valid ***iff***:

$$\begin{cases} \forall pattern_{i.a\ell} \in pattern_{i.a} \text{ where } \exists E^{v}_{i.a.\partial} \in pattern_{i.a\ell}, \\ \text{ we have: } \text{intra}^{c}(pattern_{i.a_\ell}) \text{ is verified.} \end{cases}$$

If any of the patterns in $WS\text{-}SAGAS_{i.a}$ that contain at least one `vital` element were interrupted by a failure and a backward recovery was triggered, then the WS-SAGAS failure is deduced. More formally, $\text{intra}^{\text{i}}(WS\text{-}SAGAS_{i.a})$ is valid ***iff***

$$\begin{cases} \exists pattern_{i.a_\ell} \in pattern_{i.a} \text{ where } \exists E^{v}_{i.a.\partial} \in pattern_{i.a_\ell}, \\ \text{we have: } \text{intra}^{\text{i}}(pattern_{i.a_\ell}) \text{ is verified.} \end{cases}$$

Assume that $pattern_{i.a_1}$ and $pattern_{i.a_2}$ are two consecutive patterns from $WS\text{-}SAGAS_{i.a}$, and $pattern_{i.a_2}$ is verifying the above condition. Therefore, $\text{intra}^{\text{i}}(WS\ SAGAS_{i.a})$ is verified and it requires that all the patterns in $pattern_{i.a}$ have to recover.

Every pattern activates implicitly the intrapattern interruption dependency of its predecessor when its own intrapattern interruption dependency is verified and terminated. This is ensured by every successive pattern having overlapping scopes.

5.4.4 WS-SAGAS Nesting Semantics

In the description of all the patterns semantics, we assumed that all the elements were `atomic`. However, we have defined our process with multinesting levels where an element can be at the same time part of one WS-SAGAS and parent of another WS-SAGAS. The element $E_{i.a}$ is included in $WS\text{-}SAGAS_i$ and therefore it is regarded as `atomic` elements in the same nesting level, that is, $\{\forall E_\ell \in CR(WS\text{-}SAGAS_i) - \{E_{i.a}\}\}$, E_ℓ does not know that $E_{i.a}$ is `composite`. That is, if the execution progress of the elements in $WS\text{-}SAGAS_i$ reaches the `composite` element $E_{i.a}$ the execution of $WS\text{-}SAGAS_{i.a}$ is triggered. However, for the other elements in $WS\text{-}SAGAS_i$, we assume that the execution delegation is totally transparent in the sense that the other elements in $WS\text{-}SAGAS_i$ are only waiting for the execution of the element $E_{i.a}$.

On the other hand, $E_{i.a}$ is the parent of the subtransaction $WS\text{-}SAGAS_{i.a}$. Consequently, the commitment of $E_{i.a}$ in $WS\text{-}SAGAS_i$ is equivalent to the $\text{intra}^{\text{c}}(WS\text{-}SAGAS_{i.a})$. Therefore, in the intracommitment dependency of every pattern that has $E_{i.a}$ in its scope, we have to replace the condition: "$DR(E_{i.a}).state =$ `committed`" by "$\text{intra}^{\text{c}}(WS\text{-}SAGAS_{i.a})$ is verified". Similarly, all the intra-interruption, intracompensation, and intra-abortion dependencies for each pattern, including a `composite` element within its scope, should be revised likewise.

In the same way, another form of execution dependency is required to guarantee that the nesting relation between the WS-SAGAS forming a process is respected. We introduce another form of dependency, *inter-WS-SAGAS nesting dependency* to ensure that commitment of $WS\text{-}SAGAS_i$ depends on $WS\text{-}SAGAS_{i.a}$ and that failures of $WS\text{-}SAGAS_{i.a}$ should also be cascaded to $WS\text{-}SAGAS_i$. More formally:

Let $WS\text{-}SAGAS_\ell$ and $WS\text{-}SAGAS_\partial$ be two subtransactions in P_i (i.e., ℓ and ∂ are defined in $[i..i.a.*.b.*.c]$). There is an inter-WS-SAGAS nesting dependency between $WS\text{-}SAGAS_\ell$ and $WS\text{-}SAGAS_\partial$, we note:
$\text{inter}^{\text{n}}(WS\text{-}SAGAS_\ell, WS\text{-}SAGAS_\partial)$ ***iff:***
$\{\exists E_\partial \in CR(WS\text{-}SAGAS_\ell) |\ DR(E_\partial).type =$ `composite`$\}$.

5.4.5 Process Execution Semantics

We assume a peer-to-peer execution model of a process P_i depicted as a hierarchy of recursively nested WS-SAGAS, which in turn are collections of aggregated elements. We denote a process execution instance by P_i^x where x is ranging in $[1..\alpha]$ and α designates the number of invocations of the process. The execution of a process instance assumes a dynamic WS discovery and candidate selection. For each process execution instance, we have a set of *DR*, a set of *OR*, and a set of *CR*: $DR(P_i^x)[n_i, m_i]$, $CR(P_i^x)[n_i, m_i]$, and $CR(P_i^x)[n_i, m_i]$.

A successful termination of a process execution instance is reached when all the `vital` WS-SAGAS forming the hierarchy are successfully committed and that the invocation order of the collection of elements forming the hierarchy of WS-SAGAS is correct against the prescribed order. More formally, the following conditions are satisfied:

- $inter^c(P_i^x)$ is verified and
- $OR(P_i^x)[n_i, m_i]$ respected the same prescribed order defined in $OR(P_i)[n_i, m_i]$.

$\text{intra}^{\text{c}}(P_i^x)$ is an intraprocess commitment dependency. It puts conditions on the different intra-WS-SAGAS dependencies formulated for the different WS-SAGAS in P_i. A process can commit if all the `vital` WS-SAGAS are committed. More formally, $\text{intra}^{\text{c}}(P_i^x)$ is valid ***iff:***

$$\begin{cases} \forall WS\text{-}SAGAS_\ell \in P_i \text{ verifying } DR(WS\text{-}SAGAS_\ell).vitality = \\ \texttt{vital}, \text{ we have: } \text{intra}^{\text{c}}(\texttt{WS-SAGAS}_\ell) \text{ is verified.} \end{cases}$$

If any of the `vital` WS-SAGAS in P_i were interrupted by a failure and a backward recovery was triggered, then the whole WS-SAGAS failure is deduced. A failure is cascaded up and down the hierarchy. More formally, $\text{intra}^{\text{i}}(P_i)$ is valid ***iff:***

$$\begin{cases} \exists WS\text{-}SAGAS_\ell \in P_i \text{ verifying } DR(WS\text{-}SAGAS_\ell).vitality = \\ \texttt{vital}, \text{ we have: } \text{intra}^{\text{c}}(\texttt{WS-SAGAS}_\ell). \end{cases}$$

5.5 Illustrative Example

We specify the trip reservation process P_1 using our defined textual, graphical, and formal notations. The travel itinerary reservation process $P_1[n_1, m_1]$ is described as a hierarchy of WS-SAGAS composed of $n_1 = 6$ elements distributed over $m_2 = 2$ nesting levels. The first level is $WS\text{-}SAGAS_1$ that combines $n_{1,1}$ elements: a trip information registration ($E^v_{1.1}$), a flight-booking element ($E^v_{1.2}$), a hotel reservation element ($E^v_{1.3}$), and a car rental element ($E^{\bar{v}}_{1.4}$). The second nesting level is $WS\text{-}SAGAS_{1.3}$ that has as a parent $E^v_{1.3}$ that combines $n_{1,2}$ elements: a reserve room element ($E^v_{1.3.1}$) and a reserve restaurant element ($E^{\bar{v}}_{1.3.2}$). To each atomic element, a compensating element is defined: $E^{v\prime}_{1.1}$ is the compensating element of $E^v_{1.1}$, $E^{v\prime}_{1.2}$ is the compensating element of $E^v_{1.2}$, $E^{v\prime}_{1.3.1}$ is the compensating element of $E^v_{1.3.1}$, $E^{\bar{v}\prime}_{1.3.2}$ is the compensating element of $E^{\bar{v}}_{1.3.2}$, and $E^{\bar{v}\prime}_{1.4}$ is the compensating element of $E^{\bar{v}}_{1.4}$.

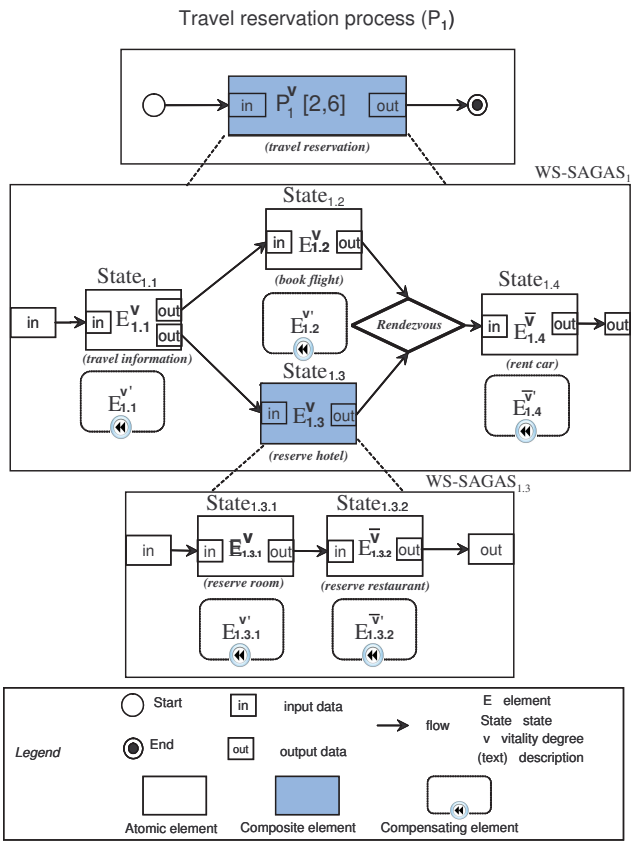


Fig. 5.3 WS-SAGAS graphical notation: example of a trip reservation process

We assume that a potential user of the process has to provide his desired destination, his desired departure and return dates, and his name. As QoS attributes, we assume that we are only interested in knowing the execution time and the reliability.

5.5.1 Textual Notation: $\mathscr{T}_1$

The textual notation of P_1 is formed by the triplet combining the list of *DR*, the list of *CR*, and the list of *OR*, as described below.

Definition Rules:

$$DR(P_1)[n_1,m_1] \equiv \bigcup_{\ell=1}^{1.3} DR(WS\text{-}SAGAS_\ell) \bigcup_{\partial=1.1}^{1.3.2} DR(E_\partial)$$
$$\equiv \left(DR(WS\text{-}SAGAS_1) \bigcup_{\partial=1.1}^{1.4} DR(E_\partial) \right)$$
$$\bigcup \left(DR(WS\text{-}SAGAS_{1.3}) \bigcup_{\partial=1.3.1}^{1.3.2} DR(E_\partial) \right) .$$

$$DR(WS\text{-}SAGAS_1): \langle name = \texttt{WS-SAGAS}_1,$$
$$description = \bigcup_{\partial=1.1}^{1.4} \texttt{DR}(\texttt{E}_\partial.\texttt{description}),$$
$$state = \bigcup_{\partial=1.1}^{1.4} \texttt{DR}(\texttt{E}_\partial.\texttt{state}), \texttt{vitality} = \texttt{vital},$$
$$behavior = \bigcup_{\partial=1.1}^{1.4} \texttt{DR}(\texttt{E}_\partial.\texttt{behavior})\rangle .$$

$$DR(E^v_{1.1}): \langle name = \texttt{E}_{1.1}, description = \texttt{travel information}, type = \texttt{atomic}, state = \texttt{Waiting},$$
$$vitality = \texttt{vital}, operation_1((in_1 = \texttt{destination},$$
$$in_2 = \texttt{depart}, in_3 = \texttt{return}, in_4 = \texttt{name}),$$
$$(out_1 = \texttt{destination}, out_2 = \texttt{depart},$$
$$out_3 = \texttt{return}, out_4 = \texttt{name})),$$
$$qos_1 = \texttt{reliability}, qos_2 = \texttt{executionTime}\rangle .$$

Composability Rules:

$$CR(P_1)[n_1,m_1] \equiv CR(WS\text{-}SAGAS_1) \bigcup CR(WS\text{-}SAGAS_{1.3}) .$$

$$CR(WS\text{-}SAGAS_1) \rightarrow \langle E^v_{1.1}, E^v_{1.2}, E^v_{1.3}, E^{\bar{v}}_{1.4}\rangle$$
$$CR(WS\text{-}SAGAS_{1.3}) \rightarrow \langle E^v_{1.3.1}, E^{\bar{v}}_{1.3.2}\rangle .$$

Ordering Rules:

$$OR(P_1)[n_1,m_1] \equiv OR(WS\text{-}SAGAS_1) \bigcup OR(WS\text{-}SAGAS_{1.3}) .$$

$$OR(WS\text{-}SAGAS_1) \rightarrow \langle E^v_{1.1}; (E^v_{1.2} || E^v_{1.3}) \diamond E^{\bar{v}}_{1.4}\rangle$$
$$OR(WS\text{-}SAGAS_{1.3}) \rightarrow \langle E^v_{1.3.1}; E^{\bar{v}}_{1.3.2}\rangle .$$

5.5.2 Graphical Notation: $\mathscr{G}_1$

(Figure 5.3) is an illustrative example of how a trip reservation process P_1 is specified using the WS-SAGAS transaction model graphical notation.

5.5.3 Formal Notation: $\mathscr{F}_1$

The formal notation of the process P_1 is described below using the syntax, in the spirit of CPS, that we defined:

$$P_1 = WS\text{-}SAGAS_1 \vdash WS\text{-}SAGAS_{1.3}$$
$$WS\text{-}SAGAS_1 = [E^v_{1.1}]; ([E^v_{1.2}] || [E^v_{1.3}] \diamond [E^{\bar{v}}_{1.4}])$$
$$= (E^v_{1.1} \div E^{v\prime}_{1.1}); (((E^v_{1.2} \div E^{v\prime}_{1.2}) || E^v_{1.3}) \diamond (E^{\bar{v}}_{1.4} \div E^{\bar{v}\prime}_{1.4}))$$
$$WS\text{-}SAGAS_{1.3} = [E^v_{1.3.1}]; [E^{\bar{v}}_{1.3.2}]$$
$$= (E^v_{1.3.1} \div E^{v\prime}_{1.3.1}); (E^{\bar{v}}_{1.3.2} \div E^{\bar{v}\prime}_{1.3.2}) .$$

6 THROWS Architecture

6.1 Motivations

The FENECIA approach defined the WS-SAGAS model to describe how a business process is specified. However, the WS-SAGAS remains only a specification because it dealt only with modeling and did not consider the evident need of a CWS enactment model to have a comprehensive methodology that tackles the WSC issue from all its different aspects. To this end, in the FENECIA approach, we investigated the issue of CWS execution.

Typically, a CWS can be organized in either a centralized or a distributed fashion. We refer to the execution mode as centralized when a single coordinator or engine, such as the BPWS4J engine, executes a CWS developed, for example using BPEL4WS [4]. In contrast to a distributed model where data are transferred directly between two points, in a centralized model all data must go through the coordinator. The coordinator may thereby become a performance bottleneck and constitute a single point of failure. In addition, allowing a large amount of irrelevant data to traverse the coordinator may overload the network and cause poor scalability and significant performance degradation.

To cope with the revealed inadequacy of the execution of CWS with a centralized control in FENECIA, we opted for a distributed model and we present *THROWS* architecture, which is an acronym for a *Transaction Hierarchy for Route Organization of Web Services*, where the composition execution control is distributed [19] over multiple *engines*, each allocated to an element from a process depicted as a hierarchy of recursively nested WS-SAGAS transactions. Rather than communicating through a central authority, the engines communicate directly with each other to transfer data and delegate execution control when required.

6.2 General Assumptions and Description of THROWS Architecture

In THROWS architecture, the execution control of a process P_i depicted as hierarchy of recursively nested WS-SAGAS is allocated to *dynamically discovered engines*. To ensure high availability of distributed execution of the CWS, we make available on each engine side the following information: the *CEL (Candidate Engines List)* and the *CEP (Current Execution Progress)*.

In THROWS, for each `atomic` element $E_{i.k}$ from a subtransaction $WS\text{-}SAGAS_i$, a *CEL* is generated by searching for WS that have capabilities satisfying the element $E_{i.k}$ functionalities. To each discovered WS, an engine is allocated, and together, they are stored as a couple, engine-ws, in the *CEL* of the element $E_{i.k}$, denoted as $CEL(E_{i.k})$. That is, $CEL(E_{i.k})$ is an ordered set of engine-ws couples, and every time an element is to be executed, $CEL(E_{i.k})$ is generated and an engine-ws couple is allocated. When allocated to execute a particular element $E_{i.k}$, the engine executes the WS it controls and that provides the required functionalities from the element $E_{i.k}$. Therefore, the engine is responsible of the invocation, execution and completion, failure information, and recovery of $E_{i.k}$.

An engine allocated to an element $E_{i.k}$ is denoted as $e^p_{i.k}$ and it controls a WS denoted as $ws^p_{i.k}$, where the subscript "$i.k$" is kept the same as the element to which it is allocated and the superscript "p" is a unique identifier ranging over $[1..|CEL(E_{i.k})|]$, with $|CEL(E_{i.k})|$ the cardinality of *CEL*.

We assume that there are two types of engines. The first type of engine is allocated to `atomic` elements and is responsible for the invocation, execution and completion, failure information, and recovery; we call this type *engine executor* and we denote it as $ee^p_{i.k}$. The second type of engines is responsible for `composite` elements. An engine allocated to a `composite` element controls an overall WS-SAGAS; we call this the *engine coordinator* and we denote it as $ec^p_{i.a}$.

Assume that for a particular value q of p in $ee^p_{i.k}$, the engine $ee^q_{i.k}$ committed successfully the element $E_{i.k}$. Then, in such a scenario, the engine $ee^q_{i.k}$ is responsible for generating the *CEL* of the direct successors of the element it controls; this is how the execution progresses in THROWS. However, if the engine fails, then a forward recovery can be attempted if there are other engines in the *CEL*; otherwise, a backward recovery is triggered.

The *CEP*, using the *state* concept defined in WS-SAGAS and with every element being updated with every state change, allows for information about failures to be collected and the backward recovery mechanism to be realized, as described in WS-SAGAS. On the other hand, the *CEL* allows an increase in the chances for the execution of a CWS commit by realizing the forward recovery mechanism, described in WS-SAGAS. In the future, both the *CEL* and *CEP*, because of their contents, can serve as a solid base for investigating and analyzing the reasons for failures that have occurred if they are stored in a history that collects the execution logs of different CWS.

To return to the engine coordinator, assume that $E_{i.a}$ is a `composite` element from the process described by (Equation 5.1). When the execution progress reaches $E_{i.a}$, an engine $ec^p_{i.a}$ is allocated to $E_{i.a}$. Because $E_{i.a}$ is the parent of the subtransaction $WS\text{-}SAGAS_{i.a}$, the engine $ec^p_{i.a}$ has to initiate the execution of $WS\text{-}SAGAS_{i.a}$.

The particularity of $ec^p_{i.a}$—as engine coordinator—is that it has no unique WS under its control; therefore, it does not invoke any WS. Instead, it has to generate the *CEL* of the first element in $WS\text{-}SAGAS_{i.a}$, that is, CEL of $E_{i.a.1}$. Then, an engine $ee^p_{i.a.1}$ is selected and the execution proceeds until it reaches the last element in $WS\text{-}SAGAS_{i.a}$; we assume this element is $E_{i.a.n_{i.2}}$ and it is allocated to an engine $ee_{i.a.n_{i.2}}$. This engine, when it terminates the execution of the element, returns execution control to the engine parent of the entire subtransaction $WS\text{-}SAGAS_{i.a}$, that is, to the engine coordinator $ec^p_{i.a}$. The engine coordinator $ec^p_{i.a}$ resumes execution termination of element $E_{i.a}$ by updating the state of $E_{i.a}$ in line with the overall state of $WS\text{-}SAGAS_{i.a}$.

We describe the coordination between engine coordinator and engine executor below in this section.

In describing our execution architecture, we only consider the case where all the elements in a process P_i are `compensatable`. The case of a process that includes one or more `noncompensatable` elements is addressed in our future work.

In searching for WS that match an element's functionalities, we assume only simple matching based on the element's predefined operations in its *DR* and the WS description included in its WSDL. We consider only simplified conditions that can be developed in a future work. We assume that we can determine easily and automatically whether a WS and an element are semantically equivalent. To date, assessing the similarity of WS to achieve the best match is an active area of research. We can apply one of the available proposals ranging from keyword-based methods to ontologies and reasoning algorithm-enriched methods. We consider the applied WS discovery and selection methods beyond the scope of this paper.

Definition 6.1 (Engine Executor ($ee^p_{i.k}$)) An engine executor $ee^p_{i.k}$ is allocated to control the execution of a WS $WS^p_{i.k}$ that provides capabilities satisfying the functionalities of a particular `atomic` element $E_{i.k}$ from a $WS\text{-}SAGAS_i$.

Because an engine is an entity that relates to the CWS execution, we define it using the following specialization of the generic *DR* introduced in the preceding section:

$$DR(ee^p_{i.k}) = \langle name, description, wsdlLink, ec^p_i, \\ (operation \times (in*, out*)) \times \rangle \ ,$$

where:

- *name* is necessary to identify the engine executor;
- *description* is a concise description of the capabilities of the WS controlled by the engine $ee^p_{i.k}$;
- *wsdlLink* is the location of the WSDL of the WS controlled by the engine $ee^p_{i.k}$;
- $(operation \times (in*, out*))$ are the different operations that a particular WS $WS^p_{i.k}$ can provide with their corresponding input and output parameters;
- every first and last engine executor controlling the first and last element in a subtransaction, respectively, must know the engine coordinator that controls the subtransaction they relate to.
  This is required because the WS-SAGAS execution starts by receiving the control from the engine coordinator, and when the execution of the last element terminates, this element has to inform the coordinator of its termination and of its execution results. However, because a WS-SAGAS execution may be subject to failure and interrupted before reaching the last element, ideally we must make this information available in all the elements. More precisely, in (Equation 5.1), assume that ec^p_i is the engine coordinator that initiated the execution of the subtransaction and that was allocated $WS\text{-}SAGAS_i$; then, in every engine executor controlling an element from $WS\text{-}SAGAS_i$, we have the information ec^p_i made available.

Definition 6.2 (Engine Coordinator ($ec^p_{i.a}$)) An engine coordinator (denoted $ec^p_{i.a}$) is allocated to control the execution progress of a `composite` element $E_{i.a}$ aggregated in a subtransaction $WS\text{-}SAGAS_i$ and the parent of another subtransaction $WS\text{-}SAGAS_{i.a}$. Because an engine is an entity that relates to the CWS execution, we define it using the following specialization of the generic *DR* introduced in the preceding section:

$$DR(ec^p_{i.a}) = \langle name, description, CR(WS\text{-}SAGAS_{i.a}) \rangle \ ,$$

where:

- *name* is necessary to identify the engine coordinator;
- *description* is a concise description of the functionalities of the engine $ec^p_{i.a}$. It verifies:

  $$DR(ec^p_{i.a}).description \equiv DR(WS\text{-}SAGAS_{i.a}).description$$

- $CR(WS\text{-}SAGAS_{i.a})$ has the same content as *CR* specified in the preceding sections. This content is required for the engine coordinator to know the elements it is responsible for invoking. In particular, as described above, $ec^p_{i.a}$ has to generate the *CEL* of the first element in $WS\text{-}SAGAS_{i.a}$; therefore, it requires full knowledge of $CR(WS\text{-}SAGAS_{i.a})$.

Definition 6.3 (The Candidate Engine List of an `Atomic` Element (CEL($E_{i.k}$))) For each `atomic` element $E_{i.k}$ from $WS\text{-}SAGAS_i$ we define *CEL* as the *list of candidate engines* potentially enabled to execute $E_{i.k}$ (i.e., they control the execution of WS providing the same semantics as $E_{i.k}$). Generating *CEL* is the responsibility of the direct predecessor of the element, that is, when the execution of $E_{i.k}$ by a certain engine $ee^q_{i.k}$ is committed. Then, the engine executor $ee^q_{i.k}$ must allocate the execution control to another engine to progress the process execution by generating the *CEL* of its direct successor; we assume it generates $CEL(E_{i.k+1})$, and then it selects an engine $ee^p_{i.k+1}$ and delegates the execution control to that engine.

Depending on the element it controls (i.e., order in the WS-SAGAS's *OR*), an engine may have to generate only one *CEL*, many *CELs*, or it may not have to generate any.

In addition, generating the *CEL* of a particular element might be the responsibility of only one engine, or it might be the responsibility of several engines. We show in what follows that distinguishing one case from another depends on the content of $OR(WS\text{-}SAGAS_i)$:

1. $(E_{i.k}\ op\ (E_{i.k+1}\ op\ E_{i.k+2}\ op\ \ldots\ op\ E_{i.k+j}))$ where every prefixed operator verifies $op \in \{||, \bigcirc, \lhd\}$: In this case, the engine responsible for executing the element $E_{i.k}$ on finishing successfully $E_{i.k}$ has to generate the *CELs* of all the elements in $\mathscr{S}(E_{i.k})_{\text{succ}}$ (the subset of elements chosen for execution from all the direct successors of $E_{i.k}$). If we assume that $ee^q_{i.k}$ is the engine that was allocated to $E_{i.k}$ and has committed it, then the generated *CELs* of all the elements in $\mathscr{S}(E_{i.k})_{\text{succ}}$ is denoted $CEL(\mathscr{S}(E_{i.k})_{\text{succ}})$, and the set of the selected engines (each for each element from $\mathscr{S}(E_{i.k})_{\text{succ}}$ is denoted $\mathscr{S}(e^q_{i.k})_{\text{succ}}$;

2. $((E_{i.k} op E_{i.k+1}\ op\ E_{i.k+2}\ op \ldots op\ E_{i.k+j})\ op\ E_{i.l})$ where every postfixed operator verifies $op \in \{\diamond, \square\!\!\rightarrow, \square\!\!\Rightarrow\}$: In this case, $CEL(E_{i.l})$ is generated by the engine(s) responsible for executing the element(s) in $\mathscr{S}(E_{i.l})_{\text{pre}}$ (the subset of elements being executed from all the direct predecessors of $E_{i.l}$); if we assume $ee^p_{i.l}$ to be the engine selected to execute $E_{i.l}$, then this set of engines is denoted $\mathscr{S}(ee^p_{i.l})_{\text{pre}}$;
3. As defined in (Equation 5.1), $E_{i.n_{i.1}}$ is the last element in this WS-SAGAS and does not have to generate any *CEL*.

We use the following notation for $CEL(E_{i.k})$:

$$CEL(E_{i.k}) = \left\{ DR(ee^1_{i.k}), DR(ee^2_{i.k}), \ldots, DR(ee^q_{i.k}), \ldots \right\} \quad (6.1)$$

where:

- the number of discovered engines in $CEL(E_{i.k})$ is variable and depends on WS availability.
- $DR(ee^q_{i.k})$ is assumed to be the *DR* of the engine executor that was allocated to $E_{i.k}$ and was successful in its execution. Every time the execution of an element $E_{i.k}$ fails, it is allocated a new engine from the $CEL(E_{i.k})$, defined with $DR(ee^p_{i.k})$ and $p \in [1..|CEL(E_{i.k})|]$.

Because each `nonvital` element has to be attempted only once, the cardinality of the different *CELs* of all the `nonvital` elements from $WS\text{-}SAGAS_i$ must verify the condition:

$$\{\forall E^{\bar{v}}_{i.k} | E^{\bar{v}}_{i.k} \in WS\text{-}SAGAS_i : |CEL(E^{\bar{v}}_{i.k})| = 1\}$$

However, because the successful commitment of all the `vital` elements is essential, the probability of their success must be increased by generating *CELs* verifying the condition:

$$\{\forall E^{v}_{i.k} | E^{v}_{i.k} \in WS\text{-}SAGAS_i : |CEL(E^{v}_{i.k})| > 1\}$$

It is possible that throughout the execution of the different instances of the same element, an engine generates a *CEL* that contains multiple candidate engines and that in almost all execution instances, the element execution was committed by the first allocated engine.

Consequently, the time spent in generating the *CEL* constitutes an overhead. It is required to determine for each element the most suitable value of the cardinality of *CEL*, which allows the element to be successfully terminated, but in addition, avoids the risk of having to trigger a recovery only because there is no available candidate engine. This can be possible by considering and analyzing the execution history of the different elements.

Definition 6.4 (The Replica Engines List of a `Composite` Element (REL($E_{i.a}$))) The *REL* of a `composite` element is defined in the same way as an *atomic* element. However, it does not require any WS discovery or selection. The main difference is that an *REL* contains the replicas from the same engine coordinator. That is, assume a composite element $E_{i.a}$ is the parent of a $WS\text{-}SAGAS_{i.a}$ in P_i. When the execution control reaches $E_{i.a}$, instead of generating a *CEL*, because this element is `composite` it makes its predecessor(s) generate $REL(E_{i.a})$, which is a list of replica engines. This avoids failure, because if one of the engines in $REL(E_{i.a})$ fails, another engine takes charge of the execution instead of the failed engine. The information in the different engines is continuously updated to avoid having any obsolete information. Moreover, any update/information that reaches any of the replica engine coordinators in $REL(E_{i.a})$ is transparently broadcast to all the others. We use the following notation for $REL(E_{i.a})$:

$$REL(E_{i.a}) = \left\{ DR(ec^1_{i.a}), DR(ec^2_{i.a}), \ldots, DR(ec^q_{i.a}), \ldots \right\} \quad (6.2)$$

where:

- the number of replica engines in $REL(E_{i.a})$ is variable and depends on the designer's judgment;
- $DR(ec^q_{i.a})$ is assumed to be the *DR* of the replica engine coordinator that was allocated to $E_{i.a}$, which was not subject to a failure, and which contains the last updated information concerning the execution progress of $WS\text{-}SAGAS_{i.a}$.

Definition 6.5 (The Current Execution Progress (CEP)) We define the concept of *CEP* to keep track of the execution progress of a process, depicted as a hierarchy of recursively nested WS-SAGAS. When an engine executor $ee^p_{i.k}$ executes an `atomic` element $E_{i.k}$, every change in that element's *state* has to be reflected on the copy of the *CEP*, stored locally on the engine executor $ee^p_{i.k}$ side. The *CEP* content is made available on each engine (executor and coordinator).

On every engine executor, only one type of *CEP* copy is available. Consider the process depicted in (Equation 5.1), where a copy of *CEP* of $WS\text{-}SAGAS_i$, which is stored on the engine executor $ee^p_{i.k}$, is formulated as:

$$CEP(WS\text{-}SAGAS_i, ee^p_{i.k}) : \left\{ \overrightarrow{DR}(E_{i.1})\ op \cdots op\ \overrightarrow{DR}(E_{i.k})\ op \cdots op\ \overrightarrow{DR}(E_{i.a})\ op \cdots op\ \overrightarrow{DR}(E_{i.n_{i.1}}) \right\},$$

where:

- $\overrightarrow{DR}(E_{i.k})$ is defined as the *Active DR* of the element $E_{i.k}$: it is equal to the *DR* of the element $E_{i.k}$ to which an attribute *engine* is added, indicating the name of the *currently allocated engine* to the element; this notation is used to indicate that the element $E_{i.k}$ is allocated to engine $ee^p_{i.k}$. The last value of the attribute *engine* is the value of the engine that either committed the element or failed to commit it.
- initially, the attribute $\overrightarrow{DR}(E_{i.k}).engine$ is set to `null` in all the elements' *active DR* in *CEP*.
- $E_{i.1}$ and $E_{i.n_{i.1}}$ are, respectively, the first and the last element of the subtransaction $WS\text{-}SAGAS_i$.
- *op* is the operator that connects the different element with $op \in \{||, ;, \star, \lhd, \diamond, \bigcirc, \square\!\!\rightarrow, \square\!\!\Rightarrow\}$. Depending on the considered operator, it can be prefixed (e.g., λ, $\lhd$, and $\bigcirc$), postfixed (e.g.,$\diamond$, $\square\!\!\rightarrow$, and $\square\!\!\Rightarrow$), or infixed (e.g.,$||$ and ;).

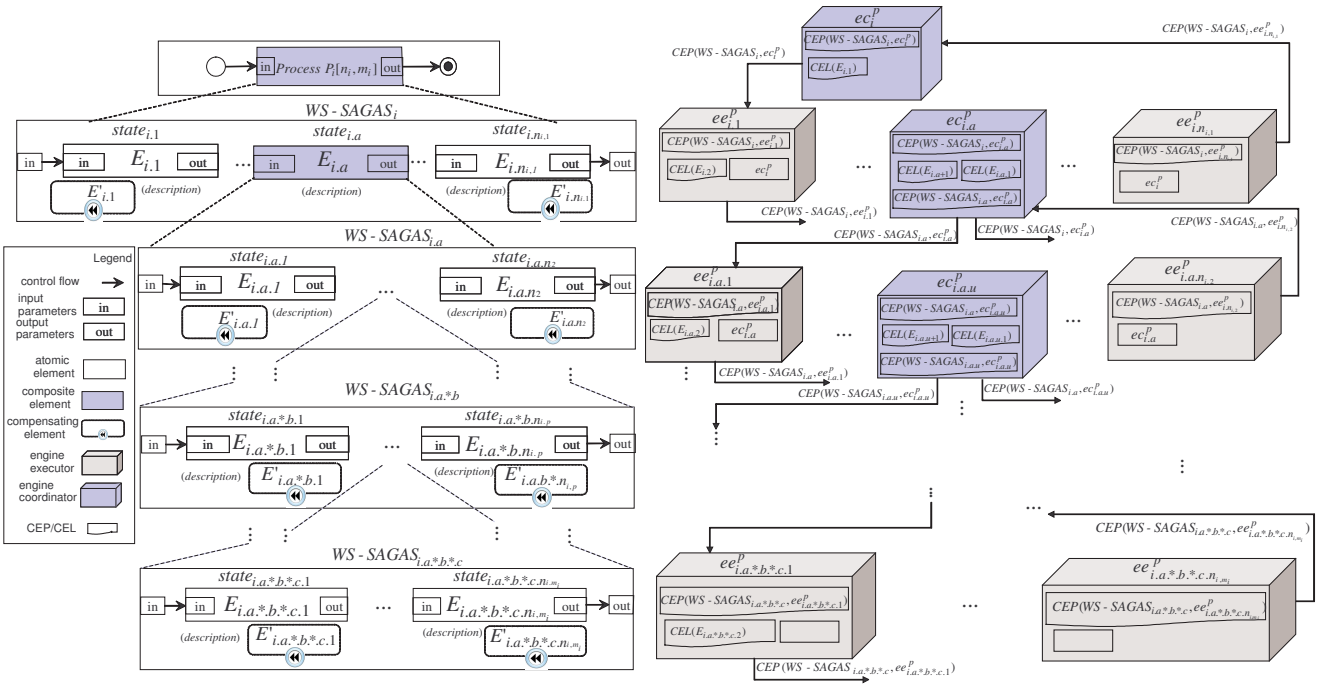


Fig. 6.1 WS-SAGAS transaction model execution in THROWS architecture: control delegation and CEP exchange between engine coordinator and engine executor

Because we consider a hierarchy of nested WS-SAGAS, we have different *CEP* expressions, one for every nesting level. More precisely, consider the process in (Equation 5.1). P_i is formed with m_i nesting levels. Therefore we have m_i different *CEP*, one for each WS-SAGAS.

Because we consider a peer-to-peer execution model, the only connection point between two subtransactions in two consecutive nesting levels, such as $WS\text{-}SAGAS_i$ in nesting level 1 and $WS\text{-}SAGAS_{i.a}$ in level 2, is the engine coordinator responsible for the `composite` element $E_{i.a}$, parent of $WS\text{-}SAGAS_{i.a}$ and part of $WS\text{-}SAGAS_i$. The important role of an engine coordinator becomes apparent here because every engine executor is the connection point level and constitutes, similar to the bridge that delegates the execution control from one level to another.

Assume that the engine coordinator $ec^p_{i.a}$ was allocated to the composite element $E_{i.a}$ from $WS\text{-}SAGAS_i$. On being allocated to execute the element $E_{i.a}$, this engine receives a copy of $CEP(WS\text{-}SAGAS_i, ee^p_{i.a-1})$ from its direct predecessor (i.e., assume it is an engine $ee^p_{i.a-1}$ allocated to an element $E_{i.a-1}$).

Because $E_{i.a}$ is composite, the engine coordinator $ec^p_{i.a}$ has to initiate the execution of $WS\text{-}SAGAS_{i.a}$; therefore, it has to generate a *CEL* of $E_{i.a.1}$ and must provide it with the copy of $CEP(WS\text{-}SAGAS_{i.a}, ec^p_{i.a})$. Therefore, on $ec^p_{i.a}$, we have two *CEPs*: the first is $CEP(WS\text{-}SAGAS_i, ee^p_{i.a-1})$ and the second is $CEP(WS\text{-}SAGAS_{i.a}, ec^p_{i.a})$. In this way we can start the execution of another nested level. The content of $CEP(WS\text{-}SAGAS_{i.a}, ec^p_{i.a})$ is described in the same way as $CEP(WS\text{-}SAGAS_i, ee^p_{i.a-1})$.

When the execution of $WS\text{-}SAGAS_{i.a}$ terminates, $ec^p_{i.a}$ has to resume execution control. On receiving a copy of $CEP(WS\text{-}SAGAS_{i.a}, ee^p_{i.a.n_{i.2}})$, it deduces its own state on the basis of the execution progress of the whole WS-SAGAS and has to generate a *CEL* for $E_{i.a+1}$—its direct successor in $WS\text{-}SAGAS_i$—select an engine, and allocate execution control to it; assume the allocated engine is $ee_{i.a+1}$, if $E_{i.a+1}$ is atomic. In this case, engine $ec^p_{i.a}$ sends to $ee_{i.a+1}$ a copy of $CEP(WS\text{-}SAGAS_i, ec^p_{i.a})$ (see Figure 6.1).

6.3 Collaboration between Peer Engines and Web Services

To execute a WS-SAGAS, the engines in THROWS architecture communicate by the different messages that we define. All the messages are sent in a peer-to-peer fashion, that is, from the source to the destination *without* going through any central entity, as they would for centralized execution. Consequently, performance bottlenecks should decrease.

In what follows, the distinction between an engine executor or coordinator is only made when the processing differs.

6.3.1 Conversation between Peer Engines:

Messages exchanged between peer engines contain:

(i) *The current execution progress (CEP):* After an engine is chosen to be in charge of a particular element, it has to be informed of the *CEP* content (last-updated version).

To guarantee that the necessary information for any potential recovery is still available, we assume that the *CEP* content is preserved in the side of every engine until the end of all the WS-SAGAS executions.
Therefore, even if a message sent between two engines is discarded or does not reach its destination for any reason (e.g., network broken, time out), it is possible to submit it again. We note that *CEP* contains the execution context (e.g., input/output variables, services invocation results) necessary for each WS invocation.

(ii) *The execution start signal, abortion request, and compensation request*: These messages enable a synchronized transactional execution of CWS. However, they are not sufficient. We also need special synchronization messages to control delegation of the execution control between engines to prevent race conditions occurring, and improper control flow signals may be triggered, leading to possible inconsistencies.

Considering that in our architecture, we are constrained by a distributed and loosely coupled environment, the distributed two-phase commit protocol (2PC) cannot apply, because the *Atomicity* property and the locking mechanisms on which it is founded are not required and a central monitor's existence is undesirable.

There are a number of proposed protocols for distributed and loosely coupled environments, including BTP [48], WS-transaction [25] with WS-coordination [49], and WS-CAF [26]. Not all the available WS support the same protocol, so instead of using any of these protocols, for flexibility in THROWS architecture we introduced the *predelegation phase*, the *synchronization phase*, the *peer-engines waiting period*, and the *engine-ws waiting period*:

- *The predelegation phase*: During this phase, a first entity (one or more engines) agrees that it will release execution control while a second entity (one or more engines) agrees that it is ready to accept delegation of the control. This phase is necessary at the beginning of each element execution. Introducing such a phase reduces the potential for cancellations (e.g., engine not ready) and improves the probability of successfully completing the composite WS execution. The first entity sends a control delegation request and it is assumed to receive a control delegation agreement from the second entity.
- *The synchronization phase*: This phase follows directly after the predelegation phase where one or more elements has to be executed in parallel. To ensure this, different engines, on receiving a start signal propagated by their direct common predecessor, start the execution simultaneously.
- *Peer engines waiting period*: This is the estimated period of time, after which the engine does not receive any confirmation or agreement, so the message for control delegation ignores that engine and chooses another engine from the *CEL* as a new candidate for control delegation; then, the *predelegation phase* is repeated.
- *Engine-web service waiting period*: It is most likely that an engine executor controlling a WS, for an unknown reason (e.g., failure) does not receive a response message from the WS for a time; in some cases, the waiting time, as an answer time may even tend to be infinite, which is unacceptable. To avoid this, a similar situation to the above occurs. We define this as the *engine-ws waiting period*. It avoids the engine executor waiting eternally for an answer that may never come: if the WS fails to respond after the *engine-ws waiting period* has elapsed and no information was received of the execution progress of the WS, then the engine executor must consider itself failed and a recovery must be triggered.

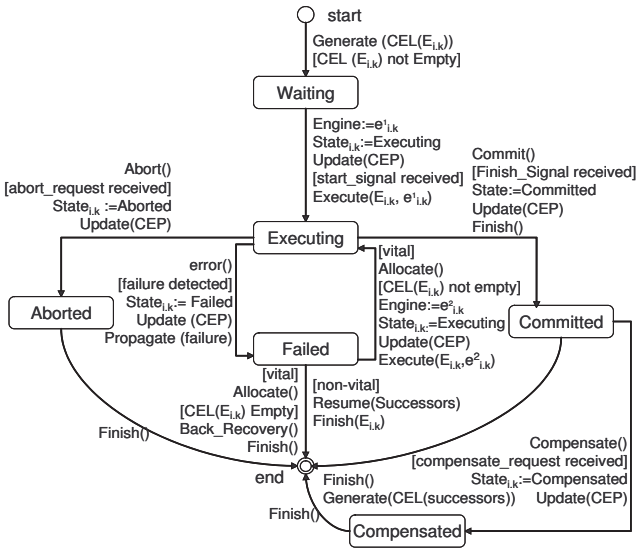


Fig. 6.2 State transition diagram of an `atomic compensatable` element executed in THROWS architecture

6.3.2 *Conversation between an Engine Executor and a WS:*

The different messages that are used to communicate between an engine executor and a WS are defined according to the WS execution progress and they are chiefly of two forms:

- The engine executor sends a notification of execution start to ask the WS it controls to start executing. At the same time it provides the WS with the necessary input data for its execution.
- When a WS finishes executing, it notifies the engine executor of its own execution results. In this case also, the *engine-ws waiting period*, which has to be estimated, is essential to avoid the engine executor finding itself eternally waiting for an answer that may not arrive. Therefore, after the *engine-ws waiting period* has elapsed, the engine executor has to be informed of the WS execution's progress. If no answer is received, it implies that the WS has failed.

6.4 Description of THROWS Architecture Functionality

The WS-SAGAS transaction model supports the specification of a process in different ways following different defined aggregation patterns. For these different patterns, we need to describe how they are to be executed. In the following, we show how the *state* and *vitality degree* concepts play a crucial role in determining how the execution is to progress, and especially, in distinguishing successful from faulty situations, in informing of failures, and in recovering from failures. In describing the functioning of THROWS architecture, we mostly describe sequential and parallel WS-SAGAS. Other patterns can be deduced in the same way, because we have defined the execution semantics of each pattern in detail in the previous section. (Figure 6.2) shows the different transition rules that THROWS defines to make an `atomic compensatable` element's state change from one state to another. The diagram is for `vital` and `nonvital` elements.

6.4.1 Initiating the Execution of a Process P_i

The entire process P_i execution is initiated by an engine coordinator ec_i^p. To ensure that it does not constitute a single point of failure, we assume that we have $REL(P_i)$, a replica engine coordinator list that contains several replicas of ec_i^p.

A process running on a server side is responsible for taking a user request for a particular process and for creating $REL(P_i)$, and starting the execution with one of the replica engine coordinators; we assume here that it is ec_i^p. More precisely, below is the content of $REL(P_i)$:

$$REL(P_i) = \left\{ DR(ec_i^1), \cdots, DR(ec_i^p), \cdots, DR(ec_i^q), \cdots \right\} \quad (6.3)$$

Every replica in $REL(P_i)$ contains the *CEP* of the overall process and the *CEP* of the WS-SAGAS for which the engine coordinator is responsible. That is, in ec_i^p we have both, $CEP(P_i, ec_i^p)$ and $CEP(WS\text{-}SAGAS_i, ec_i^p)$; $CEP(P_i)$ is formed as follows:

$$\begin{aligned} CEP(P_i, ec_i^p) : & CEP(WS\text{-}SAGAS_i, ee_{i.1}^p) \\ & \bigcup CEP(WS\text{-}SAGAS_{i.a}, ee_{i.a.1}^p) \\ & \vdots \\ & \bigcup CEP(WS\text{-}SAGAS_{i.a.*.b}, ee_{i.a.*.b.1}^p) \\ & \vdots \\ & \bigcup CEP(WS\text{-}SAGAS_{i.a.*.b.*.c}, ee_{i.a.*.b.*.c.1}^p) \ . \end{aligned}$$

$CEP(P_i, ec_i^p)$ means that the *CEP* copy is stored on engine ec_i^p. Similarly, $CEP(WS\text{-}SAGAS_{i.a}, ee_{i.a.1}^p)$ is stored on engine $ee_{i.a.1}^p$, $CEP(WS\text{-}SAGAS_{i.a.*.b}, ee_{i.a.*.b.1}^p)$ is stored on engine $ee_{i.a.*.b.1}^p$, etc (see Figure 6.1).

The engine coordinator initiates the process execution by starting the execution of the WS-SAGAS at the first nesting level, that is, $WS\text{-}SAGAS_i$. The first step is to generate the *CEL* of the element appearing first in the first pattern; in the case of $WS\text{-}SAGAS_i$, $CEL(E_{i.l})$ is generated, an engine is allocated, the predelegation phase is performed, and a copy of $CEP(WS\text{-}SAGAS_i, ec_i^p)$ is passed to $ee_{i.1}^p$ and the execution of the WS-SAGAS starts.

We describe below the internal functioning of a WS-SAGAS: the execution of patterns that only contain `atomic` elements. Then, we describe how the execution is delegated between different nesting levels.

6.4.2 Sequence Pattern Execution ($[E_{i.k}]; [E_{i.k+1}]$)

We describe the general execution model of a sequence pattern aggregating only `compensatable atomic` elements. We assume that an engine executor $ee_{i.k}^p$ has been allocated to the element $E_{i.k}$. Depending on the execution progress, different scenarios may occur:

1. *Element $E_{i.k}$ commitment*: In this scenario we assume that the execution of $ws_{i.k}^p$ controlled by $ee_{i.k}^p$ was successful. The successful execution must be reflected on the *CEP* content (to comply with our proposed notation of an engine that uses p to designate any engine and q to designate an engine that was successful, we use $ee_{i.k}^q$ instead of $ee_{i.k}^p$). Therefore, the engine $ee_{i.k}^q$ has to update the locally stored copy of *CEP*; specifically, the state of the element is to be modified as follows:

   $$CEP(WS\text{-}SAGAS_i, ee_{i.k}^q).\overrightarrow{DR}(E_{i.k}).state := \texttt{Committed} \ .$$

   The execution results are used to update the output parameters of the element:

   $$CEP(WS\text{-}SAGAS_i, ee_{i.k}^q).\overrightarrow{DR}(E_{i.k}).operation.out := DR(ee_{i.k}^q).operation.out \ .$$

   Afterwards, the engine executor has to delegate execution control. To this end, it generates $CEL(E_{i.k+1})$ and chooses an engine executor to which it must delegate the execution control; we assume here that the engine $ee_{i.k+1}^p$ was allocated to execute $E_{i.k+1}$. We note that the execution of $E_{i.k+1}$ by $ee_{i.k+1}^p$ is the execution of a WS $ws_{i.k+1}^p$ controlled by $ee_{i.k+1}^p$. The WS $ws_{i.k+1}^p$ to be mapped to $E_{i.k+1}$ must satisfy the following three conditions:

   $$CEP(WS\text{-}SAGAS_i, ee_{i.k}^q).\overrightarrow{DR}(E_{i.k+1}).description = CEL(E_{i.k+1}).DR(ee_{i.k+1}^p).description \ .$$

   (*the functionalities of the element meet the WS capabilities*)

   $$CEP(WS\text{-}SAGAS_i, ee_{i.k}^q).\overrightarrow{DR}(E_{i.k+1}).operation.in = CEL(E_{i.k+1}).DR(ee_{i.k+1}^p).operation.in \ .$$

   (*the input parameters of the element and of the WS are compliant*)

   $$CEP(WS\text{-}SAGAS_i, ee_{i.k}^q).\overrightarrow{DR}(E_{i.k+1}).operation.out = CEL(E_{i.k+1}).DR(ee_{i.k+1}^p).operation.out \ .$$

   (*the output parameters of the element and of the WS are compliant*)

Once these conditions are satisfied, the two engines start a *predelegation phase* in which one engine agrees to delegating control while the other agrees to having control delegated to it. When $ee^q_{i.k}$ receives the agreement notification from $ee^p_{i.k+1}$, it finalizes the delegation by updating and sending the *CEP* content to the allocated engine:

$$CEP(WS\text{-}SAGAS_i, ee^q_{i.k}).\overrightarrow{DR}(E_{i.k+1}).engine := ee^p_{i.k+1} \quad .$$

We emphasize that whether or not $E_{i.k}$ was `vital` does not affect the functioning of the engine for a successful execution.

2. *Vital Element $E^v_{i.k+1}$ Failure*: We assume that exceptional behavior (e.g., unavailable, timed out, error message) of the WS $ws^p_{i.k+1}$ precluded the engine $ee^p_{i.k+1}$ from successfully committing the element $E^v_{i.k+1}$. Consequently, the element $E^v_{i.k+1}$ execution attempt by the engine $ee^p_{i.k+1}$ is considered to have failed and the following steps must be performed to recover to a consistent state, because the element's success is crucial for the overall subtransaction success. First, the engine $ee^p_{i.k+1}$ has to update its locally stored copy of *CEP* with the latest progress in execution as follows:

$$CEP(WS\text{-}SAGAS_i, ee^p_{i.k+1}).\overrightarrow{DR}(E^v_{i.k+1}).state := \texttt{Failed} \quad .$$

Second, a copy of the locally stored *CEP* is sent as a *failure notification* message to the engine responsible for the execution of the predecessor of $E^v_{i.k+1}$, that is, to engine $ee^q_{i.k}$, so that it takes charge of the failure recovery process. Depending on the cardinality of $CEL(E^v_{i.k+1})$ either a *forward recovery* (i.e., trying to advance the execution process with an attempt at an execution retrial) or a *backward recovery* (i.e., to recover the CWS to a consistent state) is to be performed by $ee^q_{i.k}$.

3. *Vital Element $E^v_{i.k+1}$ Forward Recovery*: After being notified of $E^v_{i.k+1}$ execution failure by $ee^p_{i.k+1}$ (i.e., receiving a copy of *CEP* in which the element $E^v_{i.k+1}$ state was set to `Failed`), the engine $ee^q_{i.k}$ checks the cardinality of $CEL(E^v_{i.k+1})$ to determine whether trying a forward recovery, by allocating another engine in order to reattempt the execution of $E^v_{i.k+1}$, is feasible or not.

When $(|CEL(E^v_{i.k+1})| \neq 0)$ is verified, it means that other WS satisfying the element $E^v_{i.k+1}$ required functionalities are available. Therefore, execution retrial is possible: the engine $ee^q_{i.k}$ searches for the engine ranked next to $ee^p_{i.k+1}$ in $CEL(E^v_{i.k+1})$; here we assume it is the engine $ee^{p+1}_{i.k+1}$. This engine is allocated to reattempt the execution of the element $E^v_{i.k+1}$; the locally stored content of *CEP* in $ee^q_{i.k}$ is updated as follows:

$$CEP(P_i, ee^q_{i.k}).\overrightarrow{DR}(E_{i.k+1}).engine := ee^{p+1}_{i.k+1} \quad .$$

Subsequently, a *predelegation* has again been performed with the new engine $ee^{p+1}_{i.k+1}$, the *CEP* content, with the required context data for effectively starting the execution, is communicated.

4. *Vital Element $E^v_{i.k+1}$ Backward Recovery*: If it happens that $(|CEL(E^v_{i.k+1})| = 0)$, the execution retrial is impossible and a backward recovery is necessary. To this end, all the other elements from the same WS-SAGAS that are ordered before $E^v_{i.k+1}$ and have already committed are compensated for, that is, all the `vital` and `nonvital` elements that verify the following conditions:

$$CEP(WS\text{-}SAGAS_i, ee^q_{i.k}).\overrightarrow{DR}(E_{i.\ell|\ell\in[1..k]}).state = \texttt{Committed} \quad ,$$

are compensated for by executing for each element its compensating element. We emphasize that a compensation mechanism is triggered by a *compensation request* propagated for all the engines of the elements that verify the above condition. In addition, we note that the *compensation request* propagation is handled as described below.

5. *Element $E_{i.k}$ Compensation*: On receiving a *compensation request*, an engine first checks if it has any predecessor to which it must, similarly, propagate a *compensation request*. Second, it has to compensate the execution of the element for which it is responsible.

That is, if we consider the case of the engine $ee^q_{i.k}$, responsible for the element $E_{i.k}$ (applicable to the case of `vital` and `nonvital`), the engine $ee^q_{i.k}$ searches for a WS that satisfies the functionalities of $E'_{i.k}$, the *compensating element* of $E_{i.k}$.

Here, we assume that a WS $ws^{q'}_{i.k}$ was discovered. The engine $ee^q_{i.k}$ executes the WS $ws^{q'}_{i.k}$. On completing $ws^{q'}_{i.k}$ execution, $ee^q_{i.k}$ updates *CEP* and propagates it with the *compensation request* to the engines concerned:

$$CEP(WS\text{-}SAGAS_i, ee^q_{i.k}).\overrightarrow{DR}(E_{i.k}).state := \texttt{Compensated} \quad .$$

The engine that finds that it has no predecessor engine is presumably the engine responsible for the first element $E_{i.1}$. This engine executor has only to compensate the element $E_{i.1}$. Finally, the overall WS-SAGAS of this level failure is calculated. The engine executor $ee^q_{i.p}$, responsible for the element $E_{i.1}$, after compensating the element $E_{i.1}$ must propagate the failure information up the hierarchy to perform a backward recovery in other terminated WS-SAGAS. To this end, it sends its last updated copy of $CEP(WS\text{-}SAGAS_i, ee^q_{i.1})$ to the engine coordinator controlling the element parent of the $WS\text{-}SAGAS_i$. In this case, it propagates the copy to ec^q_i. We describe below in this section how an engine coordinator handles this failure information.

6. *Nonvital Element $E^{\bar{v}}_{i.k+1}$ Failure*: Assume that the direct successor of $E^{\bar{v}}_{i.k}$ is the `nonvital` element $E^{\bar{v}}_{i.k+1}$. In this case, even if the engine $ee^p_{i.k+1}$ allocated to $E^{\bar{v}}_{i.k}$ fails, it is not necessary to attempt a recovery because its success is optional. However, the engine $ee^p_{i.k+1}$ continues the execution as if the element was committed by updating the *CEP* and performing the control *predelegation phase* as usual.

6.4.3 Parallel Pattern Execution $(E_{i.k};(E_{i.k+1}||\ldots||E_{i.k+j}))$

The main difference with the execution of a sequence of WS-SAGAS is that the engine allocated to the element $E_{i.k}$—we assume here it is engine $ee^q_{i.k}$—on finishing executing $E_{i.k}$ has to generate the *CEL* of the set of elements to be executed in parallel, that is, the elements from $E_{i.k+1}$ to $E_{i.k+j}$, as noted above $\mathscr{S}(E_{i.k})_{\text{succ}}$. Similarly, $ee^q_{i.k}$ generates *CEL* for all the elements in $\mathscr{S}(E_{i.k})_{\text{succ}}$. If we assume that the *CEL* generation and the engine allocation steps were done, we have the *CEP* updated as follows:

$$CEP(WS\text{-}SAGAS_i, ee^q_{i.k}).\overrightarrow{DR}(E_{i.k+1}).engine := ee^p_{i.k+1}$$
$$CEP(WS\text{-}SAGAS_i, ee^q_{i.k}).\overrightarrow{DR}(E_{i.k+2}).engine := ee^p_{i.k+2}$$
$$\vdots$$
$$CEP(WS\text{-}SAGAS_i, ee^q_{i.k}).\overrightarrow{DR}(E_{i.k+j}).engine := ee^p_{i.k+j} \ .$$

Because starting the execution of the elements pertaining to $\mathscr{S}(E_{i.k})_{\text{succ}}$ must be synchronized, a *synchronization phase* has to be introduced.

The synchronization phase comes directly after the *predelegation phase* finishes. It aims at ensuring that the engine $ee^q_{i.k}$, after receiving the delegation agreements from the other, different engines, simultaneously propagates a start signal. In addition, because a significant delay in receiving the delegation agreement messages can seriously compromise the CWS execution, we use *the peer-engine waiting period*, as already specified. After it has elapsed without receiving the acknowledgment message from one of the engines, $ee^q_{i.k}$ must select another candidate engine. When the elements in $\mathscr{S}(E_{i.k})_{\text{succ}}$ are being executed, three situations are most likely to occur:

1. *Simultaneous Commitment of all the Vital Elements:* All the `vital` elements in $\mathscr{S}(E_{i.k})_{\text{succ}}$ have been successfully committed, that is, they verify the following condition:

$$\begin{cases} \forall E^v_{i.\ell} \mid E^v_{i.\ell} \in \mathscr{S}(E_{i.k})_{\text{succ}} \text{ and } \ell \in [k+1..k+j]: \\ CEP(WS\text{-}SAGAS_i, ee^1_{i.\ell}).\overrightarrow{DR}(E_{i.\ell}).state = \texttt{Committed} \ . \end{cases}$$

   If the above condition is verified, first, all the engines controlling the elements verifying the above condition have to exchange copies of their locally stored *CEP* contents. At the end of this update, all the engines controlling `vital` elements in $\mathscr{S}(E_{i.k})_{\text{succ}}$ have the same copy of *CEP*.
   Second, depending on the pattern used to make the different elements in $\mathscr{S}(E_{i.k})_{\text{succ}}$ converge to a single point, a set of elements from $\mathscr{S}(E_{i.k})_{\text{succ}}$ has to generate the $CEL(s)$ for the element(s) that comes directly after them. If we assume that $E_{i.l}$ is the common successor, then the elements in $\mathscr{S}(E_{i.k})_{\text{succ}}$ have to cooperate in generating the $CEL(E_{i.l})$ by taking the union of the different $CEL(s)$ generated by each of their engines.
   Finally, we note that only the results of the execution of the `vital` elements were considered because the other `nonvital` elements' success or failure does not affect the others' progress.

2. *Failure of one or more* `vital` *elements:* One or more `vital` elements in $\mathscr{S}(E_{i.k})_{\text{succ}}$ verify the following condition:

$$\begin{cases} \exists E^v_{i.\ell} \mid E^v_{i.\ell} \in \mathscr{S}(E_{i.k})_{\text{succ}} \text{ and } \ell \in [k+1..k+j]: \\ CEP(WS\text{-}SAGAS_i, ee^p_{i.\ell}).\overrightarrow{DR}(E_{i.\ell}).state = \texttt{Failed} \ . \end{cases}$$

   If a similar situation occurs, then the failed engine(s) has to inform its direct predecessor, here $ee^q_{i.k}$, of the failure(s). As described above, $ee^q_{i.k}$ tries to perform a failure recovery. The main differences reside in that, first, the failed engine(s) also have to inform the concurrent engines controlling the other elements to avoid compromising the subtransaction execution by making the others wait forever.
   Second, if a backward recovery is necessary, it is most likely that the execution of one or several elements from $\mathscr{S}(E_{i.k})_{\text{succ}}$ is still in progress. As a result, the compensation mechanism described above is not applicable.
   It is essential to abort all the elements that verify the following condition by stopping their execution:

$$\begin{cases} \exists E^v_{i.\ell} \mid E^v_{i.\ell} \in \mathscr{S}(E_{i.k})_{\text{succ}} \text{ and } \ell \in [k+1..k+j]: \\ CEP(WS\text{-}SAGAS_i, ee^1_{i.\ell}).\overrightarrow{DR}(E_{i.\ell}).state = \texttt{Executing} \ . \end{cases}$$

   To abort all the elements that verify the above condition, $ee^q_{i.k}$ has to propagate an *abort request* to all their allocated engines. Each engine that receives an *abort request* responds by immediately stopping its execution and updating its locally stored *CEP* copy, then sends it to $ee^q_{i.k}$. After all the different engines have properly handled the received *abort request*, the *CEP* content stored on $ee^q_{i.k}$ side is:

$$\begin{cases} \exists E^v_{i.\ell} \mid E^v_{i.\ell} \in \mathscr{S}(E_{i.k})_{\text{succ}} \text{ and } \ell \in [k+1..k+j]: \\ CEP(WS\text{-}SAGAS_i, ee^q_{i.k}).\overrightarrow{DR}(E_{i.\ell}).state = \texttt{Aborted} \ . \end{cases}$$

3. *One or more* `vital` *elements are still executing while the others have already committed:* At least one `vital` element in $\mathscr{S}(E_{i.k})_{\text{succ}}$ verifies the following condition:

$$\begin{cases} \forall E^v_{i.\ell} \mid E^v_{i.\ell} \in \mathscr{S}(E_{i.k})_{\text{succ}} \text{ and } \ell \in [k+1..k+j]: \\ CEP(WS\text{-}SAGAS_i, ee^q_{i.\ell}).\overrightarrow{DR}(E_{i.\ell}).state = \texttt{Executing} \ . \end{cases}$$

   In this case, two scenarios are possible:
   - *Scenario 1:* The engine(s) responsible for the committed element(s) informs, as described above, the concurrent engine(s) of their commitment(s) and generates the *CEL* of the successor(s), if they exist. However, they wait for the termination of the element(s) that is/are still being executed to agree on the *CEL* and to choose the engine to allocate to the successor(s), so that a *predelegation phase* is performed to effectively start executing the successor(s).
   - *Scenario 2:* The committed elements' engines inform the concurrent engines of their commitment and enter a latent state while waiting for the remaining unfinished elements' commitment.

A comparison of the two scenarios shows that in (*Scenario 1*) the time spent by the concurrent engines in generating the $CEL(s)$ of the successor(s) etc can be meaningless if one of the still executing engines fails. On the other hand, if all the remaining elements were committed, then the time spent in a latent state by the different engines in (*Scenario 1*) is an overhead.

6.4.4 Execution Control Delegation between an Engine Executor and an Engine Coordinator

Assume that $E_{i.a}$ is a composite element from $WS\text{-}SAGAS_i$. If the execution progress of $WS\text{-}SAGAS_i$ reaches the element $E_{i.a}$, that is, the element $E_{i.a-1}$ terminates its execution then, $ee^q_{i.a-1}$ has to delegate the execution control as described above.

The particularity here is that when $ee^q_{i.a-1}$ checks for a predecessor, the *CEP* content reveals that the next element to be executed is $E_{i.a}$ and it is `composite`. Consequently, the engine executor $ee^q_{i.a-1}$ generates an *REL* not a *CEL*, the generated *REL* being $REL(E_{i.a})$. Then, an engine coordinator is selected; let this engine be $ec^p_{i.a}$. The execution of $WS\text{-}SAGAS_i$ is suspended waiting for the element $E_{i.a}$ to be executed by $ec^p_{i.a}$. As described at the beginning of this section, $ec^p_{i.a}$ starts execution of $WS\text{-}SAGAS_{i.a}$ in exactly the same way as the engine coordinator ec^p_i starts the execution of $WS\text{-}SAGAS_i$.

6.4.5 Execution Termination of a WS-SAGAS Transaction

Assume that the execution of $WS\text{-}SAGAS_{i.a}$ was terminated and the execution reached the last element $E_{i.a.n_{i,2}}$, which was executed by the engine executor $ee^q_{i.a.n_{i,2}}$. $ee^q_{i.a.n_{i,2}}$ checks the locally stored *CEP*, $CEP(WS\text{-}SAGAS_{i.a}, ee^q_{i.a.n_{i,2}})$, for elements not yet executed. The result is that all the elements in $WS\text{-}SAGAS_{i.a}$ have finished already.

Therefore, it has to inform the engine coordinator responsible for the parent element of $WS\text{-}SAGAS_{i.a}$ that the execution was terminated. To this end, it sends a copy of $CEP(WS\text{-}SAGAS_{i.a}, ee^q_{i.a.n_{i,2}})$ to the engine $ec^p_{i.a}$.

The engine $ec^p_{i.a}$ deduces the element $E_{i.a}$'s state from the received $CEP(WS\text{-}SAGAS_{i.a}, ee^q_{i.a.n_{i,2}})$. More precisely, assume we deduced from $CEP(WS\text{-}SAGAS_{i.a}, ee^q_{i.a.n_{i,2}}).state$ that:
$DR(WS\text{-}SAGAS_{i.a}).state = \texttt{committed}$, then $ec^p_{i.a}$ updates the state of the element $E_{i.a}$ in $CEP(WS\text{-}SAGAS_i, ec^p_{i.a})$ as follows:

$$CEP(WS\text{-}SAGAS_i, ec^p_{i.a}).DR(E_{i.a}).state := \texttt{committed}$$

Then, the execution is continued as described above.

6.4.6 Execution Termination of a Process

On every termination of a WS-SAGAS, the last engine responsible for the last element in the subtransaction finalizes the execution in the way described above. Every WS-SAGAS termination is cascaded up in the hierarchy until the uppermost level controlled by the engine coordinator ec^p_i is reached. Then, this engine finalizes the overall process execution by requesting from every other engine coordinator the last updated copy of the *CEP* of the subtransaction for which it was responsible.

When all the copies have been collected, the locally stored copy of the overall process *CEP* is updated and process termination is deduced.

6.4.7 Interruption of the Execution of a WS-SAGAS with a Failure

Assume that the execution of $WS\text{-}SAGAS_{i.a}$ was interrupted by a failure of one of its constituent `vital` elements and that the forward recovery mechanism was unable to overcome the failure. As a last resort, a backward recovery is performed as described in the execution of a sequence/parallel pattern.

When the failure message (i.e., $CEP(WS\text{-}SAGAS_{i.a})$, with the state of one element set to `failed`), reaches the engine responsible for the first element in $WS\text{-}SAGAS_{i.a}$—in this case this engine is $ee^q_{i.a.1}$—this element performs a backward recovery for the element it controls and propagates the failure information up in the hierarchy. All it has to do is to send the updated copy of $CEP(WS\text{-}SAGAS_{i.a}, ee^q_{i.a.1})$ to the parent of the engine coordinator controlling the parent of $WS\text{-}SAGAS_{i.a}$. In this case, a copy reaches $ec^p_{i.a}$ and, because it checks the *CEP* content, a failure is deduced. Subsequently, a backward recovery is triggered: $ec^p_{i.a}$ initiates a backward recovery in $WS\text{-}SAGAS_i$ by changing the state of the element $E_{i.a}$ as follows:

$$CEP(WS\text{-}SAGAS_i, ec^p_{i.a}).DR(E_{i.a}).state := \texttt{failed}.$$

The failure of $E_{i.a}$ is handled in the same way as described above by compensating all the committed elements in the transaction $WS\text{-}SAGAS_i$ and aborting the other elements still executing. When the failure message reaches the uppermost level, that is, the engine ec^p_i, this engine propagates the failure down the hierarchy to all the WS-SAGAS that were terminated before the $WS\text{-}SAGAS_{i.a}$ failure (i.e., all the WS-SAGAS lower in the hierarchy must also be recovered).

7 Composite Web Services QoS Modeling and Analysis

We describe the third part of the FENECIA approach: assessing the QoS of CWS depicted as WS-SAGAS and executed following THROWS architecture. Our chief aims in defining a QoS model are that, first, it allows verification of the described CWS as WS-SAGAS are reliably serving their purpose, when executed, by achieving a high level of dependability. Second, it allows greater improvement of the quality of execution in the future by favoring the more reliable WS and discarding the WS that are most likely to be the stage failures.

Moreover, we aim to use the QoS estimation and analysis as a basis for improving the WS-SAGAS structure. To this end, our model characterizes, estimates, and analyzes several QoS properties, namely the *execution time* and the *reliability* [17,21], on the basis of the past executions collected in a history, and takes into consideration the failure repercussions.

7.1 Preliminaries

We give an overview of the QoS concepts we are concerned with because the QoS concept in itself is broad. It has been applied to many areas and, depending on the area of application, its definition varies.

Some define it as *"a set of user-perceivable attributes, which describe a service and the way it is perceived"* [50, 51]. We are not concerned with this form of QoS because it has been widely addressed and was the subject of considerable research efforts in the area of WSC. Several studies have focused on the dynamic selection of the provider [52–54] and on semantic WS description to improve the selection [55]. These studies are classified under the umbrella of maximizing user satisfaction.

A more appropriate definition of the QoS we treat in the FENECIA approach is *"the system property that consists of a set of quality requirements on the collective behavior of one or more objects, such as the information transfer rate, the latency, the system failure probability, etc."* [50,56]. That is, this category of QoS assessment chiefly targets estimating a number of QoS properties for later analysis by the system designers to verify to what extent the CWS are efficiently serving their purpose during execution (i.e., are the introduced fault-tolerance mechanisms working properly? Are the selected WS adequate?).

7.2 Motivation

The issues guiding us toward introducing such a model are summarized below.

- Most of the proposed approaches that address the estimation of the QoS issue, first, make use of either mathematical modeling or simulation tools [57–60]. Second, they typically provide a global view of the range of variation of the estimates of certain properties of the CWS as a whole, or their estimates are only applicable for static CWS, which make them inapplicable for both THROWS and WS-SAGAS for dynamic composition. However, providing more detailed estimations, especially in the case of complex CWS, is required more and more. To fulfill this requirement, our QoS model for CWS is oriented toward acquiring more practical and detailed estimates of the QoS of each element, and derives equivalent estimates for the overall CWS;
- Most of the current approaches dealing with QoS estimates in the WS context rely on the QoS information advertised by the WS owners/providers, which may be not up to date or subject to manipulation by the providers. To overcome this limitation, we compute the QoS estimations on the basis of the CWS execution *observation*, where the observation results are collected in a history. In doing so, we believe that more accurate estimates can be acquired because we do not rely on the providers' data.
- A major part of the work done up until now considers only situations where the CWS do not fail. As a result, the estimates obtained are very often regarded as too optimistic because they do not account for any failure (information, recovery) and their repercussions. In our model we account for failures and their repercussions on the effective performances of the CWS because this is particularly required in the WS architecture, in view of the WS inherent tendency to fail relatively easily (relative to other computing components). Typical causes of failure include: noncompliant WS characteristics (e.g., transactional supports, management policies, access rights) and obvious Internet limitations (e.g., latency, time-out, security).
- Finally, because WS are generally stateless, tracking the failures and determining their locations is almost impossible. To overcome this limitation, the notion of *state* that we initially introduced in WS-SAGAS is used. Introducing the *state* concept is expected to contribute in acquiring more accurate information on the location of failures and to be used later to improve the CWS QoS.

7.3 Execution Time Characterization

In our QoS model, we first estimate the execution time of each `atomic` element $E_{i.k}$ for a subtransaction $WS\text{-}SAGAS_i$ from a hierarchy of nested WS-SAGAS forming a process P_i. Then, we describe the derivation of the equivalent estimate for the entire $WS\text{-}SAGAS_i$ and the entire process. Our QoS model builds heavily on the observation of the past invocations of the process and on collecting these observations in a history. Because the execution follows the THROWS architecture, the history content is chiefly formed from the different copies of *CEP* stored in the different engines' logs that cooperated to execute the whole process. By enforcing the policy, all the copies of the different *CEPs* stored locally on the different engines must be kept until the end of each process invocation; at the end we have information about the process life cycle and of all its constituent elements. Moreover, applying the same policy to all the different *CELs* can be very interesting as each *CEL*, in itself, is a history of the engine-ws couples attempted. With both the *CEP* and the *CEL* contents, tracking failures' locations and determinations of the engine-ws couples that fail readily can, in the future, significantly improve the quality of execution.

7.3.1 The Execution Time of an `Atomic` Element

Because each `atomic` element is mapped dynamically to a WS, we investigate first the issue of estimating the execution time of an elementary WS, which has been addressed previously on several occasions. Specifically, we refer to [60,57] in which the authors defined the execution time taken by a single WS invocation with the sum of the three following constituents:

- The `service time S(WS)` is the time that the WS takes to perform its task.
- The `message delay time M(WS)` is determined by the size of the message being transmitted/returned and the load on the network through which the message is sent.
- The `waiting time W(WS)` is the delay caused by the load on the system where the WS is deployed.

This model does not comply with our approach because we target a dynamic and fault-tolerant execution. However, the above model is, first, only for CWS with one-to-one static WS-element mapping. Second, it does not take into account any eventual failure and how it may intervene in varying the performances. These two reasons preclude it from being directly applicable in the FENECIA approach, without further extensions.

In characterizing the execution time, we build on the above model and introduce the *Optimistic Execution Time* and *Probable Execution Time* where the former is limited to the correct execution situations and where the latter considers all the possible execution situations (i.e., committed execution, failed execution, compensated execution, aborted execution, etc.) of a fault-tolerant CWS. Distinguishing between these two variants provides more accurate estimates to account for the failure repercussions on the delivered performances.

Definition 7.1 (The Optimistic Execution Time(opt)) We define the *Optimistic Execution Time* (denoted $T(E^v_{i.k})_{opt}$), as the time spent by the *dynamically mapped* engine executor-WS couple in executing the `vital` element $E^v_{i.k}$. This definition considers only the best case where the execution is committed when $E^v_{i.k}$ is mapped to the first-ranked couple engine executor-WS in the corresponding *CEL*.

We note here that any `atomic` element $E^v_{i.k}$ can be mapped at runtime to more than one engine executor-WS, at most exactly $|CEL(E^v_{i.k})|$, the cardinality of $CEL(E^v_{i.k})$, with $[1..|CEL(E^v_{i.k})|]$ ranging over p, and every time the element $E^v_{i.k}$ is to be executed it is allocated an engine executor $ee^p_{i.k}$ (controlling the WS $ws^p_{i.k}$). If we assume that for a particular value of p we note q, $E^v_{i.k}$ executions by $ee^q_{i.k}$ committed successfully, then $E^v_{i.k}$ was attempted by q engines from $CEL(E^v_{i.k})$ (q verifies: $q \leq |CEL(E^v_{i.k}|)$; in all these execution attempts, $(q-1)$ executions were finished by failures. That is, we can say that $E^v_{i.k}$ was retried *q times*, and that the qth execution delegated to the engine $ee^q_{i.k}$ (controlling the WS $ws^q_{i.k}$) was successful.

We define $T(E^v_{i.k})_{opt}$ by the following equation(7.1) where $T(E^v_{i.k})_{opt}$ is the sum of the *execution time* (exactly $S(ws^q_{i.k})$)—the time taken by the WS to process its sequence of activities—and of the latency (exactly $L(ee^q_{i.k}, ws^q_{i.k})$)—the time necessary to send a request and receive a response:

$$\begin{aligned} T(E^v_{i.k})_{opt} &= T(E^v_{i.k}, ee^q_{i.k}, ws^q_{i.k}) \\ &= S(ws^q_{i.k}) + L(ee^q_{i.k}, ws^q_{i.k}) \\ &\quad \text{with: } 1 \leq q \leq |CEL(E^v_{i.k})| \end{aligned} \tag{7.1}$$

In the special case of a `nonvital` element, the execution is attempted only once; consequently the equation (7.1) is transformed as follows:

$$\begin{aligned} T(E^{\bar{v}}_{i.k})_{opt} &= T(E^{\bar{v}}_{i.k}, ee^1_{i.k}, ws^1_{i.k}) \\ &= S(ws^1_{i.k}) + L(ee^1_{i.k}, ws^1_{i.k}) \\ &\quad \text{with: } |CEL(E^{\bar{v}}_{i.k})| = 1 \end{aligned} \tag{7.2}$$

Definition 7.2 (The Probable Execution Time (prob)) We define the *Probable Execution Time*(denoted $T(E^v_{i.k})_{prob}$) as the estimate of the time spent by an `atomic` element $E^v_{i.k}$ in being executed effectively, which is equal to $T(E^v_{i.k})_{opt}$, to which we add the time necessary for recovering from failures that the same instance of the WS-SAGAS as a whole has encountered (see Equation (7.3)).

$$T(E^v_{i.k})_{prob} = T(E^v_{i.k})_{opt} + RP(E^v_{i.k}) + R(E^v_{i.k}), \tag{7.3}$$

where:

- $T(E^v_{i.k})_{opt}$ is the time to execute the WS $ws^p_{i.k}$ controlled by the engine executor $ee^p_{i.k}$. We note that p ranges over $[1..|CEL(E^v_{i.k})|]$ and that, for a particular value q of p, the execution of $E^v_{i.k}$ was committed. Where $E^v_{i.k}$ was retried with all the engines in $CEL(E^v_{i.k})$ and the execution failed for all of them, then $T(E^v_{i.k})_{opt}$ is assumed to be equal to 0 and $T(E^v_{i.k})_{prob}$ is equal to the time spent in performing a forward recovery by retrying $E^v_{i.k}$ several times (exactly $|CEL(E^v_{i.k})|$ times).
- $RP(E^v_{i.k})$ is the time spent by $E^v_{i.k}$ in informing of its own failure or in being informed about others' failure. $PR(E^v_{i.k})$ is detailed more in the following definition.
- $R(E^v_{i.k})$ is the total period of time spent in performing a forward recovery every time the element $E^v_{i.k}$ failed, to which we add the time spent by $E^v_{i.k}$ in performing a backward recovery, if it happens that any of the elements executed in parallel with it, or the elements that come directly after it fail. $R(E^v_{i.k})$ is considered in greater detail below.

Definition 7.3 (The Failure Recovery Preparation Time) We define the *failure recovery preparation time* of an `atomic` element ($RP(E^v_{i.k})$) as the time necessary to notify of a failure, or to send a recovery *(abort/compensation) request*. All the messages are one-way SOAP messages that contain the last updated copy of *CEP*. Depending on the failure location (i.e., the element itself or another element from the same process) and on the elements *state* and *vitality degree*, the defined expression of the *failure recovery preparation time* varies as follows:

- In the first case, $E^v_{i.k}$ was committed by an engine executor $ee^q_{i.k}$ after being reattempted $q-1$ times; on every failure by an allocated engine $ee^p_{i.k}$, it has to inform its direct predecessor by sending a failure information message to $e^q_{i.k-1}$ (i.e., it can be an engine executor or coordinator), thereby the notation $I(ee^p_{i.k}, e^q_{i.k-1})$. If the element is `nonvital`, performing a forward recovery is not required:

$$\text{Case 1.}: CEP(WS\text{-}SAGAS_i, ee^q_{i.k}).\overrightarrow{DR}(E_{i.k}).state = \texttt{Committed}$$

(element committed / WS-SAGAS committed)

$$RP(E^v_{i.k}) = \sum_{p=1}^{q-1} I(ee^p_{i.k}, e^q_{i.k-1}) \text{ (vital)}$$

$$RP(E^{\bar{v}}_{i.k}) = 0 \text{ (nonvital)}$$

- In the second case, $E^v_{i.k}$ was committed by an engine executor $ee^q_{i.k}$ and was reattempted $q-1$ times; however, the overall WS-SAGAS failed because of the failure of another element that was executed later in the process. If we assume the failed element to be $E^v_{i.j}$, with $(j > k)$ (handled in the same way whether it is composite or atomic), then the engine allocated to $E^v_{i.k}$ receives a *compensation request* from the engine responsible for $E^v_{i.j}$ ($CR(e^p_{i.j}, ee^q_{i.k})$ is the time spent in exchanging such a message):

$$\text{Case 2.}: CEP(WS\text{-}SAGAS_i, ee^q_{i.k}).\overrightarrow{DR}(E_{i.k}).state = \texttt{Compensated}$$

(element committed/WS-SAGAS failed)

$$RP(E^v_{i.k}) = \sum_{p=1}^{q-1} I(ee^p_{i.k}, e^q_{i.k-1}) + CR(e^p_{i.j}, ee^q_{i.k}) \text{ (vital)}$$

$$RP(E^{\bar{v}}_{i.k}) = CR(e^p_{i.j}, ee^1_{i.k}) \text{ (nonvital)}$$

- In the third case, while the element $E^v_{i.k}$ is being executed by an engine executor $ee^r_{i.k}$ (i.e., it was reattempted $r-1$ times), the overall process failed because of the failure of another element that was executed later in this WS-SAGAS. If we assume the failed element to be $E^v_{i.j}$, with $j > k$, the engine allocated to $E^v_{i.k}$ receives an *abort request* from the engine responsible for $E^v_{i.j}$ (the time spent in exchanging such a message is denoted $AR(e^p_{i.j}, ee^r_{i.k})$):

$$\text{Case 3.}: CEP(WS\text{-}SAGAS_i, ee^r_{i.k}).\overrightarrow{DR}(E_{i.k}).state = \texttt{Aborted}$$

(element still executing with $ee^r_{i.k}$ /process failed)

$$RP(E^v_{i.k}) = \sum_{p=1}^{r-1} I(ee^p_{i.k}, e^q_{i.k-1}) + AR(e^p_{i.j}, ee^r_{i.k}) \text{ (vital)}$$

$$RP(E^{\bar{v}}_{i.k}) = AR(e^p_{i.j}, ee^r_{i.k}) \text{ (nonvital)}$$

- In the fourth case, $E^v_{i.k}$ was attempted by all the engines in its CEL (i.e., it was reattempted $CEL(E^v_{i.k})$ times) but, unfortunately, it failed in all the retried times; therefore, the overall WS-SAGAS failure is deduced.
  This case is not applicable to a `nonvital` element because its failure does not entail overall process failure:

$$\text{Case 4.}: CEP(WS\text{-}SAGAS_i, ee^{|CEL(E^v_{i.k})|}_{i.k}).\overrightarrow{DR}(E^v_{i.k}).state = \texttt{Failed}$$

(vital element failed/WS-SAGAS failed)

$$RP(E^v_{i.k}) = \sum_{p=1}^{|CEL(E^v_{i.k})|} I(ee^p_{i.k}, e^q_{i.k-1})$$

Definition 7.4 (The Failure Recovery Time)
We define the *failure recovery time*(exactly $R(E^v_{i.k})$) as the time required for $E^v_{i.k}$ to recover from its own failures and from the failure of other elements. We note that an element failure can trigger at most the cardinality of its CEL forward recoveries; however, it can be subject to only one backward recovery, triggered by another element. The expression of $R(E^v_{i.k})$ is defined by the following equation:

$$R(E^v_{i.k}) = \sum_{1 \le p \le |CEL(E^v_{i.k})|} For(E^v_{i.k}) + Back(E^v_{i.k}) \quad (7.4)$$

$$For(E^v_{i.k}) = \sum_{1 \le p \le |CEL(E^v_{i.k})|} T(E^v_{i.k}, ee^p_{i.k}, ws^p_{i.k})$$

$$Back(E_i) = xor(Comp(E^v_{i.k}), Abort(E^v_{i.k}))$$

- $For(E^v_{i.k})$ is the total time spent in retrying $E^v_{i.k}$ by the other engine-ws couple from $CEL(E^v_{i.k})$ every time the allocated engine fails to commit $E^v_{i.k}$.
- $For(E^{\bar{v}}_{i.k})$ is always equal to 0 for a `nonvital` element because its execution is not retried even if it fails.
- $Back(E^v_{i.k})$: In a backward recovery, the mechanism to be applied depends on the composition specification model; the more widely used techniques are rolling back, aborting, and compensation. In the FENECIA approach, the *backward recovery time* is the time necessary to trigger a backward recovery mechanism by aborting all the elements still executing and compensating all the already committed elements. Therefore, the entity $Back(E^v_{i.k})$ can be equal to the *Compensation time*($Comp(E^v_{i.k})$) if another element from the same process that comes after $E^v_{i.k}$ failed and triggered a backward recovery when $E^v_{i.k}$ had already committed; alternatively, it is equal to the *Abort time*($Abort(E^v_{i.k})$) if another element from the same process that is executed concurrently with $E^v_{i.k}$ failed and triggered a backward recovery while $E^v_{i.k}$ is still being executed.
- Depending on location of the failure (of the element itself or of other elements) and on the element's *state* and *vitality degree*, the defined expression of $R(E^v_{i.k})$ in Equation (7.4) varies as follows:

Case 1. : $CEP(WS\text{-}SAGAS_i, ee^q_{i.k}).\overrightarrow{DR}(E_{i.k}).state =$ `Committed`

(element committed / WS-SAGAS committed)

$$R(E^v_{i.k}) = \sum_{p=1}^{q-1} For(E^v_{i.k}) = \sum_{p=1}^{q-1} T(E^v_{i.k}, ee^p_{i.k}, ws^p_{i.k}) \text{ (vital)}$$

$$R(E^{\bar{v}}_{i.k}) = 0 \text{ (nonvital)}$$

Case 2. : $CEP(WS\text{-}SAGAS_i, ee^q_{i.k}).\overrightarrow{DR}(E_{i.k}).state =$ `Compensated`

(element committed/WS-SAGAS failed)

$$R(E^v_{i.k}) = \sum_{p=1}^{q-1} For(E^v_{i.k}) + Comp(E^v_{i.k}) \text{ (vital)}$$

$$R(E^{\bar{v}}_{i.k}) = Back(E^{\bar{v}}_{i.k}) = Comp(E^{\bar{v}}_{i.k}) \text{ (nonvital)}$$

Case 3. : $CEP(WS\text{-}SAGAS_i, ee^r_{i.k}).\overrightarrow{DR}(E_{i.k}).state =$ `Aborted`

(element still executing with $ee^r_{i.k}$ /WS-SAGAS failed)

$$R(E^v_{i.k}) = \sum_{p=1}^{r-1} For(E^v_{i.k}) + Abort(E^v_{i.k}) \text{ (vital)}$$

$$R(E^{\bar{v}}_{i.k}) = Back(E^{\bar{v}}_{i.k}) = Abort(E^{\bar{v}}_{i.k}) \text{(nonvital)}$$

Case 4. : $CEP(WS\text{-}SAGAS_i, ee^{|CEL(E^v_{i.k})|}_{i.k}).\overrightarrow{DR}(E^v_{i.k})$ $.state =$ `Failed`

(element failed/WS-SAGAS failed)

$$R(E^v_{i.k}) = \sum_{p=1}^{|CEL(E^v_{i.k})|} For(E^v_{i.k})$$

7.3.2 The Execution Time of a WS-SAGAS and of a Process

The execution time of a WS-SAGAS $WS\text{-}SAGAS_i$ is derived from the estimates of the execution time of the different elements it combines. Because these elements might be orchestrated in different ways to structure the CWS, we propose defining an expression that estimates the execution time for each of the different aggregation patterns that WS-SAGAS defined (see Table 7.1).

Because WS-SAGAS are recursively nested, the execution time of the uppermost WS-SAGAS in the hierarchy is equal to the execution time of the entire process. The expression to estimate the execution time of a WS-SAGAS $WS\text{-}SAGAS_i$ is derived as follows:

$$T(P_i)_{prob} = T(WS\text{-}SAGAS_i)_{prob}$$

$$T(WS\text{-}SAGAS_i)_{prob} = T(WS\text{-}SAGAS_{i.a})_{prob} + (\prod_{\ell=i.1}^{i.n_{i,1}} T(E_\ell)_{prob} | DR(E_\ell).type = \texttt{atomic}) \ .$$

The expression $\prod$ means that the execution time is derived according to the aggregation pattern that connects the different elements that we defined in (Table 7.1). By deriving the execution time of every WS-SAGAS in the hierarchy in the same way as for $WS\text{-}SAGAS_{i.a}$, we can derive the expression below for the execution time of the whole process:

$$T(P_i)_{prob} = (\prod_{\ell=i.1}^{i.a.*.b.*.c.n_{i,m_i}} T(E_\ell)_{prob} | DR(E_\ell).type = \texttt{atomic}) \ .$$

We emphasize that the estimate of the execution time—obtained by combining the execution time of the different elements—does not include the time spent in the coordination/conversation between peer-engines, that is, the time spent in the execution control delegation (predelegation phase and synchronization phase) or the time spent in *CEP* communication, that is, the time spent in propagating the synchronization signals. Second, it does not include the time spent in generating the CEL/REL and the time spent in updating *CEP*. Any of these different times fall into the category of *control delegation time*. We define in what follows how the *control delegation time* is estimated in our approach. In general, it can be incorporated in: *(i)* the time taken by a one-way SOAP message, (ii) the time taken by a two-way synchronous SOAP message (request/response), or *(iii)* the time taken by an update/search query.

Definition 7.5 (The Control Delegation Time)
The control delegation time is the time between the moment of termination of the execution of one or many elements by a first entity (i.e., one or more engines) and the moment of starting the execution of one or more elements by a second entity (i.e., one or more engines). It typically includes the time necessary to generate the CEL/REL and select the engine(s), the time spent in the predelegation and synchronization phases, and the time spent in updating and communicating the *CEP* content.

Depending on the WS-SAGAS *OR*, the control delegation can be: *(a)* from one engine (executor or coordinator) $e^q_{i.k}$ to another engine (executor or coordinator) $e^p_{i.k+1}$ (i.e., $e^q_{i.k}$ and $e^p_{i.k+1}$ control a sequence of elements) (see case 1). *(b)* from one engine (executor or coordinator) $e^q_{i.k}$ to the set of engines $\mathscr{S}(e^q_{i.k})_{\text{succ}}$ allocated to control the set of elements $\mathscr{S}(E_{i.k})_{\text{succ}}$ (see case 2). *(c)* from many engines (executor or coordinator) $\mathscr{S}(e^p_{i.l})_{\text{pre}}$ controlling the set of elements $\mathscr{S}(E_{i.l})_{\text{pre}}$ to the engine (executor or coordinator) $e^p_{i.l}$ allocated to $E_{i.l}$ (see case 3). *(d)* from many engines (executor or coordinator) to many engines (executor or coordinator); this case can be deduced by combining the second and third cases.

Case 1 :*(one engine / one engine)*

$$D(e^q_{i.k}, e^p_{i.k+1}) = GS(e^q_{i.k}, CEL(E_{i.k+1})) + PC(e^q_{i.k}, e^p_{i.k+1}) \ ,$$

where:

- $D(e^q_{i.k}, e^p_{i.k+1})$ is the time to delegate the execution control between the engine $e^q_{i.k}$ and engine $e^p_{i.k+1}$;

Pattern	Notation	Probable execution time
Sequence	$(E_{i.k};E_{i.k+1})$	$T(E_{i.k})_{prob}+T(E_{i.k+1})_{prob}+D(ee^q_{i.k};ee^p_{i.k+1})$
Parallel	$(E_{i.k};(E_{i.k+1}\|\|...\|\|E_{i.k+j}))$	$T(E_{i.k})_{prob}+\max_{\ell=1}^{j}(T(E_{i.k+\ell})_{prob})+\max_{\ell=1}^{j}D(ee^q_{i.k},ee^p_{i.k+\ell})$
Selection	$(E_{i.k}\bigcirc(E_{i.k+1}\|\|E_{i.k+2}\|\|...\|\|E_{i.k+j}))$	$T(E_{i.k})_{prob}+\max(T(\mathscr{S}(E_{i.k})_{\mathsf{succ}})_{prob})+\max(D(ee^q_{i.k},\mathscr{S}(E_{i.k})_{\mathsf{succ}}))$
Switch	$(E_{i.k}\lhd(E_{i.k+1}\|\|E_{i.k+2}\|\|\ldots\|\|E_{i.k+j}))$	$T(E_{i.k})_{prob}+T(\mathscr{S}(E_{i.k})_{\mathsf{succ}})_{prob}+D(ee^q_{i.k},\mathscr{S}(ee^q_{i.k})_{\mathsf{succ}})$
Rendezvous	$((E_{i.k}\|\|E_{i.k+1}\|\|\ldots\|\|E_{i.k+j})\diamond E_{i.l})$	$\max(T(\mathscr{S}(E_{i.l})_{\mathsf{pre}})_{prob})+T(E_{i.l})_{prob}+\max(D(\mathscr{S}(ee^p_{i.l})_{\mathsf{pre}},ee^p_{i.l}))$
Selective merge	$((E_{i.k}\star\ldots\star E_{i.k+j})\,\square\!\!\rightarrow E_{i.l})$	$\max(T(\mathscr{S}(E_{i.l})_{\mathsf{pre}})_{prob})+\|\mathscr{S}(E_{i.l})_{\mathsf{pre}}\|T(E_{i.l})_{prob}+\max(D(\mathscr{S}(ee^p_{i.l})_{\mathsf{pre}},ee^p_{i.l}))$
Exclusive merge	$((E_{i.k}\star\ldots\star E_{i.k+j})\,\square\!\!\Rightarrow E_{i.l})$	$T(\mathscr{S}(E_{i.l})_{\mathsf{pre}})_{prob}+T(E_{i.l})_{prob}+D(\mathscr{S}(ee^p_{i.l})_{\mathsf{pre}},ee^p_{i.l})$
Iterative	$(E_{i.k};\lambda E_{i.k+1})$	$T(E_{i.k})_{prob}+\lambda T(E_{i.k+1})_{prob}+D(ee^q_{i.k};ee^p_{i.k+1})$

Table 7.1 The probable execution time expressions of WS-SAGAS patterns

- $GS(e^q_{i.k},CEL(E_{i.k+1}))$ is the time spent in generating either $CEL(E_{i.k+1})$ or $REL(E_{i.k+1})$ and in selecting an engine, we assume that the selected engine is $e^p_{i.k+1}$;
- $PC(e^q_{i.k},e^p_{i.k+1})$ is the predelegation phase duration and the time spent in updating and communicating the *CEP* content.

Case 2 :*(one engine / many engines)*

$$D(e^q_{i.k},\mathscr{S}(e^q_{i.k})_{\mathsf{succ}})=GS(e^q_{i.k},CEL(\mathscr{S}(E_{i.k})_{\mathsf{succ}}))+PC(e^q_{i.k},\mathscr{S}(e^q_{i.k})_{\mathsf{succ}})\ .$$

Case 3 :*(many engines / one engine)*

$$D(\mathscr{S}(e^p_{i.l})_{\mathsf{pre}},e^p_{i.l})=GS(\mathscr{S}(e^p_{i.l})_{\mathsf{pre}},CEL(E_{i.l}))+PC(\mathscr{S}(e^p_{i.l})_{\mathsf{pre}},e^p_{i.l})\ .$$

Definition 7.6 (The Engine-WS Waiting Period)
We define the *engine-ws Waiting Period* to avoid the situation where an engine $e^p_{i.k}$ allocated to an element $E_{i.k}$ waits eternally for an answer from a WS $ws^p_{i.k}$ that might never come, if the WS fails to respond. After the *engine-ws Waiting Period* (exactly $W(e^p_{i.k},WS^p_{i.k})$) has elapsed and no information has been received of the execution progress of the WS $ws^p_{i.k}$, then the engine $e^p_{i.k}$ must consider itself failed and a recovery has to be triggered. The question is how to determine $W(e^p_{i.k},ws^p_{i.k})$, in case the element $E_{i.k})$ has not yet been attempted.

Usually, WS providers advertise the processing time of their provided WS or offer methods to inquire about it. This could be used here to compute an initial estimate of the entity $W(e^p_{i.k},ws^p_{i.k})$. Later, when the element is invoked a number of times, $W(e^p_{i.k},ws^p_{i.k})$ can be estimated on the basis of the observation results of these past invocations(α). In Equation (7.5)), $T(E_{i.k})^1_{opt}$ is the *Optimistic Execution Time* of $E_{i.k}$ when invoked for the 1st time:

$$W(e^p_{i.k},ws^p_{i.k})=\max(T(E_{i.k})^1_{opt},\ldots,T(E_{i.k})^{\alpha}_{opt})\ . \quad (7.5)$$

7.4 Reliability Property Characterization

In this section, we describe the QoS of a CWS in terms of reliability. It is widely recognized that the way the reliability is defined and assessed is specific to the domain considered but, in general, the reliability concept is always kept somehow closely related to the system behavior and its failure history.

In our approach, in characterizing the reliability dimension, we introduce a new category of reliability, named *reliability tendency*, that builds heavily on the *state* concept attached to each element from a process, depicted as a hierarchy of nested WS-SAGAS. Our proposal was motivated by the reliability and the *state* concepts being very closely related and that the element's contribution to the overall process reliability estimation varies from one *state* to another.

To this end, we propose collecting the process past invocation in a history (i.e., the different copies of *CEP* stored in every engine execution log).

Later, the collected history is used to analyze thoroughly the different elements' behavior when executed by tracking their different *states* and by how and when they transit between different *states*. To estimate the *reliability tendency*, the process execution history is used to derive for each element the element's *Terminal States Set (TSS)*, the element's *State Tendency Set (STS)*, and the *State Reliability Contribution (SRC)*.

Definition 7.7 (The Terminal States Set (TSS))
Each `atomic` element $E_{i.k}$, after being invoked for execution as a component from a subtransaction $WS\text{-}SAGAS_i$, has a *Terminal State (exactly* $TS(E_{i.k})$*)* with which its invocation is terminated. If the element $E_{i.k}$ is allocated to an engine executor $ee^p_{i.k}$, every progress in the element's execution is reflected on the locally stored copy of *CEP* on the engine's $ee^p_{i.k}$ side by updating the attribute *state* in *CEP* of the element $E_{i.k}$. When the element's execution is finished, the element's *TS* is updated as follows:

$$TS(E_{i.k}) := CEP(WS\text{-}SAGAS_i, ee^p_{i.k}).\overrightarrow{DR}(E_{i.k}).state \ .$$

After α invocations of the same process P_i, for each element, a *Terminal State Set* (exactly $TSS(E_{i.k})$) is formed.

The $TSS(E_{i.k})$ is a set of 2-tuples where the occurrence number of each 2-tuple is associated with each *terminal state* as a *TS* after α invocation of the element $E_{i.k}$. If we assume that there are β possible *TS*, and each *TS* $TS(E_{i.k})^x$ is associated with a number of occurrences occ^x, as x ranges over $[1..\beta]$, then $TSS(E_{i.k})$ is formulated as follows:

$$TSS(E_{i.k}) = \{(TS(E_{i.k})^1, occ^1), \ldots, (TS(E_{i.k})^x, occ^x), \ldots, (TS(E_{i.k})^\beta, occ^\beta)\} \quad \text{with: } \sum_{1 \le x \le \beta} occ^x = \alpha$$

The cardinality of $TSS(E_{i.k})$ depends on the different possible *TS* of the element. In the FENECIA approach, a `compensatable atomic` element at a given moment can be in one of the following *states*:

$$DR(E_{i.k}).state \in \{\texttt{Waiting, Executing, Failed, Aborted, Committed, Compensated}\}$$

As we follow a transactional execution, the `Executing` state cannot be a *TS*; therefore, β verifies:($\beta = 5$) and the *TS* of any element can only be in:

$$TS(E_{i.k}) \in \{\texttt{Waiting,Failed,Aborted,Committed,Compensated}\}$$

$TSS(E_{i.k})$ is denoted as follows:

$$TSS(E_{i.k}) = \{(\texttt{Waiting}, occ^1), (\texttt{Failed}, occ^2), (\texttt{Aborted}, occ^3), (\texttt{Committed}, occ^4), (\texttt{Compensated}, occ^5)\} \quad \text{with: } \sum_{x=1}^{5} occ^x = \alpha$$

Definition 7.8 (The State Tendency Set (STS))
After α invocations of an `atomic` element $E_{i.k}$, at least one *Terminal State (*$TS(E_{i.k})$*)* from the different possible *TS* tends to have the largest occurrence number. We introduce the concept of *State Tendency Set* (exactly $STS(E_{i.k})$), as the set that contains the *TS* that has the largest occurrence number after α invocations of an element.

That is, $STS(E_{i.k})$ must verify the condition $STS(E_{i.k}) \subseteq TSS(E_{i.k})$; that is, $STS(E_{i.k})$ is the set of *TS* tuples that are included within the *TSS* $TSS(E_{i.k})$ of $E_{i.k}$ and that has the largest occurrence number, after α invocations of $E_{i.k}$.

Definition 7.9 (The State Reliability Contribution (SRC))
We assume that from one *TS* to another, the contribution to reliability differs: terminating the execution of an `atomic` element $E_{i.k}$ in the `Failed` *state* negatively affects the reliability, contrary to the `Committed` *state*, which would contribute positively by increasing the reliability. Accordingly, we define this concept as the *State Reliability Contribution* (exactly *SRC*) of a particular *TS*. We assume that a transition from one *TS* to another makes the *SRC* stronger if it is toward a *state* denoting execution success, and it is contributing negatively and making the *SRC* weaker if it is toward a *state* denoting a faulty execution. The definition of the *SRC* of each *state* depends greatly on the environment characteristics considered (e.g., number of TS, possible states, states transitions, etc). Initially, the different *SRC* can be allocated a value based on the designer's judgment (i.e., when the designer wishes to emphasize the more error-prone elements, a stronger *SRC* values to the faulty *TS* can be assigned). Typical values of the *SRC* of the *TS* of WS-SAGAS are as follows:

$$\{(SRC(Waiting) = 0), (SRC(Failed) = -1), (SRC(Aborted) = -0.5), (SRC(Committed) = 1), (SRC(Compensated) = +0.5)\}$$

However, in some systems, making the human intervene to define the different *SRC* may not be desirable because the system is to be completely automated. In such a case, making the system able to define automatically the different *SRC* and to revise them when required is necessary. We will address this issue in future work.

Definition 7.10 (Element Reliability Tendency (RT))
The concept of *Reliability Tendency* of an `atomic` element $E_{i.k}$ (exactly $RT(E_{i.k})$) is derived from its *TSS* and the different *SRC* values, as shown in Equation (7.6).

$$RT(E_{i.k}) = \frac{\sum_{TS(E_{i.k})^x \in TSS(E_{i.k})} occ^x.SRC(TS(E_{i.k})^x)}{|TSS(E_{i.k})|} \quad (7.6)$$

Definition 7.11 (Process Reliability Tendency)
Any process P_i depicted as a hierarchy of nested WS-SAGAS can be formed by both `vital` and `nonvital` elements. Because the failure of a `nonvital` element is not handled in the same way as the failure of a `vital` element and investigating the reasons for failure of the `nonvital` elements is secondary, we propose considering only the `vital` elements to estimate the overall process *RT* and ignoring the `nonvital` elements.

Therefore, the *RT* of a process P_i formed by n_i elements where n'_i elements are `vital` and distributed between m_i nesting levels is estimated by the following formula:

$$RT(P_i) = RT(WS\text{-}SAGAS_i) = \sum_{\ell=i.1}^{i.a.*.b.*.c.n_{i,m_i}} RT(E_\ell) | DR(E_\ell).vitality = vital / n'_i \ .$$

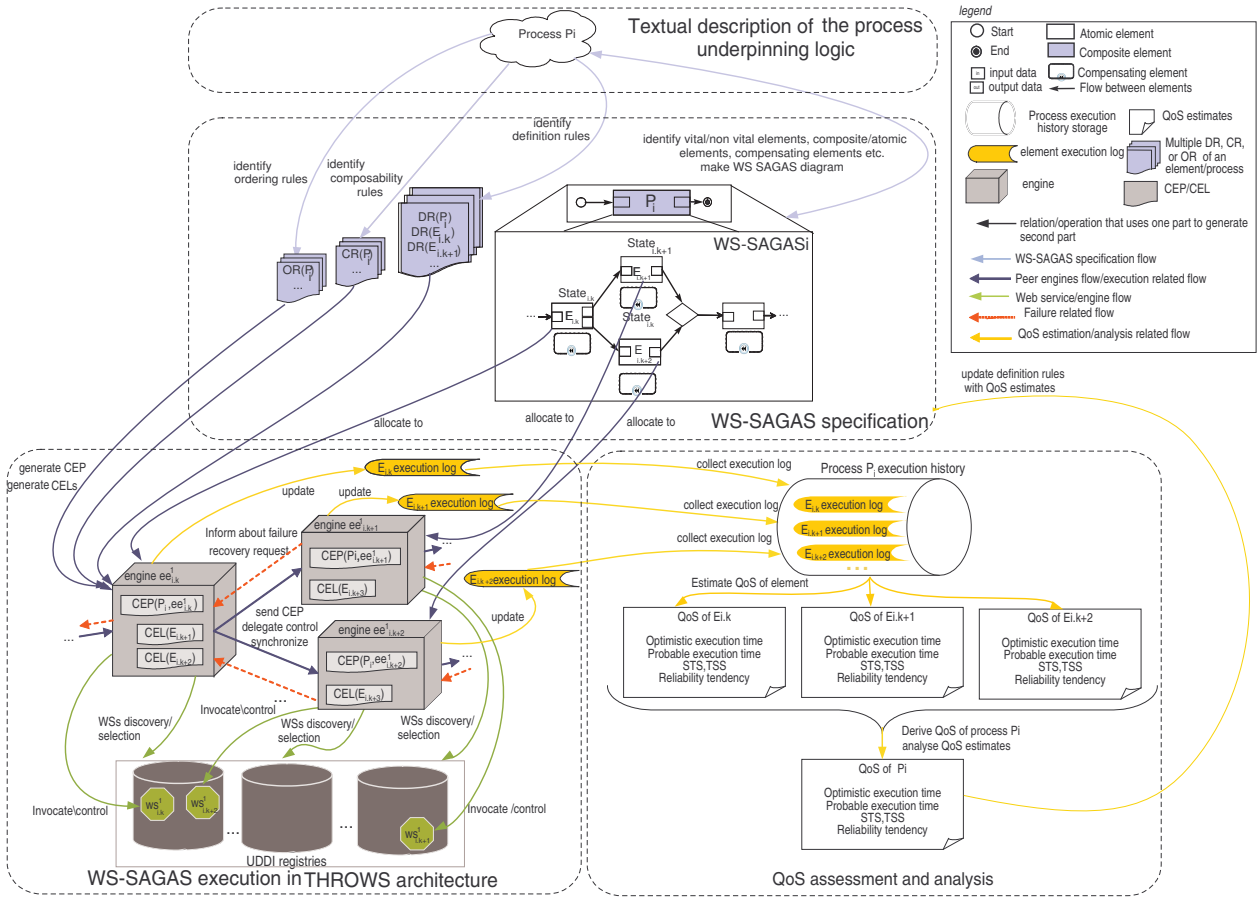


Fig. 7.1 FENECIA Framework

8 FENECIA Framework Validation

Below we describe two axes of validation for our FENECIA framework models and artifacts. In the first part of our validation, we present a prototype that provides an implementation of our execution architecture's (THROWS) main functionalities. The prototype implementation is intended to show that the failure recovery-oriented features that THROWS architecture provides are feasible with the current WS technologies.

In the second part of our validation, we show the applicability of our failure-aware QoS estimation and analysis model. To this end, we provide a case study of using our model for a real-world example of CWS assembled using Jopera [61,62], a visual WSC tool.

8.1 Prototyping

As in our prototype we target a fully automatic WSC. We have to describe the semantics of the models and elements of our FENECIA framework in a clearer and unambiguous way that can be easily automated or transformed into any platform specific code for automatic execution. In achieving this target, we translate all the FENECIA models and elements (i.e., the textual notation of a WS-SAGAS, THROWS architecture CEP, CEL, and REL concepts, and QoS model attributes) into an XML-based language. Our proposed XML-based specification language is defined and expressed according to a well-formed structure, the XML Schema description (XSD). An XML-based description of a WSC serves as an input to our prototype as we show below in this section.

The prototype implementation is heavily based on the Java programming language and on a set of WS enabling technologies. In the remainder of this section, we describe our implementation and we sketch a case study and report on its execution. We used a simplified version of the travel itinerary reservation scenario described in the sections above, with only one nesting level (i.e., all elements are assumed to be atomic).

8.1.1 Implementation Environment Choices and Motivations

Our prototype implements a logically distributed prototype of THROWS architecture to execute CWS specified as WS-SAGAS and described using our XML-based language.

We have made extensive use of Java threads and of a number of synchronization mechanisms to allow the concurrent execution of engines. Although a physically distributed prototype appears more suitable, the circumstances we cite below precluded us from implementing such a system:

- The current progress in WS architecture in terms of semantically equivalent WS availability is very limited as there are few UDDI registries in operation (maintained by IBM, Microsoft, etc.). Moreover, these registries are still very small and most of their entries do not work or do not correspond to any real service. Furthermore, most of the UDDI registries in place today are private registries operating inside companies or maintained by a set of companies privately. Therefore, they are not of use to us.
- The current unpredictability of the WS environment, which makes WS appear and disappear on daily basis, makes the dynamic WS discovery process very likely to fail in all attempts. This may considerably impair our results and may even make execution impossible.
- Even if we assume that a wide range of WS equivalents, in terms of functionalities, was provided, fully automatic and dynamic WS discovery and selection remains an unresolved issue with very few solutions. Even large enterprises agree that manual WS discovery and selection remains the most efficient approach and that automated discovery of WS requires accurate descriptions of the functionality of WS and an approach to finding WS based on the functionality they provide. This remains infeasible because it is not possible for a service client to have full knowledge of the exact form and meaning of all the service's WSDL in advance, and this for all the WS hosted on different providers.

These conditions, and in particular the last, have directed our choice toward building our private UDDI registry and publishing our own WS locally in this registry. In building our WS, we deliberately created a WS that shares the same semantics and syntax (as represented by their WSDL message definitions); thereby, an automatic WS discovery and selection can be performed successfully, the call to the service succeeds, and no unexpected results can be returned. Our prototype features the following functionalities:

- Of the eight different aggregation patterns we defined in WS-SAGAS, our prototype supports only three: the sequence, parallel, and rendezvous patterns. Adding all the different patterns to have a full-featured implementation is feasible.
- In our prototype, we only consider the case of processes formed by `compensatable` elements alone.
- In our prototype, we consider a process with only one nesting level; therefore, we have only one engine coordinator that starts the overall process execution and that is responsible for terminating the process.

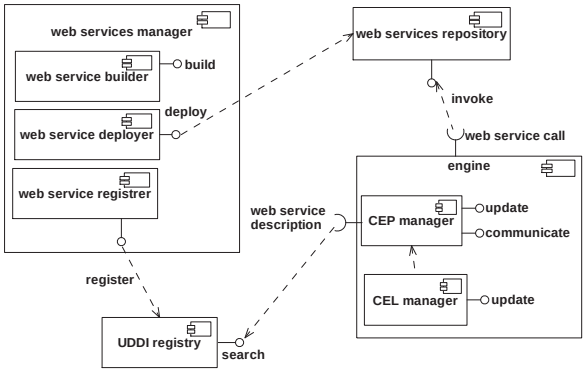


Fig. 8.1 Simplified conceptual architecture for our prototype modeled in UML with component diagram

8.1.2 Description of Implementation Tools

We have made extensive use of the different APIs provided by Sun's JWSDP 1.2 (Java Web Services Developer Pack) [63]. Of the technologies that JWSDP contains, we have chiefly used the Java API for XML Registries (JAXR) with the Registry Server for building, deploying, and publishing the WS we used. All the WS that we needed for our system were built and deployed in an XML registry that followed the UDDI specification (version 1.2). We used JAXR to access this XML registry. To build the different WS, we used Java API for XML-based RPC (JAX-RPC). The WS invocation and its context communication is done implicitly using synchronous SOAP messages over HTTP. Moreover, all the communications between the different modules used SOAP with Attachments API for Java (SAAJ). Depending on the CWS execution stage, the exchanged SOAP messages may encapsulate different forms of XML documents. Those XML documents were parsed using JAXP and manipulated with JDOM and DOM.

8.1.3 Description of Prototype Components

The implementation featured components described in the conceptual architecture model of Figure 8.1. We implemented two main modules, the *Web Services Manager* and the *Engine*. Each *Engine* encapsulates two submodules, the *CEP Manager* and the *CEL Manager*. Each *Manager* has two main functions, an information update and retrieval function and a communication function, that is, sending and receiving SOAP messages.

THE WEB SERVICES MANAGER: This implements the different functions that relate to WS creation, deployment, and invocation. It consists of the following three submodules: *(1)* the *Services Builder*, which chiefly uses JAX-RPC API and several other tools (e.g., wscompile, wsdeploy) to generate the WS endpoints, their clients, and their WSDL documents; *(2)* the *Services Deployer*, which deploys the built WS in a Web container (we used TOMCAT); *(3)* the *Services Register*, which is responsible for registering the different WS in our private UDDI registry.

```
<fnc:elementADR type="false" vitality="vital"
        description="trip_information" behavior="compensatable"
        elementID="E_1.1" engineID="null">
        <fnc:state value="waiting" />
        <fnc:operation operationID="operation1">
          <fnc:param paramID="departDate" paramtype="inout">
            january 12 </fnc:param>
          <fnc:param paramID="destination"
            paramtype="inout">
            japan </fnc:param>
          <fnc:param paramID="returnDate" paramtype="inout">
            february 12 </fnc:param>
          <fnc:param paramID="customerName"
            paramtype="inout">
            Ahmed </fnc:param>
        </fnc:operation>
</fnc:elementADR>
```

Fig. 8.2 Excerpt from the simplified trip reservation process: different elements' attributes with the values affected

THE ENGINE: Our prototype implements two types of engine: engine coordinator and engine executor. The number of instantiated engines depends on the number of elements and the number of nesting levels of a process. The main difference is that an engine coordinator is not responsible for a WS invocation. Both of the two forms of engines contain two submodules, the *CEL Manager* and the *CEP Manager*, and the encapsulated functions are the same: *(1) CEL manager*: The main function of this component is to generate the *CEL* of the next *element(s)* to be executed. To this end, it sends a query with the *element* description (available in the *active definition rule* of the element in *CEP*) to the *UDDI registry* to search for WS with functionalities matching the description and which are published in the registry. On receiving a response to the discovery query, a new engine executor is allocated and a new engine *DR* added to the *CEL* for each WS discovered. This module is also used to select an engine executor from a *CEL* document. *(2) CEP Manager*: This mainly updates and monitors the *CEP* document stored on the engine to which it appertains. Typically, an update operation changes an element's *state* when a new SOAP message is received, for example, a message that tells that the WS execution was successful.

8.1.4 Detailed Description of Typical Execution Steps of a Process in our Prototype

We describe the different steps of the execution of a simplified version of the trip reservation process that we used in the sections above.

1. CUSTOMER REQUEST SUBMISSION. The execution starts when a customer inputs his request *(the destination, the departure date, the return date, and his name)*. Submitting the request entails saving the entered values in the *CEP document*. The trip request is simulated by assigning actual values to the different fields in the XML document. (Listing 8.2) is an excerpt from the initial CEP document of the process defined in our XML language.

```
-- The selected element is : E1.1 Desc:trip parallel with :none
  this CEP number of elts is :4  dirCEL is : E1.1
 Organization Query string is trip
 Service Query string is trip
 Created connection to registry
 Got registry service and query manager
 Org name: trip organizer
 company Org description:
 trip reservation: hotel and airplane ticket booking with car rental(optional)
 Org key id: f9e4f93d-2bf9-e4f9-7297-2864692ad619
 Contact name: Neila BEN LAKHAL
 Phone number: (012) 345-678
 Email Address: neila.benlakhal@Voyager.com
 Service name: TripService
 Service description: trip service Tunisia
 Access URI: http://localhost:8080/trip-jaxrpc/trip Service
 name: JapanTripService
 Service description: trip service Japan
 Access URI: http://localhost:8080/japantrip-jaxrpc/japantrip
  2
WS name = TripService
WS name = JapanTripService
WS URI = http://localhost:8080/trip-jaxrpc/trip
WS URI = http://localhost:8080/japantrip-jaxrpc/japantrip
WS binding = tripInformation
WS binding = TripIF
WS binding = String_1
WS binding = String_2
WS binding = String_3
WS binding = String_4
WS binding = japantripInformation
WS binding = JapanTripIF
WS binding = String_1
WS binding = String_2
WS binding = String_3
WS binding = String_4
TripServicehttp://localhost:8080/trip-jaxrpc/trip
JapanTripServicehttp://localhost:8080/japantrip-jaxrpc/japantrip
--GENERATED CEL contained in file :
C:\eclipse-SDK-2.1.1-win32\eclipse\workspace\THROWS4\\E1.1\CEL.XML
```

Fig. 8.3 Excerpt from the messages output on the Java execution console to monitor the execution progress. In this part, a WS discovery is performed by querying the UDDI registry for WS to the element $E_{1.1}$; two WS are found and their binding information is used to generate a CEL with two engine elements.

This *CEP document* is updated and handled by the different *engines* throughout the process execution. By the end of its execution, the *CEP document* contains information about the execution success (e.g., flight booked, hotel ticket reserved, car reserved) or failure (e.g., no available flight).

2. Element selection and CEL generation The engine coordinator ec_1^1 runs on the server side. When it receives a new *CEP document* it starts processing by parsing the XML document and selecting a *current element*, that is, the first element to be executed. In the CEP document of Listing of (Figure 8.2), the first element is the `elementID` = "`E_1.1`". The function of going through this *CEP document* for selecting elements is attached to the *CEP Manager* module.

After an element is selected, a *CEL document* is generated. This is the responsibility of the *CEL Manager*: which receives, as input from the *CEP Manager*, a *description* of an element (here, `description` = "`trip_information`"). The `description` is used to create a query that is sent to the UDDI registry for searching WS that eventually meet the description provided.

```
<fnc:CEL>
  <fnc:engineExecutorDR engineCoordinator="ec_1-1"
     engineExecutorID="ee_1.1-1" wsdescription="TripService"
     wsdlLink="http://localhost:8080/trip-jaxrpc/trip">
     <fnc:operation operationID="tripInformation">
        <fnc:param paramID="String_1" paramtype="inout" />
        <fnc:param paramID="String_2" paramtype="inout" />
        <fnc:param paramID="String_3" paramtype="inout" />
        <fnc:param paramID="String_4" paramtype="inout" />
     </fnc:operation></fnc:engineExecutorDR>

  <fnc:engineExecutorDR engineCoordinator="ec_1-1"
     engineExecutorID="ee_1.1-2" wsdescription="JapanTripService"
     wsdlLink="http://localhost:8080/japantrip-jaxrpc/japantrip">
     <fnc:operation operationID="japantripInformation">
        <fnc:param paramID="String_1" paramtype="inout" />
        <fnc:param paramID="String_2" paramtype="inout" />
        <fnc:param paramID="String_3" paramtype="inout" />
        <fnc:param paramID="String_4" paramtype="inout" />
     </fnc:operation></fnc:engineExecutorDR>
</fnc:CEL>
```

Fig. 8.4 Excerpt from the CEL of the element E_1.1 from the trip reservation process

We show in (Listing 8.3) the progress of the execution of this step in terms of messages output on the Java console.

3. Web services discovery and selection To ensure interoperability of the *engine* (here considered as the JAXR client) and the UDDI registry implementation, the SOAP messages that contain the query (and its corresponding results) are handled completely unseen using SAAJ. Searching the UDDI registry for WS results in a list of all the organization(s) that contain(s) WS we are interested in (i.e., they have capabilities that meet the functionalities of the current element E_1.1). When we query the UDDI registry, the result is all the organizations with the name that contains the string trip.

4. CEP generation The retrieved information, as a result of the query, is parsed for details about the organization(s) and the services it/they provide(s) and is used to generate the *CEL document* (refer to Listing 8.4) for the automatically generated CEL document for the element E_1.1. To each WS, an engine executor is allocated, that is, a new engine executor ID engineid is dynamically created and stored in the *CEL Document* coupled with the WS information as an engine definition rule (see Listing 8.4 and Listing 8.3 for the *CEL* document content).

After terminating the *CEL document* generation, a candidate engine executor is selected and the *CEP Manager* updates the *CEP document*. Here, the selected engine executor is engineExecutorID = "ee_1.1 − 1".

5. CEP update and control delegation When preparing the necessary data for effectively allocating the execution control to the engine executor ee_1.1 − 1, the *CEP document* is encapsulated and sent as a SOAP message. Simultaneously, a new thread engine ee_1.1 − 1 is created, the received *CEP* document is stored locally, and a response is sent back to ec_1 − 1 notifying that the SOAP message was received and the execution launched.

```
the parameter id : destination has the value: japan is an input parameter
 the parameter id:departuredate has the value: jan12 is an input parameter
the parameter id : returndate has the value: feb12 is an input parameter
 the parameter id : name has the value: Ahmed is an input parameter

input:[destination, departuredate, returndate, name]output:[] start running
ws Endpoint
address=http://localhost:8080/trip-jaxrpc/trip Mr/Ms Ahmed, your trip
information were received
successfully
 * Your desired destination is: japan
 * Your chosen departure date is :jan12
 *Your chosen return date is :feb12
success
```

Fig. 8.5 Excerpt from the messages output on the Java execution console to monitor the execution progress. In this part, the WS allocated to element E_1.1 and controlled by engine ee_1.1 − 1 is invoked and a "success" message
returned.

6. Control delegation finalization and WS invocation preparation After receiving the execution control, the engine executor $ee_{1.1-1}$ updates in the *CEP document* the *state* of E_1.1 from Waiting to Executing, and extracts from the *CEP document* the values of the parameters with which the WS will be invoked (see Listing 8.5).

7. WS invocation The engine executor invokes the WS client. The JAX-RPC runtime is responsible for receiving this WS invocation message within the client call and for passing it to the WS endpoint. In addition, when the WS finishes executing, it passes the results to the JAX-RPC runtime. Likewise, the latter takes care of handing over these results to the *CEP Manager*.

At this point, depending on the WS execution progress, two scenarios can occur: the WS failure or success. Because we implemented the WS, their failure probability was low. The execution often terminated with success so to show how failure handling is performed we forced WS failure (i.e., fault injection). In what follows, we first describe the case of a scenario in which the WS execution was successful (see Listing 8.5).

In this process instance execution, the WS sends back the result of its execution to the engine executor, which uses this to update the *CEP document* to add the WS execution result and to add the required change in the execution progress. In the case of the engine executor ee_1.1 − 1, the only update is changing the element E_1.1's *state* from Executing to Committed.

The next step is to proceed with the execution of the process as the current element execution is committed. To this end, the engine executor ee_1.1 − 1 finds that there are two elements, elementID = "E_1.2" and "E_1.3", that are assembled in a parallel aggregation pattern. The *CEL documents* of these elements are generated and the engine executor processes as described above and allocates the engines executors (ee_1.2 − 1 and ee_1.3 − 1), respectively. The *CEP document* is updated with the new allocated engines (see Listing 8.6).

```
</fnc:CEP>...
 <fnc:elementADR type="false" vitality="vital"
          description="trip_information" behavior="compensatable"
          elementID="E_1.1" engineID="ee_1.1-1">
          <fnc:state value="committed" />
          ...
</fnc:elementADR> <fnc:elementADR type="false" vitality="vital"
          description="book_flight" behavior="compensatable"
          elementID="E_1.2" engineID="ee_1.2-1">
          <fnc:state value="waiting" />
          ...
</fnc:elementADR> <fnc:elementADR type="false" vitality="vital"
          description="book_hotel" behavior="compensatable"
          elementID="E_1.3" engineID="ee_1.3-1">
          <fnc:state value="waiting" />
          ...
</fnc:elementADR> <fnc:elementADR type="false" vitality="nonvital"
          description="rent_car" behavior="compensatable"
          elementID="E_1.4" engineID="null">
          <fnc:state value="waiting" />
  ...
<fnc:composabilityRules>
<fnc:composabilityRule OF="WS-SAGAS_1">
<fnc:member memberID="E_1.1" />
<fnc:member memberID="E_1.2" />
<fnc:member memberID="E_1.3" />
<fnc:member memberID="E_1.4" />
</fnc:composabilityRule></fnc:composabilityRules>
<fnc:orderingRules><fnc:orderingRule>
<fnc:sequence IN="WS-SAGAS_1">
<fnc:member memberID="E_1.1" /></fnc:sequence>
<fnc:parallel IN="WS-SAGAS_1">
<fnc:member memberID="E_1.2" />
<fnc:member memberID="E_1.3" /></fnc:parallel>
<fnc:rendezvous IN="WS-SAGAS_1">
<fnc:member memberID="E_1.2" />
<fnc:member memberID="E_1.3" /></fnc:rendezvous>
<fnc:sequence IN="WS-SAGAS_1">
<fnc:member memberID="E_1.4" /></fnc:sequence>
</fnc:orderingRule></fnc:orderingRules>
</fnc:CEP>
```

Fig. 8.6 Excerpt from the simplified trip reservation process: the execution progress can be monitored by the change in the state. Here, the first element was executed and committed successfully whereas the execution of the two following elements is about to start as engines are allocated to both.

```
REQUEST:

<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
 <SOAP-ENV:Body>
<directory>...</directory>
<fnc:CEP>
  ...
 <fnc:elementADR type="false" vitality="vital"
          description="trip_information" behavior="compensatable"
          elementID="E_1.1" engineID="ee_1.1-1">
          <fnc:state value="committed" />
          ...
</fnc:elementADR> <fnc:elementADR type="false" vitality="vital"
          description="book_flight" behavior="compensatable"
          elementID="E_1.2" engineID="ee_1.2-1">
          <fnc:state value="committed" />
          ...
</fnc:elementADR> <fnc:elementADR type="false" vitality="vital"
          description="book_hotel" behavior="compensatable"
          elementID="E_1.3" engineID="ee_1.3-1">
          <fnc:state value="committed" />
          ...
</fnc:elementADR> <fnc:elementADR type="false" vitality="nonvital"
          description="rent_car" behavior="compensatable"
          elementID="E_1.4" engineID="ee_1.4-1">
          <fnc:state value="failed" />
          ...
</fnc:elementADR>
 ...
/fnc:CEP></SOAP-ENV:Body></SOAP-ENV:Envelope>
```

Fig. 8.7 Excerpt from the simplified trip reservation process: the execution progress can be monitored with the change in the state. All three vital elements were executed and committed successfully whereas the execution of the last `nonvital` elements failed.

The *CEP Manager* component from the engine $\texttt{ee_1.3} - 1$ sends the *CEP document* to both of the new engines. The execution process start is almost the same as that described for $\texttt{ee_1.1} - 1$. The main difference is that that two elements $\texttt{elementID} =$ "`E_1.2`" and $\texttt{elementID} =$ "`E_1.3`" are assembled in a parallel pattern and they must wait for each other as they are also assembled in a rendezvous pattern (see Listing 8.6). Consequently, we divided the execution process into two phases; when every thread engine finishes a phase, it informs the other engine. The first phase is dedicated to the WS invocation and the second phase to preparing for control delegation, in case the WS invocation is successful.

By the end of the execution of both of the elements `E_1.2` and `E_1.3`, the engines $\texttt{ee_1.2} - 1$ and $\texttt{ee_1.3} - 1$ generate the *CEL document* of their successors (here $\texttt{elementID} =$ "`E_1.4`": each engine generates *CEL document* by itself and the resulting *CEL document* is a combination of the two documents. The engines $\texttt{ee_1.2} - 1$ and $\texttt{ee_1.3} - 1$ agree on the candidate engine to execute the element $\texttt{elementID} =$ "`E_1.4`" by merging their *CEL documents* and selecting an engine, $\texttt{ee_1.4} - 1$, to execute it. Subsequently, $\texttt{ee_1.4} - 1$ suspends $\texttt{ee_1.2} - 1$ and $\texttt{ee_1.3} - 1$, and proceeds with its execution.

Up to this point in the current process execution, all the elements executed had a *vitality degree* attribute in the *CEP document* equal to `vital`. For that purpose, when they fail, their failure is critical and causes the whole WS-SAGAS to which they appertain to fail, as described below in describing a process instance that failed.

If the WS attached to $\texttt{ee_1.4} - 1$ fails while being executed by $\texttt{ee_1.4} - 1$, then this implies that the failure of `E_1.4` is ignored and the entire WS-SAGAS execution proceeds, and the *state* of the element `E_1.4` is set to `Failed`.

As this element is the last element (i.e., parsing the locally stored *CEP document* and looking for an elements child from the same composite WS returns an empty list), then the success of the entire WS-SAGAS and of the whole process example is deduced by sending the locally stored *CEP document* to the engine coordinator of the whole WS-SAGAS; here `ec_1.1` receives the CEP document encapsulated in a SOAP message (see Listing 8.7 for an excerpt).

```
RESPONSE:

 <?xml version="1.0" encoding="UTF-8"?>
 <soap-env:Envelope
xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/">
 <soap-env:Header/>
<soap-env:Body> <Response>OK CEP is received correctly</Response>
</soap-env:Body>
</soap-env:Envelope>
```

Fig. 8.8 SOAP message encapsulating a confirmation message to indicate that a CEP document was delivered correctly

```
<fnc:CEP elementnb="4" enginecoordinatorID="ec_1-1" nestinglevel="1"
   ws-sagasID="WS-SAGAS_1">
   ...
 <fnc:ws-sagasADR nestinglevel="1" elementnb="4"
            behavior="compensatable" description="trip_process"
            ws-sagasID="WS-SAGAS_1"  type="true" vitality="vital">
            <fnc:state value="committed" />
            ...
 <fnc:elementADR type="false" vitality="vital"
            description="trip_information" behavior="compensatable"
            elementID="E_1.1" engineID="ee_1.1-1">
            <fnc:state value="committed" />
            ...
</fnc:elementADR> <fnc:elementADR type="false" vitality="vital"
            description="book_flight" behavior="compensatable"
            elementID="E_1.2" engineID="ee_1.2-1">
            <fnc:state value="committed" />
            ...
</fnc:elementADR> <fnc:elementADR type="false" vitality="vital"
            description="book_hotel" behavior="compensatable"
            elementID="E_1.3" engineID="ee_1.3-1">
            <fnc:state value="committed" />
            ...
</fnc:elementADR> <fnc:elementADR type="false" vitality="nonvital"
            description="rent_car" behavior="compensatable"
            elementID="E_1.4" engineID="ee_1.4-1">
            <fnc:state value="failed" />
            ...
</fnc:elementADR>
 ...
/fnc:CEP></SOAP-ENV:Body></SOAP-ENV:Envelope>
```

Fig. 8.9 Excerpt from the simplified trip reservation process. Here, all three vital elements were executed and committed successfully whereas the execution of the last nonvital elements failed. The state of the overall WS-SAGAS is deduced on the basis of the state of the vital elements.

8.1.5 Process Instance Execution Termination

In response to the received CEP document (see Listing 8.7 for an excerpt), the engine coordinator `ec_1.1` sends the SOAP message in (Listing 8.8) and resumes execution control.

The engine coordinator `ec_1.1` terminates the execution of the process instance and deduces the overall process success because all the `vital` elements were committed. The last version of the CEP document is then available on this engine (see Listing 8.9 for an excerpt).

8.1.6 Execution of a Example Process with Failure Handling

This process example was subject to a WS failure. Here, we intentionally modified the content of the response of the first WS (i.e., received response contains "failure" instead of "success") candidate to element `elementID` = "`E_1.2` in order to make the engine consider the WS as failed. Thereby, the engine `ee_1.1 – 1` needs to deduce its own failure and to delegate the execution control to the previous engine thread. For that, the current engine `ee_1.1 – 1` updates the current element `elementID` = "`E_1.1`" state to `Failed` and will communicate the *CEP document* to its direct predecessor. As it was responsible for the very first element in the currently executed WS-SAGAS, it must inform the the engine coordinator `ec_1.1` because that is the engine that has control delegated to it.

```
<fnc:elementADR type="false" vitality="vital"
            description="trip_information" behavior="compensatable"
            elementID="E_1.1" engineID="ee_1.1-1">
            <fnc:state value="failed" />
            ...
</fnc:elementADR> <fnc:elementADR type="false" vitality="vital"
            description="book_flight" behavior="compensatable"
            elementID="E_1.2" engineID="null">
            <fnc:state value="waiting" />
            ...
</fnc:elementADR> <fnc:elementADR type="false" vitality="vital"
            description="book_hotel" behavior="compensatable"
            elementID="E_1.3" engineID="null">
            <fnc:state value="waiting" />
            ...
</fnc:elementADR> <fnc:elementADR type="false" vitality="nonvital"
            description="rent_car" behavior="compensatable"
            elementID="E_1.4" engineID="null">
            <fnc:state value="waiting" />
            ...
</fnc:elementADR>
```

Fig. 8.10 Excerpt from the simplified trip reservation process; here the first element was executed by the engine $ee_{1.1-1}$ and a failure occurred.

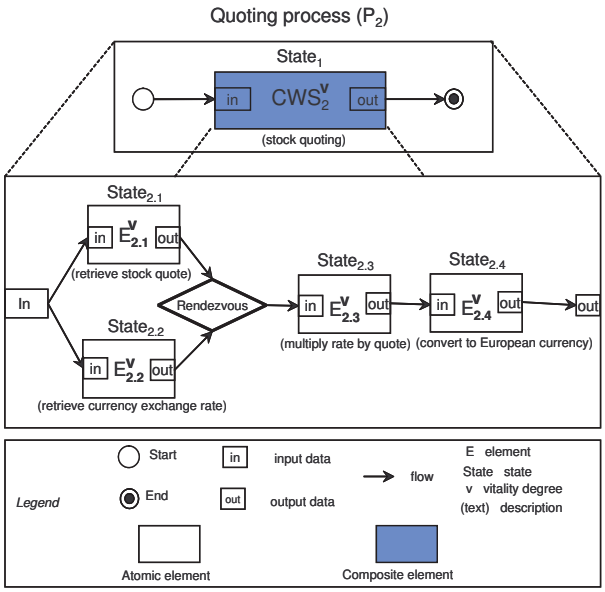
Listing of Figure 8.10 is an excerpt from the CEP document that `ee_1.1 – 1` sends to `ec_1.1`. On receiving this document in a SOAP message, `ec_1.1` handles the failure by attempting a forward recovery. First, the engine is updated (the element `E_1.1` state is set to `waiting`), the `engineid` is set to `null`, and an attempt to select another candidate engine from the *CEL document* is performed.

8.1.7 Forward Recovery in the Execution of a Process Instance

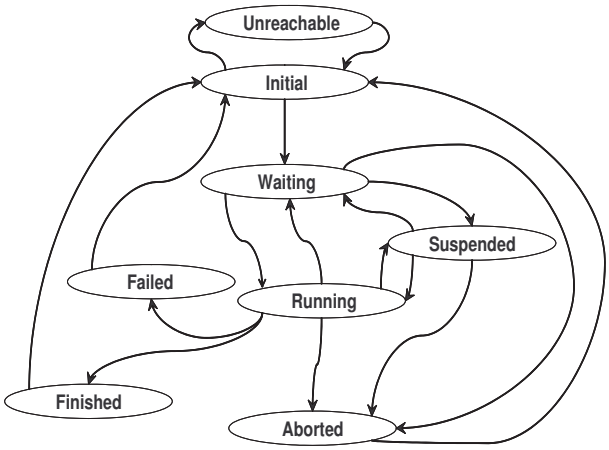
Because we have made available for each element two candidate engines, the engine coordinator `ec_1.1`, when parsing the CEL document of element `E_1.1` described in Listing 8.4, finds a second candidate engine: engine `ee_1.1 – 2`. It follows that a forward recovery is possible; `ec_1.1` updates the *CEP document* with the new selected candidate engine and the execution is resumed with `ee_1.1 – 2` as described for `ee_1.1 – 1` (see Listing 8.11).

```
<fnc:CEP elementnb="4" enginecoordinatorID="ec_1-1" nestinglevel="1"
  ws-sagasID="WS-SAGAS_1"...>
  ...
<fnc:elementADR type="false" vitality="vital"
          description="trip_information" behavior="compensatable"
          elementID="E_1.1" engineID="ee_1.1-2">
          <fnc:state value="waiting" />
          ...
</fnc:elementADR> <fnc:elementADR type="false" vitality="vital"
          description="book_flight" behavior="compensatable"
          elementID="E_1.2" engineID="null">
          <fnc:state value="waiting" />
          ...
</fnc:elementADR> <fnc:elementADR type="false" vitality="vital"
          description="book_hotel" behavior="compensatable"
          elementID="E_1.3" engineID="null">
          <fnc:state value="waiting" />
          ...
</fnc:elementADR> <fnc:elementADR type="false" vitality="nonvital"
          description="rent_car" behavior="compensatable"
          elementID="E_1.4" engineID="null">
          <fnc:state value="waiting" />
          ...
</fnc:elementADR>
```

Fig. 8.11 Excerpt from the simplified trip reservation process; here, the first element was allocated to a new engine.



(a) process depicted using our graphical notation



(b) state transition diagram from [61]

Fig. 8.12 Quoting process

8.2 QoS Model Applicability Verification

The previous section focused on showing that our failure recovery-devoted WSC specification and execution strategy is feasible only to some extent with the current level of WS technology. Our prototype could not be used to validate our QoS model unless special mechanisms and modules, dedicated to taking a log of each process instance execution in terms of execution time and change in state, need to be added. We intend to add such modules in our future work by collecting the different CEP and CEL copies in a history. In this paper, to validate our QoS model, we use data generated using JOpera [61] [62], a rapid composition tool offering a visual language and an execution platform for building distributed applications from reusable services with a CWS depicting a quoting process. Our choice of JOpera was influenced by its practicability and its similarity to our FENECIA approach in introducing the state concept. However, the JOpera tool is for static CWS with centralized execution. By using JOpera, we simultaneously show our proposal's applicability and give a foretaste of what it is like to use it with other systems.

As a process instance is executed in Jopera, the execution progress is expressed in terms of state. The execution of a WS in Jopera, when the process is invoked for execution, follows the state diagram of (Figure8.12(b)).

8.2.1 Process Description

We consider a process P_2 that retrieves quotes in a desired currency for a user-provided stock symbol. The process we defined combines four WS that we searched manually and we used from xmethods.net [64]. This process combines four `vital` elements. The first element quotes stock prices $E^v_{2.1}$ and the second performs a currency conversion $E^v_{2.2}$; these two elements are invoked in a parallel WS-SAGAS pattern and they join subsequently in a rendezvous pattern. A third element $E^v_{2.3}$ integrates the results obtained from the previous two elements. Finally, a fourth element $E^v_{2.1}$ converts the stock quote from Euro to the currency of any of the 12 Euro-participating countries and back. The quoting process P_2 is depicted using the WS-SAGAS graphical notation in (Figure 8.12(a)). However, we are obliged to delegate execution control to a centralized authority, which is responsible for execution and failure recovery of all the elements and for the WS discovery and mapping to the elements, which is performed statically, because this is how JOpera is built.

instance number	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11
$E^v_{2.1}$	1.102	0.981	2.544	1.282	1.031	0.160	0.902	1.022	1.001	5.859	31.425
$E^v_{2.2}$	1.202	1.151	2.564	0.991	1.182	0.160	1.172	1.181	9.704	5.879	31.395
$E^v_{2.3}$	1.222	2.343	2.104	0.010	20.900	0.000	0.010	0.010	0.000	0.000	0.000
$E^v_{2.4}$	2.313	1.161	6.309	0.000	0.000	0.000	2.592	1.062	0.000	0.000	0.000
WS - SAGAS $^v_2$	4.737	4.655	10.977	1.292	22.082	0.160	3.774	2.253	9.704	5.879	31.425

(a) The observed execution time (sec.) of the four Web services allocated to the elements and of the overall stock quoting process depicted as a WS-SAGAS

instance number	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11
$E^v_{2.1}$	Finished	Finished	Finished	Finished	Finished	Failed	Finished	Finished	Finished	Failed	Failed
$E^v_{2.2}$	Finished	Finished	Failed	Finished	Finished	Failed	Finished	Finished	Failed	Failed	Failed
$E^v_{2.3}$	Finished	Finished	Unreachable	Failed	Failed	Unreachable	Finished	Finished	Unreachable	Unreachable	Unreachable
$E^v_{2.4}$	Finished	Finished	Unreachable	Failed	Unreachable	Unreachable	Finished	Finished	Unreachable	Unreachable	Unreachable
WS - SAGAS $^v_2$	Finished	Finished	Failed	Failed	Failed	Failed	Finished	Finished	Failed	Failed	Failed

(b) The observed terminal states of the four Web services allocated to the elements and of the overall stock quoting process depicted as a WS-SAGAS

Fig. 8.13 Results of quoting process executed instances

8.2.2 Process Execution and Data Collection

We invoked the stock quoting process 11 times ($\alpha = 11$). The results from the invocations in terms of execution time and *Terminal States*, respectively, for each element and for the overall process are shown in (Figure 8.13) and (Figure 8.14). The reasons for failures during the running of the process instances are: *(i)* The Internet connection failed during the SOAP message roundtrip (e.g., instance ♯10). *(ii)* The WS timed out because of a network connection failure (e.g., instance ♯5). *(iii)* The WS returned a failure message because of data inconsistency (e.g., instance ♯9).

8.2.3 Execution Time Estimation and Analysis

Before stating our execution time estimate analysis, note that in Figure 8.13 if only the results of the first table are considered the only information obtained is the execution time range of the different components. There is no way to tell whether a failure took place or the reasons behind the critical variation in the execution time between instance ♯6 instance ♯11. However, even without further analysis, considering the execution progress in terms of state helps to show that failures have occurred and helps estimate the component(s) that is/are behind the failures (cells highlighted in gray in the two tables in Figure 8.13).

To analyze thoroughly the obtained data collected from executing the stock quoting process listed in Figure 8.13(a), we consider different scenarios. The differentiation into scenarios allows us to emphasize the effects of failures on the observed execution time. As shown in the two scenarios considered, how the execution time is estimated varies according to whether a failure has occurred or not.

SCENARIO 1. In this scenario, we consider the case of instances where the process P_2 execution is terminated in the `Finished` state and where no failure occurred; for example, see instance ♯1 in (Figure8.13). The following equation is defined on the basis of our proposed model to estimate the *Probable Execution Time* of *WS-SAGAS*$_2$:

$$\begin{aligned} T(P_2)_{prob} &= (\prod_{\ell=2.1}^{2.4} T(E_\ell)_{prob} | DR(E_\ell).type = \texttt{atomic}) \\ &= \max(T(E^v_{2.1})_{prob}, T(E^v_{2.2})_{prob}) + T(E^v_{2.3})_{prob} \\ &\quad + T(E^v_{2.4})_{prob} \end{aligned} \tag{8.1}$$

For instance ♯1, the expression of 8.1 is transformed as follows:

$$T(P_2)^1_{prob} = T(E^v_{2.2})^1_{opt} + T(E^v_{2.3})^1_{opt} + T(E^v_{2.4})^1_{opt}$$

Note that $T(P_2)^1_{prob}$ is used to designate the *Probable Execution Time* of P_2, when invoked. The symbol $\prod$ was introduced to indicate that the execution time is derived according to the aggregation pattern that connects the different elements that we defined. In addition, this symbol considers only `atomic` elements, which is the type for all the elements in the stock quoting process.

In addition, note that the entity *control delegation time* in the estimation of the *probable execution time* of a process is ignored because the JOpera tools provide no means to inquire about it.

Because no failure occurred when instance ♯1 was executed, we have:

$$T(E^v_{2.1})^1_{opt} = T(E^v_{2.1}, ws^1_{2.1})^1 \quad RP(E^v_{2.1})^1 + R(E^v_{2.1})^1 = 0$$
$$T(E^v_{2.2})^1_{opt} = T(E^v_{2.2}, ws^1_{2.2})^1 \quad RP(E^v_{2.2})^1 + R(E^v_{2.2})^1 = 0$$
$$T(E^v_{2.3})^1_{opt} = T(E^v_{2.3}, ws^1_{2.3})^1 \quad RP(E^v_{2.3})^1 + R(E^v_{2.3})^1 = 0$$
$$T(E^v_{2.4})^1_{opt} = T(E^v_{2.4}, ws^1_{2.4})^1 \quad RP(E^v_{2.4})^1 + R(E^v_{2.4})^1 = 0$$

This scenario considered only the case of process instances with no failure; therefore, in our model the expression that relates to the failure recovery time estimates are irrelevant.

SCENARIO 2. In this scenario, we considered the case of one of the instances in which a failure occurred. The execution retrial of the failed element was not possible because there were no other available WS to reattempt it. As a result, a backward recovery was necessary. In the following expression, we followed the case of instance ♯5 in which the WS $ws^1_{2.3}$, allocated to $E^v_{2.3}$, has failed. The expression of 8.1 is transformed as follows:

$$T(P_2)^5_{prob} = \max(T(E^v_{2.1})^5_{prob}, T(E^v_{2.2})^5_{prob}) + T(E^v_{2.3})^4_{prob} + T(E^v_{2.4})^5_{prob} \quad (8.2)$$

Because a failure occurred when instance ♯5 was executing the element $E^v_{2.3}$, we have:

$$T(E^v_{2.1})^5_{prob} = T(E^v_{2.1}, ws^1_{2.1})^5 + RP(E^v_{2.1})^5 + R(E^v_{2.1})^5$$
$$T(E^v_{2.2})^5_{prob} = T(E^v_{2.2}, ws^1_{2.2})^5 + RP(E^v_{2.2})^5 + R(E^v_{2.2})^5$$
$$T(E^v_{2.3})^5_{prob} = T(E^v_{2.3}, ws^1_{2.3})^5 + RP(E^v_{2.3})^5$$
$$T(E^v_{2.4})^5_{prob} = 0$$
$$R(E^v_{2.1})^5 = Back(E^v_{2.1})^5$$
$$R(E^v_{2.2})^5 = Back(E^v_{2.2})^5$$

8.2.4 Reliability Estimation and Analysis

From the results of the invocations of the quoting process (Figure8.13), we determined the *TSS*, *STS*, and *RT* of the different elements of the composition (see Figure 8.14).

The estimates of the *Reliability Tendency* (*RT*) of the different elements are shown in (Figure 8.14).

In determining these estimates, defining the different *State Reliability Contributions* (*SRC*) of each *Terminal State* was required. In this case study, we allocated as initial values for the *Terminal States* `Finished`, `Failed`, and `Unreachable` the *SRC* of $+1.0$, -1.0 and $+0.5$, respectively. Our motivation behind assuming such values is that, when a negative *SRC* value is assumed, the variation in the overall reliability of the estimate can be more important. Therefore, that a failure is taking place can be more readily highlighted by attracting the designer's attention to the element with the more critical reliability estimate. To realize this, we attached to the `Finished` state a more neutral value, because we are more interested in failures; we attached a negative value to the *SRC* of the `Failed` state to make its effects noticed very quickly. In addition, we attached to the `Unreachable` state a median value because in this case the element execution was about to start but it did not because its activation condition was not fired; therefore, it requires the designer's attention to check why such a situation occurred.

Typical interpretations of these results are:

- First, both $E^v_{2.3}$ and $E^v_{2.4}$ tend not to succeed in their executions in 9.1% of cases because of their own failures (i.e., in 9.1% of cases their executions terminate in the `Failed` state). For example, instance ♯5 and instance ♯4 failed because failures occurred, respectively, at $E^v_{2.3}$ ($ws^1_{2.3}$ failed to send back its response and a time-out occurred) and at $E^v_{2.4}$ (a network failure prohibited $ws^1_{2.4}$ receiving its input).
- Second, the elements $E^v_{2.3}$ and $E^v_{2.4}$ tend not to start their executions and to terminate in the `Unreachable` state in 36.4% and 45.5% of the total invocations, respectively. An element state is set to the `Unreachable` state when the condition associated with the start of its execution is evaluated as false. In such a case, its execution is skipped [62]. In the case of $E^v_{2.3}$ and $E^v_{2.4}$, their conditions were not fired because their predecessors failed (e.g., in instance ♯9, $E^v_{2.2}$ failed).
- Finally, elements $E^v_{2.1}$ and $E^v_{2.2}$ have a strong tendency to finish in the `Failed` state: up to 27.3% for $E^v_{2.1}$ and 36.4% for $E^v_{2.2}$. Their frequent failures cause overall composition failure. Therefore, the reasons behind the frequent failures of $E^v_{2.1}$ and $E^v_{2.2}$ need to be investigated. Moreover, other WS bearing the same functionalities as $E^v_{2.1}$ and $E^v_{2.2}$ need to be searched. Lastly, revising the CWS structure (i.e., order of elements, invocation conditions) has to be planned, if other candidate WS show no improvements in the quality of execution of the process.

8.3 Validation Results Discussion

In the introduction to this paper, we advocate that perfect awareness of inevitability of failures in the WS context and a failure-handling-devoted composite Web services modeling, execution, and analysis strategy are required to realize a greater gain in dependability. In this section, we have validated that claim. We have checked the applicability of our proposed ideas and shown that they are feasible and can be implemented using the available WS enabling technologies (e.g., WSDL, UDDI, and SOAP), to a limited extent. In our prototype, we only implemented part of the complete FENECIA approach features because a full-featured implementation is difficult to realize with the current state of WS technology, as described above. A full implementation requires a more mature WS technology, particularly regarding dynamic WS discovery and selection, solutions that consider the semantic and syntactic aspect of WS are needed.

	Terminal States Set (TSS)	State Tendency Set (STS)	Reliability Tendency (RT)
$(E^v_{2.1}, WS^1_{2.1})$	{(Finished,72.7%), (Failed,27.3%), (Unreachable,0%)}	{(Finished,72.7%)}	(72.7%*1+27.3%*(-1)+0%*0.5)/3=15.1%
$(E^v_{2.2}, WS^1_{2.2})$	{(Finished,63.6%), (Failed,36.4%), (Unreachable,0%)}	{(Finished,63.6%)}	9.1%
$(E^v_{2.3}, WS^1_{2.3})$	{(Finished,54.5%), (Failed,9.1%), (Unreachable,36.4%)}	{(Finished,54.5%)}	21.2%
$(E^v_{2.4}, WS^1_{2.4})$	{(Finished,45.5%), (Failed,9.1%), (Unreachable,45.5%)}	{(Finished,45.5%), (Unreachable,45.5%)}	19.7%
		Reliability tendency(RT(CWS_2))	16.3%

Fig. 8.14 Quoting process: TSS, STS, and RT

In particular, in this prototype implementation, we have shown that the different mechanisms defined by WS-SAGAS are feasible. Specifically, by describing a process in terms of elements and by removing any execution-related details, such as binding each element to only one WS, a higher dependability level can be achieved by realizing forward recovery. This cannot be said of other available WSC languages, notably BPEL where when a fault occurs at one statically bound WS; BPEL processes handle the fault by a compensation handler invoked to compensate for the faulty activity. Although BPEL adds some reliability support, declaring a process failed should nevertheless be the last resort and envisaging forward recovery with dynamic WS discovery and binding is more promising; otherwise, the WS architecture offering the possibility of switching easily from one provider to another is useless.

In our prototype description, we have also shown how the WS-SAGAS process definitions and, in particular, the way each DR encapsulates information about an element can be used as the process execution runs for dynamic WS discovery and mapping. The description of an element and its operation provided with its parameters is used to create a query that is sent to WS registries for searching WS that eventually meet the description. We emphasize that our described method for element-WS matchmaking is intentionally simplified because we consider WS discovery and selection issues beyond the scope of this paper. Assessing the similarity of WS to achieve the best match is an active area of research, so we may apply one of the available proposals, such as the keyword-based methods and ontologies and reasoning algorithm enriched methods. Therefore, the process execution can transparently resume without interruption and, even when a dynamically mapped WS fails, instead of stopping the overall execution, as in BPEL, a forward recovery can be transparently attempted by automatically allocating another WS. In addition, equivalently to BPEL, our model supports backward recovery because a compensating element is provided to each element.

In addition, we have also shown the broad scope of the applicability of our QoS model and that our failure-aware QoS analysis approach, with the state incorporation, can provide step-by-step information about the execution progress, which can help to track the location of failures and explain the reasons for failures.

In many of the available WSC languages, exemplified by BPEL, mapping between WS and partners is set when a process is invoked, and this mapping is fixed for all the executions. As the process runs, there is no means of knowing the execution progress, because WS are generally stateless and BPEL provides only a correlation-based stateful interaction that only allows identifying instances. Another mechanism is required to identify the progress of the interacting parts as the process runs, and to derive the process instances progress. This is exactly the crucial role of the state concept introduced in our proposal. Tracking the execution progress by keeping a log of all the CEPs, which are updated on every change in any of the element states, provides a step-by-step execution progress of all the process instances that we can analyze to investigate failures' reasons or locations. We have shown how this can be done when we applied our state-guided failure analysis approach to data collected using Jopera.

Finally, the case study allowed us to show that in estimating the execution time, considering all the possible execution situations and building on the *state* concept can help designers to acquire detailed data about the failure location and causes more easily, without having to use any complex modeling formalisms. Moreover, the data derived from the execution history (i.e., state tendency sets, terminal states set) are more practical and straightforward because no simulation systems are required for analysis. On the basis of such data, system designers can more readily locate error-prone component(s), reasons for failure can be more easily investigated, and eventually, the process overall structure can be altered to improve performance, if required. However, in our proposal, to reach its full potential, we need to use a more robust real-time monitoring tool that can, besides measuring and collecting the total response time of a process invocation, distinguish between a faulty and a successful invocation, measure separately the SOAP messages roundtrip time and the WS execution time, and measure the control delegation time. Several WS monitoring tools are already available but, to the best of our knowledge, they are only for elementary WS or for statically composed WS; the case of dynamically executed composite WS has not yet been considered.

9 Related Work and Discussion

9.1 WSC Approaches

A number of alternative approaches have been suggested by several authors to aggregate individual WS to produce a new CWS, enabled even to encapsulate the underpinning logic of complex business processes. However, large parts of the available solutions are oriented toward comparing the semantics of the interleaved services and checking their ports' compatibility.

The most important feature that distinguishes our approach from others is that we considered the dependability issue in all the different stages of the WSC process, that is, from the specification, to the execution, to the QoS assessment.

The WSC platform StarWSCoP (Star Web Services Composition Platform) [65] is very closely related to our work and it follows a similar approach to ours because it focuses on dynamic composition. It provides a service execution information library that stores trace information of CWS execution; in our approach, this library is equivalent to the history that collects the observed execution progress of the CWS instances (i.e., the copies of *CEP*).

Another similarity with our approach is that in StarWSCoP, the authors developed the notion of a wrapper for each WS, which is very similar to the engine notion, because it is also used to act as a proxy of the WS; others communicate with the wrapper instead of the WS. The wrapper initiates, freezes, and continues the WS according to the requests sent by the requester. However, the wrapper does not have any predefined mechanism to handle potential failures such as those we defined in our approach, where each engine, beside conversing with the WS it wraps, also communicates with different engines in a peer-to-peer fashion to progress the execution and to handle failures, which is completely absent in the StarWSCoP approach. Furthermore, each wrapper implements different managers to deal with the security, transport, and data type mismatch issues. We recognize that security is a very important dependability attribute, especially in the context of WS. However, because the techniques for security assessment are still rudimentary in the WS architecture, security is not addressed in this paper. Considering the execution aspect, our approach is more scalable than StarWSCoP because execution control in StarWSCoP is allocated to a centralized engine whereas in our approach it is distributed among different engines to avoid the possibility of bottlenecks and of having a single point of failure.

For QoS estimation and analysis, the StarWSCoP approach estimates real-time QoS metrics of the CWS by extending WSDL to support QoS metrics, such as cost, time, and reliability. The defined QoS metrics are very simplistic compared to ours: they do not consider the repercussions of failures and the authors do not state how reliability is estimated. In addition, the real-time QoS estimations are used to check if a particular CWS fits the user's predefined QoS requirements, which means that StarWSCoP targets user satisfaction. However, our approach is oriented more toward allowing the system designers to assess the system quality, analyze it, and eventually produce some improvements.

Similarly, we also consider two other approaches where the CWS is created dynamically by describing the functionalities of interest that components should have without referencing any specific WS. The first is eFlow, a platform developed by HP [7,8]; the second approach is SELF-SERV, a framework developed by the University of New South Wales [41,10,9]. In the eFlow platform, the definition of a service node encloses a service selection rule written in a particular query language. When invoking the service node, the rule is executed to select a specific service. Concerning SELF-SERV, it exploits the concept of service community, a container of alternative services. At runtime, a community delegates any requests it receives to one of its current members. The eFlow platform contrasts with our approach because it works with a centralized scheduler. As with starWSCoP, it may suffer from scalability problems and no QoS model is explicitly supported. On the other hand, SELF-SERV uses an approach similar to ours: a distributed execution system where coordinators (i.e., software components hosted by WS providers) may control a set of WS, rather than only one. Although the SELF-SERV strategy avoids having a single point of failure, to execute a CWS the different coordinators need to manage routing tables, statically generated from the coordinators' precondition and postcondition states transition tables; a large amount of data needs to be exchanged among coordinators. Doing so may easily provoke failures of the coordinators because of bottlenecks. To the best of our knowledge, the SELF-SERV strategy does not provide any mechanisms for handling similar situations. On the other hand, in our approach, the CEP concept, equivalent to the coordinators' routing tables in SELF-SERV, allows a dynamic decision of the execution control delegation of the engines based on the CWS different components execution states; however, the advantage is that a minimal amount of data is exchanged, compared with SELF-SERV. Furthermore, an engine is allocated only if it is in good condition. Moreover, execution retrial and compensation mechanisms are provided in case of failure, which cannot be said of SELF-SERV. Note that the notion of community in SELF-SERV differs greatly from the notion of CEL in our approach because there is no defined policy to handle the case where one or more component services that form a community fail. Therefore, considering extending the composition model to integrate transactional semantics, as in our approach, is very interesting for SELF-SERV. However, an unresolved issue remains and needs to be addressed for our approach/SELF-SERV: how to decide on the size of the community/CEL to increase the chances of successful execution. Here, the idea of using the history of past executions of a CWS can be used to dynamically optimize an ongoing execution—according to a given set of parameters, such as time, price, and QoS—and to decide the suitable number of CELs available in view of the success percentage of the different WS invoked in the different CWS instances.

One of the most profound features that is of great importance for designing and developing dependable composite services is transaction support. WS are well known for being autonomous, heterogeneous units, where each WS provider has its own management policies; such characteristics make implementing CWS with a transaction support more difficult, but essential. Very few proposals contain transaction support in their composition. [39,40] introduces a framework called WebTransact, which provides the necessary infrastructure to build reliable CWS.

WebTransact is composed of a multilayered framework. It uses WSDL to describe the WS functionalities and adds a Web Services Transaction Language (WSTL) on top of WSDL, enhancing it with functionalities facilitating composite WS by describing transaction support for a WS. As in our approach, WebTransact defines different types of transaction behavior. However, it supports compensation and retriability behaviors and introduces virtual-compensatable behavior for operations whose underlying system supports 2PC and pivot behavior for the operations, which are neither compensatable nor retriable.

However, the main differences between our approach and WebTransact are, first, the WS are statically integrated in WebTransact by the developer who plays the role of WS integrator. However, this is not a flexible method of WS integration. Second, the WebTransact framework is mainly for integrating WS that have (and expose) their own local transaction support; however, this condition is not always verifiable because not all WS have transaction support and presumably, if they do, not all are compliant with each other, or are limited only to the above suggested transaction support of WebTransact. This is what made us consider completely ignoring transaction support that the different WS may provide and to decide to offer/append at a higher level the same transaction support at the composition WS level instead. Currently in our approach, WS-SAGAS supports compensatable, noncompensatable, retriable, vital, and nonvital behaviors. Finally, our approach can complement the WebTransact framework because our QoS model can be very important in auditing and analyzing the WS execution to improve the quality and efficiency of the mediator service composition given that QoS assessment is not yet addressed in Webtransact.

In [66], an approach to selecting services based on their semantics as well as their quality, as judged by users, is proposed. To this end, a query language based on DAML that accommodates several essential query and manipulation templates is developed. The users'/providers' estimations of the QoS may be incorrect and/or biased by the users' subjectivity. In our approach, we do not rely on the users'/providers' QoS rating; instead, designers observe the CWS execution and collect the execution results in a history to use later as a basis to estimate the QoS properties.

In [67,58], the authors introduce a QoS-aware middleware for CWS. They focus on a dynamic and quality-driven approach to select component services for a composite service. Multiple QoS criteria, such as price, execution time, and reliability, are considered. They propose a global planning approach to optimize the overall QoS using linear programming techniques. Their approach is effective with respect to reaching QoS optimality. However, their complex Workflow patterns, such as using branching and frequent loop iterations, seems to make their approach less efficient and increasingly complex for business processes. Furthermore, potential failure repercussions on the global QoS have not been considered. Moreover, reliability is mapped directly to the reliability of each WS individual. Reliability is defined as the probability that a request to a particular WS receives a correct response within a maximum expected time frame. This method of characterizing the reliability is not extendable to dynamically assembled CWS.

Similarly to our approach, the authors of [68,57,69,70] have proposed building new CWS that are QoS-optimized and have either defined their own QoS models or been inspired by other models. However, all these approaches are only applicable for statically aggregated CWS. In addition, the authors of [57,59,69,70] have investigated different QoS dimensions, such as time, cost, reliability, and fidelity. However, they have not considered how the different states and effects of failure cause the QoS estimates vary. To characterize the reliability dimension, their proposed models are derived from a more general work [59], in which the discrete-time stable reliability model proposed in [71] is followed to describe the reliability of tasks in the Workflow context:

$$R(t) = 1 - (\text{system failure rate} + \text{process failure rate}) \ .$$

This equation is only applicable for static CWS as it only gives a global idea of the reliability estimates of a component. Our approach for reliability estimation goes further because it obtains more detailed estimates with the possibility of knowing what component(s) was/were behind the considerable variation in the overall reliability estimates and the reason (i.e., failure).

9.2 Standards and Commercial Platforms

WS are becoming an important part of mainstream IT. Every day, it seems, a new acronym is introduced and added to the mass of acronyms ranging from SOAP to UDDI to WSDL. Moreover, ongoing massive standardization efforts seek to enable CWS; these include, among others, business process modeling languages such as WSCI, WSFL (Web Services Flow Language), and, most recently, BPEL4WS [4], which have been developed to model CWS. Of these, only BPEL4WS considers failure handling but it offers only limited support because it introduces fault handlers to specify the actions to be taken when a WS execution fails. However, these fault handlers are defined in a way similar to the exception-handling techniques exploited in programming languages. Moreover, the handlers are dedicated to trying to recover from the effects of the failed service but they do not attempt to investigate the causes behind the failure, as we do in our approach.

In addition, other existing standards, such as BTP [48], the WS-Transaction [25] proposed by IBM, and WS-TXM (from WS-CAF framework) [26] by Sun define models to support transactions between loosely coupled systems in the WS context. They define models for centralized and peer-to-peer transactions, which support a two-phase coordination of WS. These standards build on extended transactional models to specify how different WS are coordinated. The different entities have to agree a priori on the transaction model. Consequently, they inherit the advanced transactional models' lack of functionality and performance when used in applications that involve dynamic composition of heterogeneous services in a peer-to-peer context [72]. Hierarchical QoS Markup Language (HQML), Web Ontology Language (OWL-S), and Web Service Level Agreement language (WSLA) are examples of specifications that have addressed the need for a QoS model.

The common point of these specifications is that they describe the QoS of WS. For example, DAML-S has included constructs that specify several QoS parameters, namely, the quality rating and the degree of quality. However, these specifications have not supplied any precise characterization of the different parameters and they are only suitable for WS and not for compositions.

Finally, examples of commercial platforms that deal with WS automation include Microsoft's .NET and BizTalk tools and IBM's WebSphere. These applications provide support for SOAP, WSDL, and UDDI connectivity. However, to the best of our knowledge, they provide little or no support for CWS.

9.3 Conventional Composite Systems

Making several entities work in tandem to reach a common goal is not a new challenge in itself, because it has been widely addressed for decades in several areas, including Workflow management systems, software engineering, and artificial intelligence. Many argue that when considering CWS, it is important to take into account, and use experience and knowledge from, these closely related areas [73, 43], because the main difference is that composition in WS architecture chiefly aims at taking XML-based standards and the Internet as starting points to reach the same goal. In the different parts of our approach, we promote the same idea because we were inspired by several works in related areas, specifically in the area of Workflow technology and software engineering.

In defining the different aggregation patterns for the CWS specification model, we chose to build on Workflow patterns to define the different WS-SAGAS aggregation patterns because the typical control flow dependencies encountered in Workflow modeling arguably apply as well in the context of CWS, because the situations they capture are noticeably similar. In [43], the authors showed that the Workflow patterns apply to existing CWS languages such as BPEL4WS and BPML. Our approach builds on the state concept that was used well in the context of software engineering to define QoS models, where many mathematical techniques have been developed. The models that are closely related to our approach are the structural models of reliability [74] and the Markov reward models [75], which form the basis of all performability models. In the former, a state diagram that depicts the system behavior is used.

Based on Markov chain properties, the transition between states is assumed to be a Markov process. This means that the components to be executed in the next state depend only on the components of the current state and the components of the next state do not depend on the history of the current state. In the latter, the system is assumed to be modeled as a Markov process with a finite state space, and a reward rate (performance measure) is associated with each state.

Our approach complements these models because we use the state concept to define in the same way the behavior of the different components in terms of transition between different states; we augment this by making the state concept play an important role in enhancing failure information, defining the QoS estimates, and analyzing the QoS estimates. On the other hand, our approach differs in its simplicity from these proposals and from other techniques proposed in [76, 77], which are supported by underlying modeling formalisms (e.g., block diagrams, Markov chains, Petri-nets, logics, etc.), because the acquired estimates are easily analyzed, which is not the case of the techniques above, widely known for requiring considerable expertise and effort. Very often, the designers are not eager to build such models because of their inherent complexity. Finally, the models obtained are not straightforward to interpret so further simulations have to be performed.

10 Conclusions

In this paper, we introduce FENECIA, our framework for CWS specification, execution, and QoS assessment. Our approach puts forward the view that WS/CWS failures are not exceptional situations, as often claimed, but takes a radically different view by accepting that failures are inevitable for any WS/CWS. In addition, our approach emphasizes that when earlier failures are taken into consideration, by defining in advance proper failure-handling mechanisms, there are greater chances of seeing a CWS perform with greater dependability. To achieve this vision, our work's main contributions are summarized below.

First is the construction of WS-SAGAS, which provides the framework required to build a transaction model specifically tailored to fit the characteristics of the WS architecture, thereby allowing movement away from the constraints imposed by the traditional transaction model. WS-SAGAS inherited several interesting features from previously proposed transaction models, specifically, arbitrary nesting, relaxed ACID properties, state, vitality degree, forward and backward recovery, and compensation.

We demonstrate how these concepts, which were adapted from conventional composite systems, need to become part of the WS architecture pillars to provide major contributions in dependability enhancement.

We also show how our model provided a powerful construct for extending other approaches to support WSC more expressively, with an increasing level of flexibility and dependability, by defining a textual notation that is as free as possible from programming constructs and as explicit as possible. This would allow it to be easily understood and updated, which cannot be said of the syntax of other existing proposals, which are heavily based on XML. Furthermore, our textual notation that describes a CWS in terms of definition rules (DR), composability rules (CR), and ordering rules (OR) is particularly useful for us to define our transaction model operational semantics and correct executions. Because we consider a peer-to-peer execution model, the use of strict serializability, adopted in traditional transaction models, is inadequate. The description of a process in term of *DR*, *CR*, and in particular, *OR*, contributes partly to avoiding inconsistencies, because the different *OR* allow definition of the correct control flow in a process. To ensure the semantics of each element are respected when it is executed, particularly the nesting, transactional behavior, and vitality degree, we build on the state concept and we define several forms of dependency that must hold between the different elements combined in the same pattern, the same WS-SAGAS, and in the same process; we call these *intrapattern dependencies*, *intra-WS-SAGAS dependencies*, and *intraprocess dependencies*.

We have attached a graphical syntax to our model to exploit the perceptual capabilities of designers by allowing them to capture the models at different levels of detail, whereas other solutions either define no graphical notation or advocate the use of state-charts for ease of use, although they may not allow expression of all their models' semantics.

From WS-SAGAS for WSC dependable specification as a hierarchy of recursively nested transactions comes our second contribution toward defining an execution environment that supports the abstract concepts suggested by WS-SAGAS, which we named THROWS architecture. The execution of WSC, depicted as WS-SAGAS, is made possible by the confluence of several novel ideas. First, most existing WSC systems only support the integration of WS in a centralized model, consisting of dedicated centralized engine(s). They have totally ignored the inherent nature of the WS environment where interaction follows a peer-to-peer model and where each peer WS owner provides a set of services that comprise CWS. We take a radical approach and propose an architecture where the execution control is hierarchically delegated to distributed engines discovered dynamically. Executing the CWS in a distributed fashion allows us to avoid having a single point of failure and to split the messages that the central authority is required to manage among the distributed engines.

In addition, because WS are in essence loosely coupled, integrating them into a CWS makes the system reliability and availability a critical issue. To deal with this issue, we propose generating the CEL dynamically, where a list of dynamically discovered WS-engine couples is ranked. Moreover, the CEL concept allows the execution retrial with alternative candidates. Because CEL are dynamically generated, engine sequences of invocation cannot be known beforehand. Here, we propose the CEP concept so that the execution control delegation between engines can be performed by keeping track of the execution progress.

We also introduce a model to assess the QoS of CWS. In our model, rather than relying on the QoS information advertised by the different WS providers (which may be not up to date), we estimate the QoS properties on the basis of CWS execution observations, which are collected in a history that consists of the different copies of *CEP* and its different updated copies. Second, we confer paramount importance to the failure repercussions on the CWS performances; in fact, not only were correct execution instances examined to estimate the QoS and later analyze it, but also our model was oriented toward considering the system in all of its possible states (e.g., correct, faulty, recoverable, executions). By doing so, we intend to make our model capable of reflecting the real state of the typical case of CWS, with their inherent tendency to fail rather easily compared with others. Third, we use the concept of element state, initially introduced in WS-SAGAS, so that the more error-prone elements can be more readily located. Finally, our model does not use any complex modeling techniques, thus making it directly usable without requiring a difficult learning curve.

Our method illustrates how conferring paramount importance to failure repercussions on the CWS performances can turn the observed failures throughout a process execution cycle from a difficulty to a benefit. We demonstrate in our case study how the history of execution of faulty process instances can serve as solid basis for analyzing the robustness of fault-tolerance mechanisms by tracking failures to find the most error-prone element in a process. We also show how such observations are used to restructure the process definition to achieve better quality of execution and how our flexible process definition in terms of DR, CR, and OR support such a method.

We demonstrate that the abstract concepts and artifacts defined by FENECIA can be implemented to some extent in a prototype in the context of a research project. While our prototype implementation suffers from technological limitations, it does demonstrate that our proposal is within the realm of feasibility. The possibility of implementing a fully fledged implementation of this work will depend greatly on the evolution speed of the service industry and research.

The FENECIA approach proposal allows us to realize that basic concepts that exist in conventional composite systems, namely the element state and, more generally, the component behavior, need to be made available for WS as they can assist greatly in obtaining information about the execution and in adding improvements. Moreover, in conventional composite systems, where the same components are connected, only static composition strategies were available.

However, in the WS context, throughout the different parts of our approach, we show that such a solution is not viable, and a dynamic composition strategy is far more preferable. However, to realize fully a dynamic composition strategy, much remains to be done in the WS architecture because it still suffers significantly from being heavily based on the Internet.

In the case of some business processes where failures are not permissible (e.g., banking), effective realization of the FENECIA vision, besides making the failure recovery mechanism possible, requires proper failure avoidance mechanisms, which may constitute an interesting extension to our present work. In addition, at present the CWS execution in FENECIA is done independently by different engines, where the engines are volatile (i.e., on each CWS invocation, new CEL are built to avoid using WS that are no longer available or obsolete). Making the different engines nonvolatile and assigning QoS attributes to the different engines to estimate their performance can lead to a more optimized execution, because the more reliable engines are selected. Later, a more elaborate model of collaboration between the different engines can be developed. Finally, because there is a wide range of toolkits supporting WS development, another interesting research direction will be finalizing the implemented configuration of WS-SAGAS for THROWS, experimentally measuring its performance, and comparing it with others.

Acknowledgements Part of this research was supported by CREST of JST (Japan Science and Technology Agency), a Grant-in-Aid for Scientific Research on Priority Areas from MEXT of the Japanese Government (#16016232 and #18049026), and the 21st Century COE Program Framework for Systematization and Application of Large-scale Knowledge Resources.

References

1. W3C. Web services description language (wsdl). http://www.w3.org/TR/wsdl, 2005.
2. W3C. Simple object access protocol (soap). http://www.w3.org/TR/soap, 2005.
3. W3C. Universal description, discovery, and integration (uddi). http://www.uddi.org, 2005.
4. IBM, BEA Systems, Microsoft, SAP AG, and Siebel Systems. Bpel4ws business process execution language for web services, 2005.
5. IBM. The emerging technologies toolkit (ettk), 2005.
6. Microsoft. Microsoft web services strategy.net. http://www.microsoft.com/net/, 2005.
7. Fabio Casati, Ski Ilnicki, Li-Jie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. eflow: A platform for developing and managing composite e-services. In *AIWORC '00: Proceedings of the Academia/Industry Working Conference on Research Challenges*, page 341. IEEE Computer Society, 2000.
8. Fabio Casati, Ski Ilnicki, Li jie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in eflow. In B. Wangler and L. Bergman, editors, *CAiSE '00: the 12th international Conference on Advanced information Systems Engineering*, volume 1789 of *LNCS*, pages 13–31, London, June 05 - 09 2000. Springer-Verlag.
9. Quan Z. Sheng, Boualem Benatallah, Marlon Dumas, and Eileen Oi-Yan Mak. Self-serv: A platform for rapid composition of web services in a peer-to-peer environment. In *VLDB*, pages 1051–1054, 2002.
10. Boualem Benatallah, Quan Z. Sheng, and Marlon Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1):40–48, Jan. 2003.
11. Dan Wu, Bijan Parsia, Evren Sirin, James A. Hendler, and Dana S. Nau. Automating daml-s web services composition using shop2. In *The Semantic Web - ISWC 2003, Second International Semantic Web Conference, Sanibel Island, FL, USA*, volume 2870 of *Lecture Notes in Computer Science*, pages 195–210. Springer, 2003.
12. Nikola Milanovic and Miroslaw Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004.
13. N. J. Davies, D. Fensel, and M. Richardson. The future of web services. *BT Technology Journal*, 22:118–130, Jan. 2004.
14. Ahmed K. Elmagarmid. *Database transaction models for advanced applications*. Morgan Kaufmann, San Mateo, California, 1992.
15. Neila Ben Lakhal, Takashi Kobayashi, and Haruo Yokota. Ws-sagas:a transaction model for reliable web services composition specification and execution. *DBSJ letters*, 2(2):7–20, Oct. 2003.
16. Neila Ben Lakhal, Takashi Kobayashi, and Haruo Yokota. Distributed architecture for reliable execution of web services. Technical Report DBWS2003 2B, IEICE, 2003.
17. Neila Ben Lakhal, Takashi Kobayashi, and Haruo Yokota. Reliability and performance estimation for enriched ws-sagas. In *WIRI '05: Proceedings of the International Workshop on Challenges in Web Information Retrieval and Integration, In conjunction with ICDE2005*, pages 54–63, Tokyo, Japan, Apr. 2005. IEEE Computer Society.
18. Neila Ben lakhal, Takashi Kobayashi, and Haruo Yokota. Dependability and flexibility centered approach for composite web services modeling. In *14th International Conference on Cooperative Information Systems (CoopIS2006)*, volume 4275(OTM2006) of *LNCS*, pages 163–182, Montpellier, France, Nov. 2006.
19. Neila Ben Lakhal, Takashi Kobayashi, and Haruo Yokota. Throws: An architecture for highly available distributed execution of web services compositions. In *IEEE 14th International Workshop on Research Issues on Data Engineering: Web Services for E-Commerce and E-Government Applications (RIDE'04)*, pages 103–110, Boston, USA, March 2004. IEEE.
20. Neila Ben Lakhal, Takashi Kobayashi, and Haruo Yokota. A simulation system of throws architecture with ws-sagas transaction model. *DBSJ Letters*, 3(1):89–92, June 2004.
21. Neila Ben Lakhal, Takashi Kobayashi, and Haruo Yokota. A failure-aware model for estimating and analyzing the efficiency of web services compositions. In *PRDC '05: Proceedings of the 11th Pacific Rim International Symposium on Dependable Computing (PRDC'05)*, pages 114–124, Washington, DC, USA, 2005. IEEE Computer Society.
22. Neila Ben Lakhal. *A framework for modeling, executing, and analyzing dependable transactional Web services compositions*. Phd.thesis, Tokyo Institute of Technology, Tokyo, Japan, 2007.
23. A. Gorbenko, V. Kharchenko, P. Popov, A. Romanovsky, and A. Boyarchuk. Development of dependable web services out of undependable web components. Technical Report 863, University of Newcastle upon Tyne, School of Computing Science, Oct 2004.
24. Jean-Claude Laprie and Brian Randell. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable Secure Computing*, 1(1):11–33, 2004. Fellow-Algirdas Avizienis and Senior Member-Carl Landwehr.
25. F. Cabrera and et al. Specification: Web services transaction (ws-transaction). http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/, 2002.
26. Oracle Fujitsu, IONA and Arjuna Technologies Sun. Web services composite application framework(ws-caf). http://www.arjuna.com/standards/ws-caf/, 2003.
27. Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD Conference*, pages 249–259, 1987.

28. W.M.P. van der Aalst, A.H.M.ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
29. A. Sheth, K. Kochut, and J. Miller. Meteor project page at large scale distributed information systems (lsdis)laboratory. http://lsdis.cs.uga.edu/proj/meteor/meteor.html.
30. Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.
31. W. Du and Ahmed Elmagarmid. Quasi serializability: a correctness criterion for global concurrency control in interbase. In *VLDB '89: Proceedings of the 15th international conference on Very large data bases*, pages 347–355, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.
32. Krithi Ramamritham and Panos K. Chrysanthis. A taxonomy of correctness criteria in database applications. *The VLDB Journal*, 5(1):085–097, 1996.
33. Ahmed K. Elmagarmid, Yungho Leu, Witold Litwin, and Marek Rusinkiewicz. A multidatabase transaction model for interbase. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 507–518, San Francisco, CA, USA, August 13-16 1990. Morgan Kaufmann Publishers Inc.
34. Sami Bhiri, Olivier Perrin, and Claude Godart. Ensuring required failure atomicity of composite web services. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 138–147, New York, NY, USA, 2005. ACM Press.
35. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
36. J.E.B. Moss. *Nested transactions: an approach to reliable distributed computing*. Cambridge, Massachusetts, 1985.
37. Hector Garcia-Molina, Dieter Gawlick, Johannes Klein, Karl Kleissner, and Kenneth Salem. Modeling long-running activities as nested sagas. *Data Engineering Bulletin*, 14(1):14–18, March 1991.
38. Mansoor Ansari, Linda Ness, Marek Rusinkiewicz, and Amit P. Sheth. Using flexible transactions to support multi-system telecommunication applications. In *VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases*, pages 65–76, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.
39. Paulo F. Pires, Mario R. F. Benevides, and Marta Mattoso. Mediating heterogeneous web services. In *2003 Symposium on Applications and the Internet (SAINT 2003)*, pages 344–347, 27-31 January 2003 - Orlando, FL, USA.
40. Paulo F. Pires, Mario R. F. Benevides, and Marta Mattoso. *Web, Web-Services, and Database Systems*, volume 2593 of *Lecture Notes in Computer Science*, chapter Building Reliable Web Services Compositions, pages 59–72. Springer, 2003.
41. Boualem Benatallah, Marlon Dumas, and Quan Z. Sheng. Facilitating the rapid development and scalable orchestration of composite web services. *Distributed and Parallel Databases*, 17(1):5–37, Jan. 2005.
42. Gwen Salan, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, page 43, Washington, DC, USA, 2004. IEEE Computer Society.
43. W.M.P.V.D. Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, 18(1):72–76, March 2003.
44. J. A. Bergstra, A. Ponse, and S. A. Smolka. *Handbook of Process Algebra*. Elsevier, 2001.
45. R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
46. C.A.R.Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
47. Michael J. Butler, C. A. R. Hoare, and Carla Ferreira. A trace semantics for long-running transactions. In *25 Years Communicating Sequential Processes*, pages 133–150, 2004.
48. OASIS Committee. Business transaction protocol version 1.0, 2004.
49. F. Cabrera and et al. Web services coordination. http://www.ibm.com/developerworks/library/ws-coor/, 2002.
50. AT van Halteren. *Towards an adaptable QoS aware middleware for distributed objects*. Phd.thesis, University of Twente, Enschede, the Netherlands, The Netherlands, 2002.
51. ITU/ISO. Open distributed processing reference model, part 2: Foundations, international standard. 10746-2 ITU-T Recommendation X.902, 1995.
52. Christoph Schuler, Roger Weber, Heiko Schuldt, and Hans Jorge Schek. Peer-to-peer process execution with osiris. In Springer, editor, *International Conference on Service-Oriented Computing*, volume 2910 of *LNCS*, pages 483–498, Italy, December 2003.
53. Ulrike Greiner and Erhard Rahm. Quality-oriented handling of exceptions in web-service-based cooperative processes. In *Proceedings of the GI-/GMDS Workshop on Enterprise Application Integration (EAI-04), Oldenburg, Germany, February 12-13, 2004*, volume 93 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.
54. Markus Keidl, Stefan Seltzsam, and Alfons Kemper. Reliable web service execution and deployment in dynamic environments. In *Technologies for E-Services the fourth International Workshop (TES 2003)*, volume 2819 of *Lecture Notes in Computer Science*, pages 104–118. Springer, September 2003.
55. Mark H. Burstein, Jerry R. Hobbs, Ora Lassila, David L. Martin, Drew V. McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry R. Payne, and Katia P. Sycara. Daml-s: Web service description for the semantic web. In *First International Semantic Web Conference, (ISWC 02)*, volume 2342 of *Lecture Notes in Computer Science*, pages 348–363. Springer, 2002.
56. Meyer B. Applying design by contract. *IEEE Computer (Special Issue on Inheritance and Classification)*, 25(10):40–52, October 1992.
57. S. Chadrasekaran, J.A. Miller, G. Silver, I.B. Arpinar, and A. Sheth. Composition, performance analysis and simulation of web services. *Electronic Markets: The International Journal of Electronic Commerce Business Media*, 2003.
58. Liangzhao Zeng, Boualem Benatallah, Marlon Dumas, Jayant Kalagnanam, and Quan Z. Sheng. Quality driven web services composition. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 411–421, New York, NY, USA, 2003. ACM Press.
59. J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut. Quality of service for workflows and web service processes. *Journal of Web Semantics*, 2004.
60. J. Cardoso and A. Sheth. Semantic e-workflow composition. *Journal of Intelligent Information Systems*, 2003.
61. Cesare Pautasso, Thomas Heinis, and Gustavo Alonso. Autonomic execution of web service compositions. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, pages 435–442. IEEE Computer Society, 2005.
62. C.Pautasso. *A Flexible System for Visual Service Composition*. PhD thesis, ETH, July 2004.
63. SUN. Java web services developer pack v1.2 (jwsdp). http://java.sun.com/webservices/, 2003.
64. xmethods. http://www.xmethods.net, 2004.
65. H. Sun, X. Wang, B. Zhou, and P. Zou. *Research and Implementation of Dynamic Web Services Composition*, volume 2834, pages 457 – 466. Springer LNCS, 2003.
66. A. Soydan Bilgin and Munindar P. Singh. A daml-based repository for qos-aware semantic web service selection. In *ICWS*, pages 368–375, 2004.
67. Zeng L.and Benatallah B., Ngu A., Dumas M., Kalagnanam J., and Chang H. Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311 – 327, May 2004.
68. M.C. Gronmo, R.and Jaeger. *LNCS*, volume 3543, chapter Model-Driven Methodology for Building QoS-Optimised Web Service Compositions, pages 68 – 82. Springer, Jan. 2005.
69. Amit P. Sheth, Jorge Cardoso, John A. Miller, Krzysztof J. Kochut, and M. Kang. Service-oriented middleware. In *Proceedings of*

The sixth World Multiconference on Systemics Cybernetics and Informatics(Invited Session on Web Services and Grid Computing), volume 8, Orlando, FL, 2002.

70. J.Cardoso. *Quality of Service and Semantic Composition of Workflows*. Ph.d. dissertation, Dep. of Computer Science, University of Georgia, Athens, GA., 2002.
71. E.C.Nelson. A statistical basis for software reliability assessment. Technical report, TRW Systems Report, March 1973.
72. Nektarios Gioldasis and Stavros Christodoulakis. Utml: Unified transaction modeling language. *In third International Conference on Web Information Systems Engineering (WISE02)*, 0:115, 2002.
73. Wil M.P. van der Aalst, Marlon Dumas, and Arthur H.M. ter Hofstede. Web service composition languages: Old wine in new bottles? *euromicro Conference*, 00:298– 305, Sept. 2003.
74. R.C. Cheung. A user-oriented software reliability model. *IEEE Transactions On Software Engineering*, 6(2):118, March 1980.
75. M.A.Qureshi and W.H.Sanders. Reward model solution methods with impulse and rate rewards:an algorithm and numerical results. *Performance evaluation*, 1994.
76. H. Kobayashi. *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology*. Addison-Wesley, 1978.
77. J-C. Laprie. Dependable computing and faut tolerance: concepts and terminology. In *Proc. of the* 15th *int. Sym. on Fault-tolerant Computing (FTCS-15)*, pages 2–11, 1985.