AN EVENT-B DRIVEN APPROACH FOR ENSURING RELIABLE AND FLEXIBLE SERVICE COMPOSITION

Imed Abbassi, Mohamed Graiet, Lazhar Hamel and Zied Jaoua
ENIT, University of Tunis El Manar
ISIMM, Monastir University
abbassi_imed@ymail.com, {mohamed.graiet1, lazhar.hamel}@gmail.com, jaouazied2002@yahoo.fr

Abstract
In context of pervasive environment, the availability and reliability of Web services cannot be always guaranteed due to the dynamic and uncertain nature of these environments. Therefore, ensuring reliable and flexible service composition is one of the most difficult tasks. To overcome this problem, we propose an Event-B driven approach to easily design reliable and flexible Services compositions. We introduce a new paradigm, called dynamic transactional pattern. This new paradigm is a convergence concept of dynamic workflow patterns and advanced transactional model. It combines dynamic Control-Flow flexibility and transactional processing reliability. A composite Service is defined as a set of dynamic transactional patterns instances properly connected together. We show how the transactional behavior of a composite Service is automatically defined in an efficient way. In this work, we combine the modeling and verification activities related to formal development process supported by Event-B. The verification activity is based on theorems proving and model-checker. Afterwards, the model is validated using ProB animator in order to ensure that it behaves as expected.

Keywords: Web service; Web services composition; Transactional pattern; Dynamic Control-Flow pattern; Event-B; Proof; ProB

1. INTRODUCTION

Web services are modular and self-describing software applications that is advertised, located, and invoked across the Internet connection (Snell, 2009). Providing new services by combining on the fly existing ones is a major ongoing idea in pervasive computing. The Web service composition consists of creating and providing high-level functionalities from available services in order to meet the user requirements (Mustafa, 2009).

In context of pervasive environment, the availability and reliability of Web services cannot be always guaranteed due to the dynamic and uncertain nature of these environments as well as the heterogeneity of networks and services (Gilbert, 2013). Moreover, the state of these environments changes frequently for several reasons: services joining and leaving environments at run-time, new service partners with better quality are detected, services in use fail or their quality decline, change in user needs, and loss of services due to a network problem. Therefore, ensuring reliable and flexible Web service composition remains one of the most difficult tasks. A composition is reliable if its execution is correct. An execution is correct if it is successfully committed or it fails. By flexible composition, we mean a composition automatically reconfigured not only at design step, but also at runtime. Various existing studies are proposed trying to resolve this problem. In (Bhiri, 2005), authors propose an approach that combines the business process adequacy of workflow systems and the reliability of transactional processing with a static reconfiguration. They introduce the concept of transactional patterns to ensure reliable composite Services. Particularly, (Montagut, 2006) proposes a solution to automate the design of transactional composite Web services. However, it does not respond to unexpected events in context of a pervasive environment, where available components and user requirement are variable. For example, how to do if a component Service became temporarily unavailable at runtime? In addition, defining a set of accepted termination states does not guarantee to find the appropriate Web service for each task. For instance, the designer may specify that Web services to be selected to execute a task T should be compensatable. Therefore, there is no guarantee to find a compensatable Web service for the task T.

In this paper, we propose an approach to easily define flexible and reliable Services compositions. To achieve this goal, we introduce a new concept called dynamic transactional pattern. A dynamic transactional pattern is a convergence concept of dynamic workflow (Lam, 2008) and advanced transactional models (Elmagarmid, 1991). It can be seen as a dynamic coordination and a structured transaction. We integrate the expressiveness power of dynamic workflow models and the reliability of Advanced Transactional Models. Contrary to (Bhiri, 2005) and (Montagut, 2006), our approach responds to unexpected...
events. By unexpected events, we mean the changes that may occur at runtime such as the appearance, disappearance or modification of a component Service.

The remainder of the paper is organized as follow. Section 2 presents an overview of the related works. Section 3 introduces our motivating example. In Section 4, we present the dynamic transactional composite Web services, while Section 5 introduces the concept of dynamic transactional patterns. In Section 6, we show how we use the dynamic transactional patterns to specify reliable and flexible dynamic composite Services. Section 7 introduces our specification and the verification approach. In Section 8, we draw some conclusions and provide some perspectives related to our future research.

2. RELATED WORK

The formal specification is necessary to properly research, analyze and manipulate Web services. Several formalisms have been proposed, such as Petri nets (Benatallah., 2003), event calculus (Kowalski, 1989), or Event-B (Abrial, 2010). Petri nets (Benatallah, 2003) can be used to model Web services. This approach, based on Web service algebra, allows the expression of specific operators to manage the control flow (e.g. sequence, alternative, parallel, selection). Petri nets provide mechanisms to analyze the simulation process, but they do not allow the execution of an orchestration.

Temporal theories have been introduced to represent temporal knowledge, to specify areas of dynamic objects or to reason to solve problems. The approach introduced in (Bai, 2008) is based on event calculus (Kowalski, 1989). First, it allows the formalization of a BPEL orchestration through a translation step that enables the representation of an orchestration as a set of predicates and the verification of functional and non-functional properties before, and during the execution of the orchestration. These properties are themselves expressed as a set of predicates described in event calculus. However, the description of properties in event calculation requires some expertise, remote standard in the Web services field. In addition, the translation phase between BPEL4WS and his formalization language remains present as in the other approaches, which leads to the same restrictions, i.e., potential loss of semantics. LTSA-WS (Foster, 2006) is an approach allowing the comparison of two models, the specification model and implementation one to specify and verify the Web service composition. In case of no consistency of the executions traces of the model generated by the visual tool LTSA, the implementation is fully resumed: in this approach, the verification step is too late which is considered as a weak point.

Event-B (Abrial, 2010) is a state oriented method like the transactional model of Web service and it provides a rigorous specification and verification of distributed or concurrent system using proof. Key features of Event-B are the use of set theory as a modeling notation, the use of refinement to represent systems at different abstraction levels and the use of mathematical proofs to verify consistency between refinement levels. Event-B comes with a tool support in the form of the Rodin Platform that stores templates in a database and provides new powerful provers, which manipulated using a graphical interface. Proof based approaches do not suffer from the classical state space explosion problem. Another advantage is ProB tool (Leuschel M., 2003) which allows the animation and model checking of Event-B specifications.

Workflow (Jablonski, 1996) has become a key technology for business process automation, providing a great support for organizational aspects (business-to-business interaction), user interface, monitoring accounting, distribution and heterogeneity. In (van der Aalst, 2000), the authors propose a collection of control-flow patterns that can be categorized into six categories: basic control flow patterns, advanced branching and synchronization patterns, structural patterns, multiple instance patterns, state-based patterns and cancellation patterns. Unlike (van der Aalst, 2000) and (Russell N. a., 2006) that mainly focused on static workflow patterns, we will focus in this work on dynamic workflow patterns as well as dynamic fork, dynamic join and dynamic sequence (Lam, 2008). Other closely related works include (Barros, 2005) and (Russell N. a., 2005) that deal with workflow data patterns and Service interaction patterns.

Several existing standards such as BTP (Ceponkus, 2002) and WS-transaction (OUT, 2004) support a two-phase Web
services coordination. They are based on a set of extended models to specify Web services interaction. Firstly, the Web services participants should agree on a specific model before starting interactions. Then, the corresponding coordination layer technologies support the appropriate messages exchanges according to chosen transactional model. These proposals inherit the rigidity of the extended transactional models. In (Bhiri, 2005), the authors specified flexible and reliable composite Services. It is true that this solution is reliable. However, it is not clear how it aggregates new transactional properties and responds to unforeseen events in context of pervasive environment. For example, how to do when a component Service is no longer available at runtime? Our approach can complement these efforts and overcome these issues. Indeed, it can be used to specify reliable and flexible compositions with a dynamic reconfiguration. Thus, if a component becomes unavailable, another Web service providing the same output and satisfy the same transactional requirements can automatically substitute it.

Advanced Transaction Models (Elmagarmid, 1991) are proposed to support new database applications by relaxing transaction isolation and atomicity to better match the new requirements. Their limitations come mainly from their inflexibility to incorporate different transactional semantics as well as different behavioral patterns into the same structured transaction (Nektarios, 2002). In addition, workflow systems (van der Aalst, 2000), as the key technology for business process automation (Medjahed, 2003), lack rigorous mechanisms for reliability and correctness. To overcome these limitations, several works were proposed to ensure reliability of Web services compositions. (Rusinkiewicz, 1995) proposed a transactional Workflow system supporting multitask and multisystem activities where: (i) different tasks may have several execution behaviors or properties, (ii) designers define the coordination of the different tasks, and (iii) specify their required failure atomicity. In (Bhiri, 2005), the failure atomicity requirements are defined by specifying a set of Accepted Termination States (ATS). However, these proposals do not respond to unexpected events (the changes that may happen at runtime such as Web service can appear, removed, or be updated).

### 3. Motivating Example

In the following, we introduce a scenario to illustrate our approach. Let us consider a Touristic Circuit Reservation (TCR) application (see Figure 1).

TCR application is initially composed of multiple tasks performed by different processing entities (Web services), namely flight booking (T1), car booking (T2), and hotel booking (T3), online payment (T4) and finally sending documents (T5). First, the customer performs the flight-booking task. As shown in Figure 1, several Web services can be selected to execute a task. For instance, both Flight Booking (FB) and Flight Reservation (FR) Web services can provide the flight-booking task. Next, the customer may only perform hotel reservation, as it is not required to book both Hotel and Car for a trip, so the Car booking Service should be optional. Only the Car Booking (CB) Web service provides the car reservation task. For the hotel reservation, either Hotel Reservation (HR), or Hotel Booking (HB) Services can be selected. Whenever, the hotel and the flight booking tasks are performed successfully, the customer is requested to pay. One of the following Web services: Credit Card (CC), Check (CH) and caSH (SH) carries out the payment procedure. Finally, the reservation documents are sent to customers using one of the following Web services: Send Document by Fedex (SDF), Send Document by DHL (SDH) and Send Document by TNT (SDT). For a given task, several Web services can be selected. For that purpose, we use a multi-criteria selection (Alireza Afshari, 2010) based on the quality of Service (QoS) attributes to determine the best ones to be invoked to perform the user’s request.

During the execution of the TCR application, several faults can occur. For example, the service performing the hotel-booking task is no longer exist or out-of-service temporarily; the application can choose another service,
which has exactly the same functionality and satisfying the same requirements. Thus, the application can automatically switch to a new service without harm. TCR application can be updated as follows. An activity, called Customer Requirements specification (CRS), is defined, allowing customers to specify their preferences (see Figure 2). CRS is quality of services constraints (e.g. cost, execution time, availability). User Requirements Definition (URD) Web service performs it. The execution order and the recovery mechanism should be also modified. Indeed, CRS activity is performed just after receiving a customer request. Then, flight, hotel and car booking are executed simultaneously and are synchronized. Once these activities are successfully performed, the sending documents task is carried out after the termination of the payment task. The recovery mechanism is automatically generated based on this new structure and the transactional properties of the selected Web services.

Modeling this example using either Advanced Transactional Models, or dynamic workflow patterns is not easy. Indeed, Advanced Transactional Models is too rigid to enable a control structure, and they do not support bottom-up applications design, starting from predefined business process and using pre-existing systems or Services with diverse semantics. In addition, dynamic workflow systems require sound mechanisms for reliability and correctness in case of failure. For example, if the task T2 (car booking) or T4 (payment) fails for any reason or became unavailable, then how to do to automatically recover the global transaction?

4. DYNAMIC TRANSACTIONAL COMPOSITES SERVICES MODEL

In this section, we propose a formal model for dynamic Web services composition. Our model integrates the expressiveness and the power of dynamic workflow models and the reliability of Advanced Transactional Models. The originality of this model is the flexibility offered not only to designers, to specify their requirements in terms of control structure, but also to the customers to specify their needs (for instance, QoS, need for new activities, temporal constraints). We start from an abstract specification to define the transactional behavior (Section 4.2) using the Event-B formalism (Section 4.1), then, we specify the dynamic aspect of Web service (Section 4.3). In this section, we show how to specify formally the DTCS model (section 4.4). To do so, we refine the abstract model by introducing the Service dependencies.

4.1 Event-B

Event-B (Abrial, 2010) is a formal method based on the theory of sets, enabling incremental development of software through sequential refinement. A model uses two types of entities to describe a system: machines and contexts. A machine represents the dynamic parts of a model. Machine may contain variables, invariants, theorems, variants and events whereas contexts represent the static parts of a model. It may contain carrier sets, constants, axioms and theorems. Those constructs appear on Figure 3.

A machine is organized in clauses: VARIABLES represents the defined variables of the model. The clause INVARIABLES represents the invariant properties of the system and must allow at least the typing of variables declared in the VARIABLES clause. THEOREMS contain properties that can be derived from invariant properties. EVENTS clause contains the list of events of the model. An event is modeled with a guarded substitution, is fired when its guards are evaluated to true. The events occurring in an Event-B model affect the state described in VARIABLES clause. An Event-B model may refer to a context or a machine. A context consists of the following clauses: SETS describe a set of abstracts and enumerated types. The clause CONSTANTS represent the constants of the model whereas AXIOMS contain all the properties of the constants and their types.

The concept of refinement is the main feature of Event-B. It allows an incremental design of systems. In any level of abstraction, we introduce a detail of the modeled system. A series of proof obligations must be discharged to ensure the correctness of a refinement including the proof obligations of the concrete initialization, the refinement of events, the variant and the proof that the concrete and the abstract machine are deadlock-free. Establishing proof obligations (POs) generated by a RODIN platform tool called proof obligations generator ensures the Correctness of Event-B machines. M is an Event-B model with v being variables and I(v) is an invariants. Let E be an event of M with guard G(v) and before-after predicate R(v,v'). The initialization event is a generalized substitution of the form v: Init(v). Initial proof obligation guarantees the satisfaction of the invariants of the initialization: Init(v) ⇒ I(v). The second proof obligation is related to events. Each event E should preserve invariants after its triggering. The Feasibility statement (FIS) and the invariants preservation (INV) are given in the following predicates (Voisin, may 2005):

- FIS: I(v)∧G(v)⇒∃v'. R(v,v')
- INV: I(v)∧G(v)∧∀R(v,v')⇒I(v')
An Event-B model M with invariants I is well formed, denoted by \( M \models I \), only if M satisfies all invariants I.

### 4.2 Transactional Web Service

Web services are self-contained and self-describing software components that are made available and accessible across a computer network.

Transactional Web service is a Service that emphasizes transactional properties for its characterization and usage (Bhiri, 2005). Therefore, a Web service can be considered as a simple transaction, which can be compensatable (c), retriable (r), or pivot (p) (Mehrotra, 1992). Formally, we define a transactional service as an abstract type, called SERVICE, where the function properties (properties \( \in \) SERVICE \( \rightarrow \) \( \mathbb{P} \{ (r,p,c) \} \)) defines the transactional properties of Web services. A Web service, ts, is said to be retrievable (properties(ts)=\{r\}) if it is sure to complete after a finite number of activations. It is said to be compensatable (properties(ts)=\{c\}) if it offers compensation policies to semantically undo its effects; whereas it is said to be pivot(properties(ts)=\{p\}) if once it successfully completes, its effects remain and cannot be semantically undone.

\[
\text{MACHINE MCS0}
\]
\[
\text{SEES CSWT}
\]
\[
\text{VARIABLES}
\]
\[
\text{DTCS}
\]
\[
\text{rank}
\]
\[
\text{states}
\]
\[
\text{availability}
\]
\[
\text{INVIANTS}
\]
\[
typing1: \text{DTCS} \in \text{P(SERVICE)}
\]
\[
typing2: \text{states} \in \text{DTCS} \rightarrow \text{STATE}
\]
\[
typing3: \text{rank} \in \text{SERVICE} = \mathbb{N}
\]
\[
\text{END}
\]

\[
\text{CONTEXT CSWT}
\]
\[
\text{SETS}
\]
\[
\text{SERVICE STATE OPERATION TP}
\]
\[
\text{CONSTANTS}
\]
\[
\text{active initial aborted failed cancelled completed compensated properties}
\]
\[
\text{\{c,r\} ∈ \text{R at CT}}
\]
\[
\text{AXIOMS}
\]
\[
\text{axm11: RtsSERVICE} \rightarrow \text{N}
\]
\[
\text{axm12: AitSERVICE} \rightarrow \text{N}
\]
\[
\text{axm13: CtsSERVICE} \rightarrow \text{N}
\]
\[
\text{END}
\]

**Figure 4. The abstract model**

A service can combine several transactional properties, which leads to a new behavioral property. For instance, a service can be pivot and retrievable (properties(ts)=\{p,r\}). Similarly, a service can be compensatable and retrievable which leads to a new behavioral property (properties(ts)=\{c,r\}). However, it cannot be compensatable and pivot. Thus, the set of all possible combinations for the behavioral property of a service is \( \{ \{r\}, \{c\}, \{p\}, \{r,p\}, \{c,r\} \} \). An enumerated set, STATE, whose possible values are active, initial, aborted, completed, compensated, failed and canceled, is used to describe the Service state. All these concepts are defined in a context called “CSWT” (see Figure 4). The state failed is not a termination state of a retrievable service, whereas compensated is a termination state of a compensatable one.

The dynamic part of our initial model (e.g., variables, invariants, events) is encapsulated in a machine component namely “MCS0”, which sees the context “CSWT”, described above (see Figure 4). A variable namely, DTCS is defined as a set (DTCS∈\( \mathbb{P}(\text{SERVICE}) \)) which contains the selected Web services that will be invoked to respond to the customer’s request. The service states, represented by a total function states (states\( \in \)DTCS→STATE). In our approach, we use the concept of event to model the internal behavior of a Service. Indeed, the event “ACTIVATE” fires simultaneously the execution of a set of services. Therefore, the event “COMPLETE” normally terminates the execution of a Service ts (states(ts)= completed). The event “RETRY” reactivates the execution of a service if it is retrievable (r\in\text{properties(ts)}), whereas the event “COMPENSATE” semantically compensates the already committed work.

### 4.3 Web Service Selection

In the previous section, we described the transactional behavior of a Web service while the goal of this section is to model the dynamic selection.

The Web services selection (Alireza Afshari, 2010) (Raj, 2010) is an essential process for the Web services composition. It consists in finding the appropriate available Web services based on quality of Service (QoS) factors to fulfill an objective defined by a customer at runtime. In this work, we consider the following QoS factors: response time (Rt), Availability (At), Cost (Ct). These QoS attributes are defined as functions (see Figure 4).

\[
\text{RECEIVE ANY }
\]
\[
\text{Wa Wc Wr} \to \text{R}
\]
\[
\text{WHERE}
\]
\[
\text{grd1: Wa} \in \mathbb{N} \land \text{Wc} \in \mathbb{N} \land \text{Wr} \in \mathbb{N}
\]
\[
\text{grd2: R} \in \text{P(OPERATION)}
\]
\[
\text{grd3: Wa+Wr+Wc=100}
\]
\[
\text{THEN}
\]
\[
\text{act11: rank} := \{s \in \text{SERVICE} \mid s:= (Rt(s)+Wc+At(s)+Wa+Ad(s))\}
\]
\[
\text{act12: operations} := \text{R}
\]
\[
\text{END}
\]

**Figure 5. User requirements specification**

The customers can define their preferences by assigning for each QoS factors a weight value. Formally, we introduce the event “RECEIVE” (see Figure 5). The input parameters of the event “Receive” Wr, Wa and Wc indicate the customer preferences, while the parameter “R” contains the requested operation. In this event, we compute the rank for a given service on the basis of one of its operations which appears in R. In this work, we suppose that a Web service may feature only one requested operation for the same composite. The rank of an individual service is computed using the following formula:

\[
\text{rank} (s) = \sum_{q \in \text{QoS}} q(s) \times W(q)
\]

Where q(s) denotes the ith QoS parameter value of s and W(q) defines the weight of the ith QoS parameter representing the user priorities. The weight satisfies the constraint: W(q1) + W(q2) + ... + W(qn) = 100.
The QoS attributes are normalized and multiplied by 100 in order to obtain an integer value in the interval [0,100]. In our event-B model, the rank value is computed by the substitution action “act11” of event “RECEIVE” (see Figure 5).

Once the Web service are ranked, the selection procedure is triggered. The Web services maximizing the ranking function “rank” are selected to perform a set of functionalities. Formally, the event “SELECT” (see Figure 6) returns the set of candidates, SWT, that better satisfy the customer needs.

4.4 Dynamic Transactional Composite Services

A dynamic composite Service, which is called also objective based composition, is a conglomeration of existing Web services working in tandem to offer a new value-added Service (Mustafa, 2009) and (Jureta, 2007). It is composed at runtime, i.e., the Web services that participate in composition process are selected and composed at runtime.

A Dynamic Transactional Composite Service (DTCS) is a dynamic composite Service emphasizing a transactional property for the composition and the synchronization of its components. Thus, we consider a DTCS as a structured transaction, where the components services are the sub transaction and the interactions are the Service dependencies. The dependency relationships are created during the execution of a DTCS by executing a set of rules. Briefly summarized, the following three steps illustrate our dynamic composition process:

- A Services Selection Engine (SSE)
- A Transaction Management Engine (TME)
- A Composition Generator Engine (CGE)

SSE module assigns, for each task, the appropriate Web services based on the QoS properties and user requirements from the Services Store. TME module, which includes a flow generator component, generates the transactional flow by executing a set of consistency rules. The CGE component checks firstly the availability of each selected Web services before generating the requested DTCS. The Services Store is a reduced Service registry containing the description of potential (can be selected to perform a task t) Web services, including the provided Service contract and binding information. It can be updated automatically once an activity is added, removed, or new Web services (that can be selected) are available. In this register, Web services are organized according to their class features function. According to our motivating example (see Figure 1), the Services Store initially organized in classes namely flight booking, hotel booking, car booking, payment and finally sending document. The main goal is facilitating and minimizing the complexity of the dynamic composition process.

The interaction between DTCS components Service expresses several kinds of relation in form of dependencies such as sequence, compensation, abortion and cancellation. These dependencies should respect some semantic restrictions. To do so, the following safety requirements should be considered:

**REQ-1:** An abortion dependency from s1 to s2 can exist only if there is an activation dependency from s1 to s2.

**REQ-2:** A compensation dependency from s1 to s2 exists if (1) s1 may fail, (2) s2 is compensable, and (3) it exists an activation dependency from s2 to s1, or s1 and s2 execute in parallel and are synchronized.

**REQ-3:** A cancellation dependency from s1 to s2 exists only if s1 may fail and s1 and s2 execute in parallel and are synchronized.

These consistency rules are specified in Event-B formalism as predicates using the concept INVARIANTS namely REQ-1, REQ-2 and REQ-3 (see Figure 7).

The activation dependency that is formally defined by a variable’s name depAct (see Figure 7) expresses a partial ordering of component Services execution. An activation dependency from s1 to s2, s1→s2, exists only if the completion of s1 can fire the activation of s2.

A compensation dependency, s1→s2, from s1 to s2 exists only if the failure or the compensation of s1 may fire the compensation of s2 (states(s1) ∈ {failed, compensated }). A cancellation dependency, which is formally defined by variable’s name depCn1 (see Figure 7) allows signaling a Service execution failure to other Service(s) being carried out in parallel by canceling their execution if necessary. A cancellation dependency from s1 to s2, s1→s2, exists only if the failure of s1 may fire the cancellation of s2 (states(s1) = "cancelled "}.


http://hipore.com/jjsc

![Figure 6. Web services Selection](http://hipore.com/ijsc)

![Figure 7. The consistency rules are specified using invariant concept](http://hipore.com/ijsc)
failed). An abortion dependency that is defined by variable’s name depAbt (see Figure 7) allows propagating failures (causing the DTCS abortion) from one Service to its successor(s) by aborting them. An abortion dependency from s1 to s2, s1⇒s2, exists only if the failure, cancellation or the abortion of s1 may fire the abortion of s2 (states(s1)∈{failed, cancelled, aborted}).

In this section, we introduce the concept of dynamic transactional pattern, a new paradigm, which we propose to specify reliable dynamic Web services compositions. Dynamic transactional patterns extend dynamic workflow patterns (Lam, 2008) with transactional dependencies, thus allowing bridging their transactional lack.

5.1 Dynamic Control-Flow Patterns

The dynamic workflow patterns (Lam, 2008) are mainly used for handling situation where decision is made during runtime. They allow decision to be made just before the instantiation of activities. Unlike (van der Aalst, 2000) and (Russell N. a., 2006) that mainly focus on static workflow patterns, the dynamic Workflow patterns can be automatically reconfigured at runtime if a subprocess is no longer available or is added. Moreover, dynamic Workflow patterns support multiple instantiation, thus allowing for more than one instance of the same activity to be active at the same time.

In this paper, we study only the following three patterns: DAD-split (Dynamic AndSplit), DAD-join (Dynamic AndJoin) and DSQ (Dynamic SeSequence) to explain and illustrate our approach. The concepts presented here can be applied similarly also to others basic pattern like DOR-split (Dynamic OR-split), DOR-join (Dynamic OR-join) and DNM-join (Dynamic N out Of M Join). In the following, we describe the dynamic transactional patterns and also their formal semantic.

4.5 Control and Transactional Flow

The transactional dependencies are totally defined by the compensation, abortion and cancellation dependencies between components services. The activation and transactional dependencies express at a higher abstract level respectively the control flow and the transactional flow of a DTCS.

The control flow of a DTCS defines a partial ordering of component Services activations. Intuitively, the DTCS control flow is defined by the set of its activation dependencies. In our approach, we define a control flow as a function, cflow, taking as input a set of Web services and returning set of activation dependencies relationships.

The transactional flow of a DTCS defines recovery mechanisms for handling failures and recovery. Intuitively, the DTCS transactional flow is defined by its component Services transactional properties and its set of transactional dependencies. In our approach, we define a transactional flow as a function, tflow, which returns a set of transactional flow given a set of Web services.

5. Dynamic Transactional Patterns

One of the most interesting features that Web service technology offers is the possibility of creating dynamically new value-added Web service by combining those that better meet the customer’s requirements. For this reason, we introduce new event “COMPOSE” (see Figure 8). This event is used to define the services composition. Indeed, it takes a set of Web services “SWT” and a workflow structure “WF” as input parameters and returns the corresponding transactional dependencies. The transactional dependencies are generated by executing a set of derivation rules.
Formally, we define the DAD-split pattern as a function taking as input a Web service and returning a set of Web services (see typing31 in Figure 9).

As shown in Figure 10, the resulting control flow of the application of the DAD-split pattern to a set \{in,s1,s2\} is as follow: cflow(in\{in,s1,s2\})=\{in\rightarrow s1, in\rightarrow s2\}. This control flow can be redefined automatically when a single subprocess is either added (see Figure 10.a), or removed (see Figure 10.c) from the business process. In the later case (see Figure 10.c), a DAD-split can be replaced by a DSQ pattern. Moreover, another case may occur in which a subprocess can be replaced by an equivalent one (see Figure 10.b).

A DAD-join pattern (see Figure 11) is a dynamic join pattern. It is specified as a point in a dynamic process where multiple parallel subprocess converge into one single process of control, thus synchronizing multiple process. Formally, we define the DAD-join pattern as a function taking as input a Web service and returning a set of Web services (see typing32 in Figure 9).

As shown in Figure 11, the resulting control flow of the application of the DAD-join pattern to a set \{s1,s2,out\} is as follow: cflow(\{s1,s2,out\}) = \{s1\rightarrow out, s2\rightarrow out\}. This control flow can be redefined automatically when a single subprocess is either added (see Figure 11.a), or removed (see Figure 11.c) from the business process. In the later case, a DAD-join can be replaced by a DSQ pattern. Moreover, Figure 11.b shows that subprocess can be substituted by another one.

A DSQ (see Figure 12) is a dynamic sequence pattern. It specifies a point in a dynamic process. It specifies that a single subprocess be activated after the termination of another one. It provides a dynamic control flow since a single subprocess can be added (see figures 12.b, 12.c) to the business process, or another (see Figure 12.a) can replace a single subprocess. Formally, we define the DSQ pattern as a function that returns the successor of a given Web service (see typing33 in Figure 9). The resulting control flow of the application of DSQ to a set \{s1,s2\} is defined as follow: cflow(\{s1,s2\})=\{s1\rightarrow s2\}.

5.2 Extending Dynamic Control-Flow Patterns

To extend the dynamic Control-Flow pattern with transactional dependencies, we refine the machine “MCS2” by introducing new variable “tflow”, which is defined as a function taking as input a parameter a set of Web services, S, and returning sets of compensation (tflow(S)(2)), the abortion (tflow(S)(0)) and the cancellation (tflow(S)(1)) from the business process. In the later case (see Figure 13). At this refinement level, the abstract variables “depAbt”, “depCnl” and “depCps” are removed because they are implemented by concrete one “tflow”. To ensure the preservation of the abstract variables value and therefore a correct refinement of machine, gluing invariants are required (see gl41, gl42, gl43).

Figure 12. Dynamic reconfiguration of DSQ pattern after adding and replacing of a Web service

The DAD-split pattern can be extended with a set of transactional dependencies for handling failure and recovery mechanism. It supports abortion, cancellation and a compensation dependencies. A compensation (or a
cancellation) dependency from \( s_1 \) to \( s_2 \) can be defined only if \( s_1 \) and \( s_2 \) are synchronized. Similarly, the DAD-join pattern can be extended DAD-join with a set of any kind of transactional dependencies (e.g., abortion, cancellation, or compensation), while DSQ can be extended only with abortion and compensation dependencies.

### 6. Pattern-based Services Composition

In this section, we describe how to generate automatically a consistent transactional flow and therefore ensuring reliable Web services compositions.

#### 6.1 Services Composition

The use of dynamic Workflow patterns is an interesting idea to compose a set of existing Web services in pervasive environment, where Web services may dynamically appear or disappear.

The composition process refers to defining the control flow and the transactional flow of the selected set of Web services. We use the composition pattern to specify the control flow, while the transactional flow are defined using the functions “\( cflow \)” and “\( tflow \)”. We define, an instance of pattern as the application of a pattern to either a set of Web service or a single web service. For example, \( P_1 = \{ \text{URD} \rightarrow \{ \text{FR,CB,HR} \} \} \) is an of the pattern DAD-split (see Figure 15). In our approach, the control flow of a DTCS is defined as the union of a set of instances. Each instance of the dynamic Workflow pattern is extended with a set of transactional dependencies using the function “\( tflow \)”. For example, Figure 15 illustrates how we can specify a composite Services by connecting a set of patterns instances: \( P_1, P_2 \) and \( P_3 \).

---

**Figure 14. Services composition using dynamic workflow patterns**

Formally, we refine the event “COMPOSE” specified in the machine “MCSI” in order to express the Web services composition using dynamic transactional patterns. We add a new parameter’s name skeleton, which its type is expressed by the predicate \( \text{typing31} \) (see Figure 14). In addition, we define a witness predicate, thus allowing to define the link between the abstract parameter, WF and the concrete one skeleton (WF = union(ran(skeleton))).

**Figure 15. Services composition using dynamic workflow patterns**

Connecting a set of dynamic transactional patterns instances can lead to a control flow and/or a transactional flow inconsistency. For instance, control consistency problem can rise when instances are disjoined without shared Services allowing connecting them. Similarly, transactional inconsistency can rise when a component Service fails causing the entire DTCS abortion, without compensating the partial work already done.

#### 6.2 Ensuring a Reliable Services Composition

A DTCS is said to be reliable in a transactional context if its execution respects the all or nothing rule, i.e., all components are either fully completed or not executed at all. To achieve this goal, we adopted the following approach:

- **Step-1.** Definition of a consistent control flow;
- **Step-2.** Generating a consistent transactional flow.

The control flow defines the structure (skeleton) of the dynamic composite Service. It is generated automatically by a component called flow generator that is included in TME module. In this work, we are using the following rules to ensure the consistency of the DTCS structure:

- **REQ-4:** A consistent control flow should begin with a DSQ, or a DAD-split pattern;
- **REQ-5:** A DSQ pattern can be followed by one of the following patterns: DSQ or DAD-split;
- **REQ-6:** A DAD-split pattern can be followed by a DAD-join pattern.

These rules can be generalized by considering other dynamic control-Flow patterns such as DOR-split, DOR-join and DNM-join. In addition, a component Service in a given DTCS can be itself a dynamic composite Service where its control flow is consistent (it respects the above rules), thus allowing to use dynamic transactional patterns in a nested way inside a dynamic composition.
Back to our example (see Figure 15), the control flow of the dynamic composite Service that performs the touristic circuit reservation respects the proposed rules (REQ-4, REQ-5 and 6). Indeed, the control flow is started by an instance of DAD-split pattern, which is followed successively by instances of DAD-join and DSQ patterns.

The next step consists in defining a consistent transactional flow of a set of Web services respecting the constraints (REQ-1, REQ-2, and REQ-3). We need not only to a consistent control flow, but also to the transactional properties to generate it. A transactional flow is consistent if it respects the following recovery actions:

**RA-1:** Try to look for alternative service to replace the failed component.

**RA-2:** Compensate the work already done.

**RA-3:** Cancel all running executions in parallel.

The second and the third recovery action (RA2 and RA3) define alternative recovery actions, which consecutively executed for the first one. They are applied when RA1 is not enough for resuming a failed process. RA1 is performed either if a Web service is considered as unavailable, or due to a malfunctioning error.

The next step consists in generating a consistent transactional flow. For that purpose, we need not only to a consistent control flow structure, but also to the transactional properties of the selected Web Service. From the constraints R1, R2 and R3, we can deduce the following three rules to generate the transactional flow of a set of Web service:

**Rule-1:** if it exist an activation dependency from s1 to s2, then define an abortion dependency from s1 to s2;

**Rule-2:** if (1) s2 is compensatable, (2) s1 is not retriable and (3) there is an activation dependency from s1 to s2, or s1 and s2 run in parallel and are synchronized, then define a compensation dependency from s1 to s2; and

**Rule-3:** if s1 is not retriable, s1 and s2 run are synchronized, then define a cancellation dependency from s1 to s2.

In our Event-B model, these rules are specified in the clause THEN of the event “COMPOSE” in the machine “MCS2” by the substitutions act21, act22, act23 and act24 (see Figure 8). By using these rules, we generate automatically a consistent transactional flow based on the transactional properties of the selected Web services.

### 6.3 Ensuring a Flexible Services Composition

The main advantage behind the dynamic workflow patterns is the ability to incorporate alternative execution paths within a process model at design time allowing customer to select one or more task from the set of available tasks. For example, a customer may request only the task T1, T3, T4 and T5 for booking touristic circuit. Moreover, another advantage is the ability to modify a process model at runtime such that one or all of the currently executing process instances are migrated to a new process model. Unlike the most of existing Workflow patterns, dynamic workflow patterns support the previously mentioned flexibility kinds. Indeed, their structures are defined at runtime and they can be automatically reconfigured.

REPLACE

ANY old new
WHERE

grd11: new ∈ SERVICE ∧ old ∈ SERVICE
grd12: old ∈ DTCS ∧ new ∈ DTCS
grd13: properties(old)=properties(new)

THEN

act31: DSQ= {s ∈ DTCS | (old)∧old=DSQ[s]=new}∪
{t ∈ DTCS | (old)∧old=DSQ[∥new∥]∪
{t ∈ DTCS | (old)∧old=DSQ[s]=new}∪
{t ∈ DTCS | (old)∧old=DSQ[∥new∥]∪

act32: DAD_split= (old)∪(s ∈ dom(DAD_split)∧ old∈DAD_split(s)↦(DAD_split(s)∖{old}))∪
{s ∈ dom(DAD_split)∧ old∈DAD_split(s)↦(DAD_split(s)∖{old}))∪

act33: DAD-join= (s ∈ dom(DAD_join)∧ old∈DAD_join(s)↦-old∈DSQ[∥new∥]∪
(\(s₁ \sqcup s₂ \in dom(DAD_join))∧ old∈DAD_join(s₁ \sqcup s₂)⇒ old∈DAD_join(s₁ \sqcup s₂))∪

END

Figure 16. Automatically reconfiguration of a composite service in case of failure

In this work, we only treat the problem of unavailability of components service. For this reason, we introduce new event called "REPLACE". This event is defined in the machine “MCS3". The actions act31, act32 and act33 reconfigure automatically all instances of the patterns DSQ, DAD-split, and DAD-join by substituting the faulty component (old) in all instances where it belongs by the replacing service (see Figure 16).

### 7. Modeling and Verification

In our approach, we have combined the modeling and verification activities related to formal development process supported by Event-B method. The verification activity is based on the theorem proving and ProB (Leuschel M., 2003) model-checker integrated to the Rodin platform.

#### 7.1 Specifying an Event-B Model

We start our approach by specifying an Event-B model. We have used the CONTEXT structure offered by Event-B to identify the parameters related to the DTCS. These parameters are modeled in Event-B by sets and constants. We have established theorems describing interesting properties that can be deduced from obvious properties (axioms) attached to these parameters. In addition, Event-B machines with variables and events model the dynamic aspects attached to the DTCS. We paid particular attention to typing and invariant properties of these variables. Similarly, events are identified based on the abstraction level of the concerned Event-B machine. We carefully

http://hipore.com/jjsc
studied and justified the gluing invariants linking the abstract and concrete variables at every refinement step. This is also true for new events introduced at each refinement step.

7.2 PROOF-ASSISTED MODEL CHECKING

The next step consists in discharging proof obligations, which are generated by a component, called Proof Obligations generator. To achieve this goal, we must prove two things:

- The initialization leads to a state where the invariant is valid.
- Assuming that the machine is in a state where the invariant is valid, every enabled event leads to a state where the invariant is valid.

The initialization of a machine is a special form of event. It has neither parameters nor guards.

**Figure 17. Proof obligations**

Rodin generates a set of proof obligations for every invariant that can be affected by an event, i.e., the invariant contains variables that can be changed by an event. The name of the proof obligation is then “evt/label/INV”. It can be either automatically/interactively discharged (marked with ☑), or undischarged (marked with ☐). The symbol “A” means that the PO is automatically discharged. The goal of such a proof is to assert that when all affected variables are replaced by new values from the actions, the invariant still holds. For instance, the Figure 17 illustrates an example of proof obligations where the invariants REQ-5, g31 and typing34 are satisfied after its triggering. It indicates also that the invariant REQ-4 and REQ-6 may not satisfied after its executing since the proof obligations “COMPOSE/REQ-4/INV” and “COMPOSE /REQ-6/INV” are undischarged.

Refinement is a main concept in Event-B. Refinements are used to gradually introduce the details and complexity into a model. We have to prove if the concrete machine, “MCS2”, behaves in a way that corresponds to the behavior of abstract one. Consequently, several proof obligations are generated. For instance, the concrete event “COMPOSE” of the machine “MCS2” can only occurs when the abstract one occurs. The resulting proof obligation “COMPOSE/grd21 /GRD” shown in Figure 17.b ensure that under the assumption that the concrete event, “COMPOSE”, is enabled (i.e. its guards are true) and the invariants (both the abstract and the concrete) hold, the abstract guards holds as well.

The advantage of the proof-based approach is that it can handle very complex systems because it does not have to directly check each and every state and because the logics are typically more expressive. The drawback is that it requires human insight and creativity to complete the proofs. Another shortcoming is the inability to produce counterexamples in the event of a failed proof because one does not know whether the required property is not derivable or whether the person conducting the derivation is not ingenious enough. To overcome these problems, we incrementally combined the activity of proof and model checking to correct modeling errors. The main goal is to facilitate and accelerate the software’s design step. We use ProB as model-checker for models. This model-checker tries to find invariant violations not by starting from the initial states and executing operations, but by constraint solving. More precisely, it tries to find a state that satisfies the invariant and enables an operation, such that the resulting state of the operation violates the invariant. Depending on situations, we have made several modifications related to the invariant, guard or to the action of event. For instance, the invariant violation (see Figure 18) found by ProB, signals a specification error. This error

**Figure 18. The evaluation view of the model-checking process**
occurred after triggering event “COMPOSE”. To prevent the invariants, REQ-4 and REQ-6 and others, from being violated (and therefore to allow all proof obligations to be discharged), we have to modify the event “COMPOSE” in machine “MCS2”.

7.3 Validation

ProB is a helpful tool for debugging models, supporting systematic animation of Event-B Models, and non-deterministic operations. It provides fully automatic animation of many B specifications and checks a specification for a range of errors.

We start the animation step by executing the INITIALISATION event. Afterward, we can interact with the model by triggering events. This is done by double-clicking on an enabled event or by right-clicking it and selecting a set of parameters, if applicable. We first trigger the event “RECEIVE” with parameters: Wc=5, Wa=80, Wr=6 and R={T1,T2,T3,T4,T5} to define the user requirements. In this particular case, the user requires that the Web services will be selected should be surely available during the execution of his request. Next, we trigger the event “SELECT” for determining the best available services that better meet this particular requirement. Finally, we trigger “COMPOSE” event in order to dynamically choose the set of Web services. The generated composite Service is the DTCS shown in Figure 15. At this moment, we can launch the execution of DTCS and verify if the defined transactional behavior satisfies all consistency rules by executing the enabled events. In particular, we can verify if the execution of the event “REPLACE” leads usually the DTCS to a valid state in which all invariants are satisfied.

Unfortunately, triggering events to detect invariant violations is not very efficient because we cannot manually cover all possible cases. But, ProB can perform model checking automatically using constraint based checking tool. This checker tries to find invariant violations not by starting the initial states and executing operations, but by constraint solving. More precisely, it tries to find a state that satisfies the invariant and enables a certain operation, such that the resulting state of the operation violates the invariant. However, this process may fail if the model is complex (partial checking).

8. Conclusion

In this paper, we have proposed an Event-B driven approach to specify and validate the dynamic Web services composition. We introduced a new concept called dynamic transactional patterns to specify flexible and reliable DTCS. This concept integrates the expressiveness and the power of dynamic workflow models and the reliability of Advanced Transactional Models. We presented how to dynamically define a consistent transactional flow. A DTCS is gradually verified using a proof assisted model-checking approach. It is finally validated using ProB animator in order to ensure that it behaves as expected. Our verification approach combines the power of the proof and model-checking.

In our future work, we will work on modelling dynamic Web service composition using constraint satisfaction problem CPS. Afterwards, we will propose a heuristic based algorithm solving this CSP. This algorithm will help to reduce the search space of the possible solutions while guaranteeing at the same time optimal solutions.

9. References


http://hipore.com/jsc

Authors

**Mr. ABBASSI Imed** received a diploma of the second cycle in Computer Science with high distinction in 2010 and a master's degree in computer science in 2012 from the Faculty of Sciences of Monastir, Monastir University. He is studying towards a PhD degree in computer science at the National Engineering School of Tunis. His research interests include Service/cloud computing, service composition, service Reliability/flexibility, and formal verification.

**Mr. Lazhar Hamel** is a PhD student in computer science at Sfax University. He received his Master’s degree in Computer Science and Multimedia from the University of Sfax in 2011. His main research interests are software engineering, service composition verification, and formal verification.

**Dr. Zied Jaoua** received the state engineering degree in computer engineering from national engineering school in 2003 from Tunisia, the M.S degree in information technology in 2004 from University Poitiers, France and the Ph.D. degree in 2010 from University Paris 13, France. From 2006 to 2010 he worked at the Laboratory of Signals and Systems (LSS, at Gif sur Yvette France) where his research activities were on robust mobile transmission and joint source-channel decoding problems. Since 2010 Dr. Zied Jaoua is an associate professor in High School of Computer Science and Mathematics, Monastir University. He is currently teaching Distributed Systems and Artificial Intelligence.

http://hipore.com/jsc