Towards an Approach of Formal Verification of Web Service Composition

Mohamed Graiet
MIRACL,ISIMS
Sfax, Tunisia
mohamed.graiet@imag.fr

Lazhar Hamel
MIRACL,ISIMS
Sfax, Tunisia
lazhar.hamel@gmail.com

Raoudha Maraoui
MIRACL,ISIMS
Sfax, Tunisia
maraoui.raoudha@gmail.com

Mourad Kmimech
MIRACL,ISIMS
Sfax, Tunisia
mkmimech@gmail.com

Mohamed Tahar Bhiri
MIRACL,ISIMS
Sfax, Tunisia
tahar_bhiri@yahoo.fr

Walid Gaaloul
Computer Science Department Telecom SudParis
Paris, France
walid.gaaloul@it-sudparis.eu

***Abstract*—Web services can be defined as self-contained modular programs that can be discovered and invoked across the Internet. Web services are defined independently from any execution context. A key challenge of Web Service (WS) composition is how to ensure reliable execution. Due to their inherent autonomy and heterogeneity, it is difficult to reason about the behavior of service compositions especially in case of failures. Therefore, there is a growing interest for verification techniques which help to prevent service composition execution failures. In this paper, we present a proof and refinement based approach for the formal representation, verification and validation of Web Services transactional compositions using the Event-B method.**

***Keywords*-Web service; transactional; composition; Event-B; verification; proof;**

I. Introduction

Web services are emergent and promising technologies for the development, deployment and integration of applications on the internet. One interesting feature is the possibility to dynamically create a new added value service by composing existing web services, eventually offered by several companies. Due to the inherent autonomy and heterogeneity of web services, the guarantee of correct composite services executions remains a fundamental problem issue. An execution is correct if it reaches its objectives or fails properly according to the designer's requirement or users needs. To deal with the web services heterogeneity we proposed in [1] a formalisation of web service composition mediation with the ACME ADL(Architecture Description Language). The problem, which we are interested in, is how to ensure reliable web services compositions. By reliable, we mean a composition for which all executions are correct.

Our work deals with the formal verification of the transactional behavior of web services composition. In this paper, we propose to address this issue using proof and refinement based techniques, in particular the Event-B method [2] [3] used in the RODIN platform [4]. Our approach consists on a formalism based on Event-B for specifying composite service (CS) failure handling policies. This formal specification is used to formally validate the consistency of the transactional behavior of the composite service model at design time, according to users' needs. We propose to formally specify with Event-B the transactional service patterns. These patterns are formally specified as events and invariants rule to check and ensure the transactional consistency of composite service at design time. Most previous work is based on the model checking technique and does not support the full description of transactional web services. Refinement and proof techniques offered by Event-B method are used to explore it and in Section 6 we discuss this approach.

This paper is organized as follows. Section 2 presents a summary of related work on this topic, i.e. on approaches for modeling the behavior of web services. In Section 3 we introduce a motivating example. Section 4 presents the Event-B method, its formal semantics and its proof procedure and introduces our transactional CS model. In Section 5, we present how we specify a pattern-based of the transactional behavior using the Event-B. An overview of the validation methodology is given in Section 6.

II. Related Works

Some web services are used in a transactional context, for example, reservation in a hotel, banking, etc.; the transactional properties of these services can be exploited in order to answer their composition constraints and the preferences made by designers and users. However, current tools and languages do not provide high-level concepts for express transactional composite services properties [5]. The execution of composite service with transactional properties is based on the execution of complex distributed transactions which eventually implements compensation mechanisms. A compensation is an operation which goal is to cancel the effect of another transaction that failed to be successfully completed. Several transactions models previously proposed in databases, distributed systems and collaborative environments but these models face problems of integration and

transaction management. When a service is integrated into the composition, it is probable that its transaction management system does not meet the needs of the composition. In order to manage with this focus many specifications proposed to response to this aspects. Many research in this field, WS-Coordination [6], WS-AtomicTransaction [7] and WS-BusinessActivity [8].), aiming for instance to guarantee that an activity is cancelable and / or compensable. The verification step will help ensure a certain level of confidence in the internal behavior of an orchestration. Several approaches have been proposed in this direction, based on work related to the transition system [9], process algebras [10], or the temporal theories [11].

LTSA-WS [12] is an approach allowing the comparison of two models, the specification model (design) and implementation model in order to specify and verify the web service composition. In case of no coherence (consistency) of executions traces of the model generated by the visual tool LTSA , the implementation is fully resume: as a weak point in this approach is the verification phase is too late.

The approach presented in [9] formalizes some operators used for orchestration of Web services as a Petri net and enabling some checks. Petri nets provide mechanisms for analyzing simulation process but do not allow execution. The temporal theories have emerged through the application of logic in Artificial Intelligence. The work presented in [11] is based on one of these theories: Event Calculus. First, the approach allows the verification of functional and non-functional properties. Second, it allows the verification of BPEL4WS orchestration at a static level (prior to execution) and all along the execution of an orchestration. The translation phase between BPEL4WS and its formalization language is presented as in other approaches, which leads to the same restrictions, namely the potential loss of semantics.

In last work [1] [13], SOA (Service Oriented Architecture) defines a new Web Services cooperation paradigm in order to develop distributed applications using reusable services. The handling of such collaboration has different problems that lead to many research efforts. We addresses in these works the problem of Web service composition. Indeed, various heterogeneities can arise during the composition. The resolution of these heterogeneities, called mediation, is needed to achieve a service composition. Then, we propose a sound approach to formalize Web services composition mediation with the ADL (Architecture Description Language) ACME [14]. To do so, we, first, model the meta-model of composite service manager and mediation. Then we specify a semi formal properties associated with this meta-model using OCL (Object Constraint Language) [15]. Afterwards, we formalize the mediation protocol using Armani [16], which provides a powerful predicate language in order to ensure service execution reliability.

The approach presented in [17] consists in extracting an Event-B model from models expressing service compositions and description. These models expressed with BPEL. This approach consists in transforming a BPEL process into an Event-B model in order to check the relevant properties defined in the Event-B models by the services designers The verification of an orchestration before its execution can theoretically limit any undesired behavior, or current work introduces one or more phases of translation between the description and formalization of the orchestration. It is therefore not possible to affirm that what is verified is exactly what is described. In addition, BPEL4WS has no formally defined operational semantics, it is not possible to affirm that what is executed is exactly what is described. That is essentially what we will try to solve in our approach of verifying services compositions.

III. MOTIVATING EXAMPLE

In this section, we present a scenario to illustrate our approach we consider a travel agency scenario (Figure 1). The client specifies its requirement in terms of destinations and hotels via the activity ”Specification of Client Needs” (SCN). After SCN termination, the application launches simultaneously two tasks ”Flight Booking” (FB) and ”Hotel Reservation” (HR) according to customer’s choice. Once booked, the ”Online Payment” (OP) allows customers to make payments. Finally travel documents (air ticket and hotel reservations are sent to the client via one of the services ”Sending Document by Fedex” (SDF) ,”Sending Document by DHL” (SDD) or ” Sending Document by TNT” (SDT). To guarantee outstanding reliability of the service the designers specify that services FB, OP and SDT will terminate with success. Whereas on failure of the HR service, we must cancel or compensate the FB service (according to his current state) and in case of failure of the SDF, we have to activate the SDD service as an alternative.

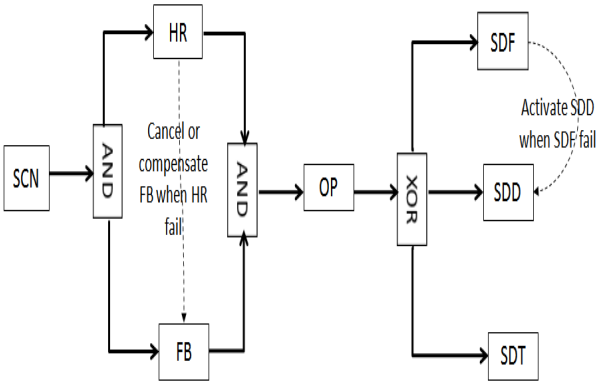

Figure 1. Motivating example

The problem that arises at this level is how to check / ensure that the specification of a composite service ensures reliable execution in accordance with the designer’s requirements. To do so, the verification process should cover the

composite service lifecycle. Basically, at design time the designer should respect the transactional consistency rules. For instance, one has to verify that there is no cancellations dependencies between no concurrent services (for instance, SDD, SDT, SDF), as a cancellation dependency can intrinsically exist only between services executed at the same time. Indeed, discovering and correcting such kind of senseless and potentially costly behavior improve the composite service design. In the other side, after runtime one can discover that in reality the users express the need to cancel or compensate the FB service in failure of the HR service. Starting from this observation, we should propose a technique to discover these discrepancies between the initially designed model and users' evolution needs. Indeed, taking in account this new transactional behavior improves composite service reliability.

IV. Formalizing transactional composite service with Event-B

To better express the behavior of web services we have enriched the description of web services with transactional properties. Then we developed a model of Web services composition. In our model, a service describes both a coordination aspect and a transactional aspect. On the one hand it can be considered as a workflow services. On the other hand, it can be considered as a structured transaction when the services components are sub-transactions and interactions are transactional dependencies. The originality of our approach is the flexibility that we provide to the designers to specify their requirements in terms of structure of control and correction. Contrary to the ATMs [18] [19], we start from designers specifications to determine the transactional mechanisms to ensure reliable compositions according to their requirements. We show how we combine a set of transactional service to formally specify the transactional CS model in Event-B.

The work presented in [20] uses Event Calculus to specify models of web services. Event calculus uses a languages of predicates that requires verification. However Event Calculus are not backed by verification tools. Therefore verification and validation of these models become more complex.

Compared to [20] the big advantage of Event-B is the RODIN platform, which is based on Eclipse. RODIN stores templates in a database and provides new powerful provers which can be manipulated using a graphical interface. Another interesting aspect is the possibility of extending RODIN using plug-ins. Another advantage is ProB tool [21] which, allows the animation and model checking of Event-B specifications. In other words, ProB can visualize the dynamic behavior of a machine B and one can systematically explore all accessible states of an Event-B machine. With this plug-ins RODIN becomes a platform where the user can edit, animate and proving models.

Event-B uses successive refinement to verify that a system satisfies the requirements of a specification; it can repair errors during the development. The complexity of the system is distributed; the step by step proofs are easier. Event-B offers more flexibility and expressivity than the input languages of model checkers.

A. Event-B

B is a formal method based on he theory of sets, enabling incremental development of software through sequential refinement. Event-B is a variant of B method introduced by Abrial to deal with reactive system. An Event-B model contains the complete mathematical development of a discrete system. 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 2.

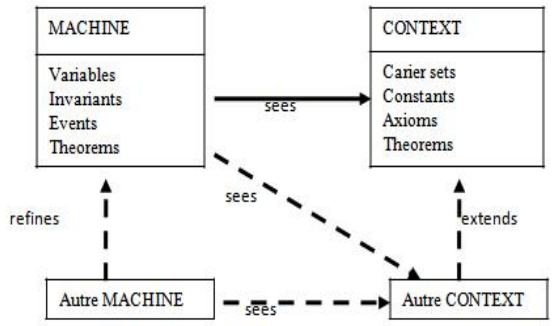

Figure 2. Event-B constructs and their relationships

A machine is organized in clauses:

- VARIABLES represent the state variable of the model.
- INVARIANTS represents the invariance properties of the system, must allow at least the typing of variables declared in the VARIABLES clause.
- THEOREMS contain properties that can be derived from properties invariance.
- EVENTS clause contains the list of events of the model. An event is modeled with a guarded substitution, is fired when its guards evaluated to true. The events occurring in an Event-B model affect the state described in VARIABLES clause.

Each event in the EVENTS clause is a substitution, and its semantics is the calculation of Dijkstra's weakest preconditions. An event consists of a guard and a body. When the guard is satisfied, the event can be activated. When the guards of several events are satisfied at the same time, the choice of the event is to enable deterministic. An Event-B model may refer to a context.

A context consists on the following clauses:

- SETS describe a set of abstract and enumerated types.
- CONSTANTS represent the constants of the model.
- AXIOMS contain all the properties of the constants and their types.
- THEOREMS contains properties deduced from the properties present in the clause AXIOMS.

Refinement: The concept of refinement is the main feature of Event-B. it allows incremental design of systems. In any level of abstraction we introduce a detail of the system modelled. A series of proof obligations must be discharged to ensure the correction of refinement as the proof obligations of the concrete initialization, the refinement of events, the variant and the prove that no deadlock in the concrete and the abstract machine.

Correctness checking: Correctness of Event-B machines is ensured by proving proof obligations (POs); they are generated by RODIN to check the consistency of the model. For example: the initialization should establish the invariant, each event should be feasible (FIS), each given event should maintain the invariant of its machine (INV), and the system should ensure deadlock freeness (DLKF). The guard and the action of an event define a before-after predicate for this event. It describes relation between variables before the event holds and after this. Proof obligations are produced from events in order to state that the invariant condition is preserved.

Let M be an Event-B model with v being variables, carrier sets or constants. The properties of constants are denoted by P(v), which are predicates over constants, and the invariant by I(v). 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 that the initialization of the machine must satisfy its invariant: $Init(v) \Rightarrow I(v)$. The second proof obligation is related to events. Each event E, if it holds, it has to preserve invariant. The feasibility statement and the invariant preservation are given in these two statements [22] [23].

- $I(v) \wedge G(v) \wedge P(v) \Rightarrow \exists v' R(v, v')$
- $I(v) \wedge G(v) \wedge P(v) \wedge R(v, v') \Rightarrow I(v')$

An Event-B model M with invariants I is well-formed, denoted by $M \vDash I$ only if M satisfies all proof obligations.

B. Transactional web service model

By Web service we mean a self-contained modular program that can be discovered and invoked across the Internet. Each service can be associated to a life cycle or a statechart. A set of states (*initial, active, cancelled, failed, compensated, completed*) and a set of transitions (*activate(), cancel (), fail(), compensate (), complete()*) are used to describe the service status and the service behavior.

A service ts is said to be retriable(r) if it is sure to complete after finite number of activations. ts is said to be compensatable(cp) if it offers compensation policies to semantically undo its effects. ts is said to be pivot(p) if once it successfully completes, its effects remain and cannot be semantically undone. Naturally, a service can combine properties, and the set of all possible combinations is r; cp; p; (r; cp); (r; p)[24].

The initial model includes the context *ServiceContext* and the machine *ServiceMachine*. The context *ServiceContext* describes the concepts *SWT* which represents all transactional web services and *STATES* represents all the states of a given *SWT*. These states are expressed as constants.

- A set named *STATES* is defined in the SETS clause which represents the states that describe the behavior of such a service.
- A set named *SWT* is defined in the SETS clause which represents all transactional web services.
- A subset named *SWT_C* is defined in the VARIABLES clause which represents the compensable transactional services.
- A subset named *SWT_R* is defined in the VARIABLES clause which represents the retriable transactional services.
- A subset named *SWT_P* is defined in the VARIABLES clause which represents the transactional services pivot.
- The service state which is represented by a functional relation *service_state* defined in VARIABLES clause gives the current state of such a service.

CONTEXT ServiceContext
SETS
SWT
$STATES$
CONSTANTS
$active$
$initial$
$aborted$
$cancelled$
$failed$
$completed$
$compensated$
AXIOMS
Axm1:$STATES = \{active, initial, aborted, cancelled, failed, completed, compensated\}$
END

The transactional behavior of a transactional web service is modeled by a machine. *Inv1* the invariant specifies that *service_state* is a total function, and that each service has a state.

In our model, transitions are described by the event. For instance the *activate* event changes the status of a service and pass it from *initial* status to *active*. The *compensate* event enables to compensate semantically the work of a service and pass it from *completed* status to *compensated*. The *retry* event changes the status of a service and activate it after his

failure and pass it from *failed* status to *active*. The *complete* event enables to finite the execution of a service with success and pass it from *active* status to *completed*.

MACHINE ServiceMachine
SEES ServiceContext
VARIABLES
$service\_state$
SWT_C
SWT_P
SWT_R
INVARIANTS
Inv1: $service\_state \in SWT \rightarrow STATES$
Inv2: $SWT\_C \subset SWT$
Inv3: $SWT\_R \subset SWT$
Inv4: $SWT\_P \subset SWT$
EVENTS
activate $\triangleq$
ANY
s
WHERE
grd1 : $s \in SWT$
grd2 : $service\_state(s) = initial$
THEN
act1 : $service\_state(s) := active$
END
compensate $\triangleq$
ANY
s
WHERE
grd1 : $s \in SWT$
grd2 : $service\_state(s) = completed$
THEN
act1 : $service\_state(s) := compensated$
END
Retry $\triangleq$
ANY
s
WHERE
grd1 : $s \in SWT\_R$
grd2 : $service\_state(s) = failed$
THEN
act1 : $service\_state(s) := active$
END
complete $\triangleq$
ANY
s
WHERE
grd1 : $s \in SWT$
grd2 : $service\_state(s) = active$
THEN
act1 : $service\_state(s) := completed$
END

C. Transactional composite service

A composite service is a conglomeration of existing Web services working in tandem to offer a new value-added service [25]. It orchestrates a set of services, as a composite service to achieve a common goal. A transactional composite (Web) service (TCS) is a composite service composed of transactional services. Such a service takes advantage of the transactional properties of component services to specify failure handling and recovery mechanisms. Concretely, a TCS implies several transactional services and describes the order of their invocation, and the conditions under which these services are invoked.

To formally specify in Event-B the orchestration we introduced a new context *CompositionContext* which extends the context *ServiceContext* that we have previously introduced.

The first refinement includes the context *CompositionContext* and the machine *CompositionMachine* which refine the machine introduced at the initial model. In this section we show how formally the interactions between CS are modeled. We introduce the concept of dependencies(*depA*, *depANL*, *depCOMP*, etc.).

Dependencies are specified using Relations concept. It is simply a set of couples of services. For example *depA* represents the set of couples of services that have an activation dependency.

CONTEXT CompositionContext
EXTENDS ServiceContext
CONSTANTS
$depA$
$depAL$
$depANL$
$depABD$
$depCOMP$
AXIOMS
Axm1 : $depA \in SWT \leftrightarrow SWT$
Axm2 : $depAL \in SWT \leftrightarrow SWT$
Axm3 : $depANL \in SWT \leftrightarrow SWT$
Axm4 : $depABD \in SWT \leftrightarrow SWT$
Axm5 : $depCOMP \in SWT \leftrightarrow SWT$
END

These dependencies express how services are coupled and how the behavior of certain services influences the behavior of other services. Dependencies can express different kinds of relationships (inheritance, alternative, compensation, etc.) that may exist between the services. We distinguish between ”normal” execution dependencies and ”exceptional” or ”transactional” execution dependencies which express the control flow and the transactional flow respectively. The control flow defines a partial services activations order within a composite service instance where all services are executed without failing cancelled or suspended. Formally, we define a control flow as TCS whose dependencies are only ”normal” execution dependencies.

The transactional flow describes the transactional dependencies which specify the recovery mechanisms applied following services failures (i.e. after fail() event). We distinguish between different transactional dependencies types(compensation, cancelation and alternative dependencies). Alternative dependencies (*depAL*) allow us to define forward recovery mechanisms. A compensation dependency (*depCOMP*) allows us to define a backward recovery mechanism by compensation. A cancellation dependency (*depANL*) allows us to signal a service execution failure to other service(s) being executed in parallel by canceling their execution. It exists an abortion dependency (*depABD*) between a service s_1 and a service s_2 if the failure, the cancellation or the abortion of s_1 can fire the abortion of s_2.

MACHINE CompositionMachine
REFINES ServiceMachine
SEES CompositionContext
activate$\triangleq$ REFINES activate
ANY
s
WHERE
grd1 : $s \in SWT$
grd2 : $service\_state(s) = initial$
grd3 : $(\forall s0.s0 \in SWT \wedge s0 \mapsto s \in depA \Rightarrow service\_state(s0) = completed)$
$\vee(\exists s0.s0 \in SWT \wedge s0 \mapsto s \in depAL \Rightarrow service\_state(s0) = failed)$
THEN
act1 : $service\_state(s) := active$
END
compensate$\triangleq$ REFINES compensate
ANY
s
WHERE
grd1 : $s \in SWT\_C$
grd2 : $service\_state(s) = completed$
grd3: $\exists s0.s0 \in SWT \wedge s0 \mapsto s \in depCOMP$
$\Rightarrow ((service\_state(s0) = failed) \vee$
$(service\_state(s0) = compensated))$
THEN
act1 : $service\_state(s) := compensated$
END

Activation dependencies (*depA*) express a succession relationship between two services s_1 and s_2. But it does not specify when s_2 will be activated after the termination of s_1. The guard added to the activate event which refines the activate event of the initial model expresses when the service will be active as a successor to other (s) service (s) (only after the termination of these services).

For example, our motivating example defines an activation dependency from HR and FB; to OP such that OP will be activated after the completion of HR and FB. That means there are two normal dependencies: from HR to OP and from FB to OP. At this level the refinement of the compensate event is a strengthening of the event guard to take into consideration the condition of compensation of a service when a service will be compensated.

The guard grd4 in the *compensate* event expresses that the compensation of a service s is triggered when a service s0 failed or was compensated and there is a compensation dependency from s to s0. Therefore compensate allows to compensate the work of a service after its termination, the dependency defines the mechanism for backward recovery by compensation, the condition added as a guard specifies when the service will be compensated.

The guard grd4 in the *activate* event expresses when a service will it be activated as a successor of other (s) service (s) (i.e. only after termination of these services) or when will be activated as an alternative to other (s) service (s) (i.e. only after the failure other (s) service (s)).

V. Transactional service patterns

The use of workflow patterns [26] appears to be an interesting idea to compose Web services. However, current workflow patterns do not take into account the transactional properties (except the very simple cancellation patterns category [27]). It is now well established that the transactional management is needed for both composition and coordination of Web services. That is the reason why the original workflow patterns were augmented with transactional dependencies, in order to provide a reliable composition [28]. In this section, we use workflow patterns to describe TCS's control flow model as a composition pattern. Afterwards, we extend them in order to specify TCS's transactional flow, in addition to the control flow they are considering by default. Indeed, the transactional flow is tightly related to the control flow. The recovery mechanisms (defined by the transactional flow) depend on the execution process logic (defined by the control flow).

The use of the recovery mechanisms described throw the transactional behavior varies from one pattern to another. Thus, the transactional behavior flow should respect some consistency rules(INVARIANT) given a pattern. These rules describe the appropriate way to apply the recovery mechanisms within the specified patterns. Recovering properly a failed composite service means: trying first an alternative to the failed component service, otherwise canceling ongoing executions parallel to the failed component service, and compensating the partial work already done. The transactional consistency rules ensure transactional consistency according to the context of the used pattern. In the following we formally specify these patterns and related transactional consistency rules using Event-B.

Our model introduces a new context *And-patternContext* which extends the context *CompositionContext* and a machine *transactionalpatterns* which refines the machine *Com-*

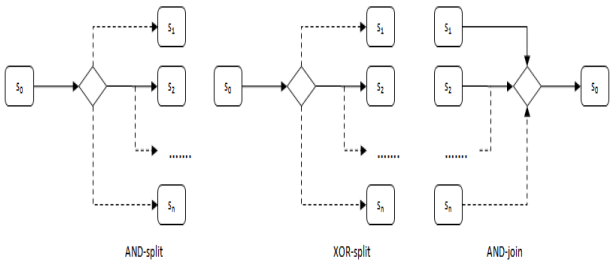

Figure 3. Studied patterns

positionMachine. To extend these patterns we introduce new events that can describe them. For example, to extend the pattern AND-split the machine introduces a new event *AND-split* which defines the pattern AND-split. Due to the lack of space, we put emphasis on the following three patterns AND-split, AND-join and XOR-split to explain and illustrate our approach, but the concepts presented here can be applied to other patterns.

An AND-split pattern defines a point in the process where a single thread of control splits into multiple threads of control which can be executed in parallel, thus allowing services to be executed simultaneously or in any order. The SWToutside represent the set of services (s_1,..,s_n) and s_0 is represented by S0.

AND-split $\triangleq$
ANY
$S0$
$SWToutside$
WHERE
grd1 : $SWToutside \subset SWT\_AS$
grd2 : $S0 \in SWT\_AS \setminus SWToutside$
grd3 : $service\_state(S0) = completed$
THEN
act1 : $SWToutstate := activated$
END

The Event-B formalization of this pattern indicates that all *SWToutside* services will be activated when S0 is successfully completed and this is ensured by adding a theorem indicating that *SWToutstate* is activated is equal to all the services from this subset is in the active state. The *SWT_AS* subset represents the AND-split services and covers all *SWToutside* and $S0$ services.

To verify the transactional consistency of these patterns we add predicates in the INVARIANT clauses. These invariants ensure transactional consistency according to the context of use. These rules are inspired from [29] which specifies and proves the potential transactional dependencies of workflow patterns. The transactional consistency rules of the AND-split pattern support only compensation dependencies from *SWToutside* (Inv 23).

- Inv 23: $\forall s.s \in SWToutside \Rightarrow sAS \mapsto s \notin depCOMP$

The compensation dependencies can be applied only over already activated services. The transactional consistency rules supports only cancellation dependencies between only the concurrent services. Any other cancellation or alternative or compensation dependencies between the pattern's services (Inv 11, 12) are forbidden.

- Inv 11: $\forall s.s \in SWT\_AS \Rightarrow s \mapsto sAS \notin depANL$
- Inv12: $\forall s, s1.s \in SWT\_AS \wedge s1 \in SWT\_AS \Rightarrow s \mapsto s1 \notin depAL$

Our example illustrates the application of AND-split pattern to the set of services (SCN, HR, FB) and specifies that exist a dependency of compensation from HR to FB and a cancellation dependency also from HR to FB. The guard of the AND-split event represents the conditions of activation of the pattern. In our example SCN must terminates its work before activating the pattern. In order to ensure a normal execution of the event an invariant must be preserved by AND-split event that express that all *SWToutside* services have an activation dependency from S0

- Inv 13: $\forall s.s \in SWToutside \Rightarrow sAS \mapsto s \in depA$

An AND-join pattern defines a point in the process where multiple parallel subprocesses/services converge into one single thread of control, thus synchronizing multiple threads. To extend the pattern AND-join, the machine introduces a new event *AND-join* which defines the control flow of the AND-join pattern.

AND-join $\triangleq$
ANY
$S0$
$SWToutside$
WHERE
grd1 : $SWToutside \subset SWT\_AJ$
grd2 : $S0 \in SWT\_AJ \setminus SWToutside$
grd3 : $\forall s.s \in SWToutside \Rightarrow service\_state(s) = completed$
THEN
act1 : $service\_state(S0) := active$
END

The Event-B formalisation of this pattern indicates that $S0$ will be activated after the termination with success of all *SWToutside* services. The *SWT_AJ* subset represents the AND-join services and covers all *SWToutside* and $S0$ services.

The transactional consistency rules of the AND-join pattern supports only compensation dependencies for *SWToutside*, $S0$ can not be compensated by *SWToutside* services as they are executed after (inv 24).

- Inv 24: $\forall s.s \in SWToutside \Rightarrow s \mapsto S0 \in depCOMP$

The transactional consistency rules of the AND-join pattern support also cancellation dependencies between only the

concurrent services. Any other cancellation or alternative or compensation dependencies between the pattern's services are forbidden.

- Inv25: $\forall s.s \in SWToutside \Rightarrow s \mapsto S0 \in depANL$

Our example illustrates the application of AND-join pattern to the set of services (HR, FB, OP). The guard of the AND-join event represents the conditions of activation of the pattern. HR and FB must terminates its work before activating the pattern. The termination of HR is necessary and not efficient to activate the pattern. All *SWToutside* , HR and FB, services must complete their work.

An XOR-split pattern defines a point in the process where, based on a decision or control data, one of several branches is chosen. To extend the pattern XOR-split, the machine introduces a new event *XOR-split* which defines the pattern XOR-split.

XOR-split $\triangleq$
ANY
$S0$
$SWToutside$
sw
WHERE
grd1 : $SWToutside \subset SWT\_XS$
grd2 : $S0 \in SWT\_XS \setminus SWToutside$
grd3 : $service\_state(S0) = completed$
grd4 : $sw \in SWToutside$
THEN
act1 : $service\_state(sw) := active$
END

The Event-B formalization of this pattern indicates that *sw* will be activated after the termination of $S0$. The *SWT_XS* subset represents the XOR-split services and covers all *SWToutside* and $S0$ services.

The XOR-split pattern supports alternative dependencies between only the services *SWToutside*, as the alternative dependencies can exist only between parallel and non concurrent flows. The XOR-split pattern support also compensation dependencies from *SWToutside* to *sXS*.

- Inv18: $\forall s.s \in SWT\_XS \setminus sXS \Rightarrow s \mapsto s0 \in depCOMP$

Any other cancellation or alternative or compensation dependencies between the pattern's services are forbidden.

- Inv15: $\forall s.s \in SWT\_XS \Rightarrow s \mapsto s0 \notin depAL$
- Inv22: $\forall s.s \in SWT\_XS \setminus sXS \Rightarrow s0 \mapsto s \in depCOMP$

Our example illustrates the application of XOR-split pattern to the set of services (OP, SDD, SDF, SDF) and specifies that exist an alternative dependency from HR to FB. The guard of the XOR-split event represents the conditions of activation of the pattern. The execution of OP service must be completed for activate XOR-split pattern. After the activation one service from (SDD, SDF, SDF) will be active.

VI. Validation

The hierarchy of web services model obtained by the development process described in the last two sections contains the different contexts and specific machine model. We present it in three levels:

- The first level expresses the transactional behavior of web services in terms of events and states.
- The second level represents the combinations of a set of services to offer a new value-added service. It introduces the dependency concept between services to express the relation that can exist between services and expresses how the behavior of certain services influences the behavior of other services.
- The third level presents the concept of composition patterns and introduces two machines and contexts. We extend them in order to specify TCS's transactional flow. We add transactional consistency rules in INVARIANTS clause to check the consistency of used patterns.

In the previous section, we showed how to formally specify a TCS using Event-B. The objective of this section is to show how we verify and validate our model using proof and ProB animator.

In the abstract model the desired properties of the system are expressed in a predicate called invariant, it has to prove the consistency of this invariant compared to system events by a proof. We find many proof obligations (Figure 4). Each of them has got a compound name for example, "evt / inv / INV". A green logo situated on the left of the proof obligation name states that it has been proved (an A means it has been proved automatically).

Axm1: $SWT = \{SCN, HR, FB, OP, SDD, SDF, SDT\}$

INITIALISATION $\triangleq$
$service\_state = \{SCN \mapsto initial, HR \mapsto initial, FB \mapsto initial, OP \mapsto initial, SDF \mapsto initial, SDD \mapsto initial, SDT \mapsto initial\}$
$SWT\_C = \{SCN, FB\}$
$SWT\_P = \{OP, SDT\}$
$SWT\_AS = \{SCN, HR, FB\}$
$SWT\_AJ = \{HR, FB, OP\}$
$SWT\_XS = \{OP, SDD, SDF, SDT\}$
$depA = \{SCN \mapsto FB, SCN \mapsto HR, HR \mapsto OP, FB \mapsto OP, OP \mapsto SDF, OP \mapsto SDD, OP \mapsto SDT\}$
$depCOMP = \{HR \mapsto FB, HR \mapsto SCN\}$
$depAL = \{SDF \mapsto SDD\}$
$depANL = \{HR \mapsto FB\}$
END

In our case shown in Figure 4 the tool generates the following proof obligations ”activate / inv1 / INV” and ”compensate / inv1 / INV”. This proof obligation rule ensures that the invariant inv1 in the CompositionMachine is preserved by events activate and compensate. Figure 4 show also the proof obligations ”compensate / grd2 / WD”. This proof obligation rule ensures that a potentially ill-defined guard is indeed well defined.

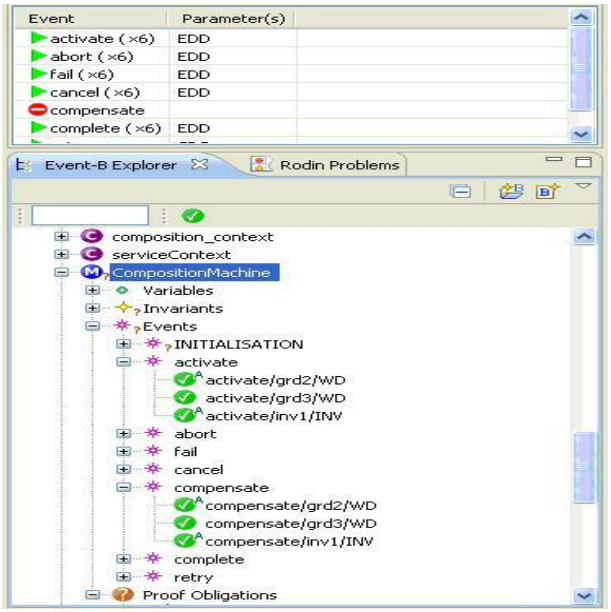

Figure 4. Proof obligations

Our work is proof oriented and covers the transactional web services. All the Event-B models presented in this paper have been checked within the RODIN platform. The proof based approaches do not suffer from the growing number of explored states. However, the proof obligations produced by the Event-B provers could require an interactive proof instead of automatic proofs.

Concerning the proof process within the Event-B method, the refinement of transactional web services Event-B models can be performed. This refinement allows the developer to express the relevant properties at the refinement level where they are expressible. The refinement is a solution to reduce the complexity of proof obligations.

In our example the designer can initially specify, as CS transactional behavior, that FB will be compensated or cancelled if HR fails, SDD is executed as alternative of SDF failure. The Event-B formalization of our motivating example defines a cancellation dependency and compensation dependency from HR to FB and alternative dependency from SDF to SDD.

For example, by checking the compensation dependency between SCN and HR the RODIN platform mentioned that the proof obligations has not been discharged (Figure 5). As HR is executed after, it can not exist a compensation dependency from SCN to HR. A red logo with a ”?” appear in the proof tree and it means that is not discharged. This basic example shows how it is possible to formally check the consistency of transactional flow using Event-B. To repair this error we can refer to the initialization of the machine and verify the compensation dependencies. After

Figure 5. A red logo indicates that the proof obligations is not discharged

the initialization of the *ServiceMachine* the compensate event is disabled and after the termination of the execution of a service the event will be enabled. ProB offer to the developer which parameter is used in the animation by clicking right on the event (Figure 6). In the development

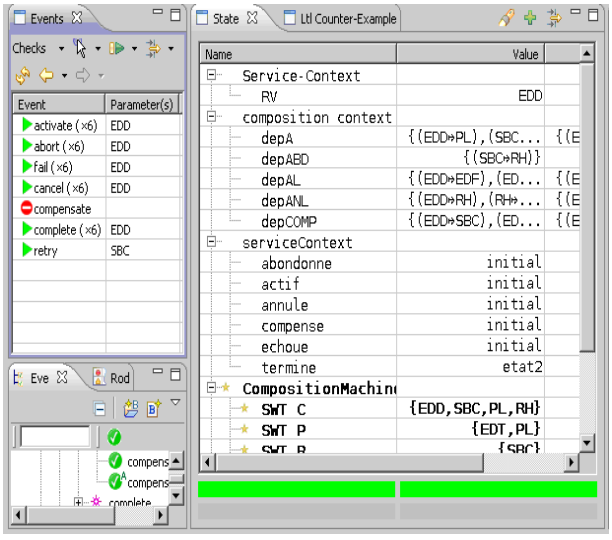

Figure 6. Animation with the ProB animator

of our model some proof obligations are not discharged but the specifications is correct according to our work in [20] which is specified and validated using Event Calculus. To do so, we use ProB animator to verify our specification of

transactional web services. This case study has shown that the animation and model-checking are complementary to the proof, essential to the validation of Event-B models. In other case, many proved models (proof obligations are discharged) still contain behavioral faults, which are identified with the animators. The main advantage of Event-B develop that can repair errors during the development. It allows the backward to correct specification. With refinement, the complexity of the system is distributed; the step by step proofs are more readily. Event-B offers more flexibility and expressivity than the input languages of model checkers.

VII. Conclusion

The paper addresses the formal specification, verification and validation of the transactional behavior of services compositions within a refinement and proof based approach. The described work uses Event-B method, refinement for establishing proprieties. This paper presented our model of Web service, enriched by transactional properties to better express the transactional behavior of web services and to ensure reliable compositions. Then we describe how we combine a set of services to establish transactional composite service by specifying the order of execution of composed services and recovery mechanisms in case of failure. Finally we introduced the concept of composition pattern and how we uses it to specify a transactional composite service.

In our future works we are considering the following perspectives:

- Using automation approach of MDE type to verify transactional behavior of services compositions.
- We extend this work to consider the dynamic evolution of a composite service. By controlling the dynamic of a composition, we preserve the architectural and comportemental properties of a composite service during its evolution and not lead configurations that may damage the operation of the composite service.

References

[1] R. Maraoui, M. Graiet, M. Kmimech, M.T. Bhiri and B. Elayeb, *Formalisation of protocol mediation for web service composition with ACME/ARMANI ADL*, Service Computation IARIA 2010-Lisbon-Portugal, Nov 2010.

[2] J.R. Abrial, *The B Book: Assigning programs to meanings*, Cambridge University Press, 1996.

[3] J.R. Abrial, *Modeling in Event-B: System and Software Engineering*, cambridge edn. Cambridge University Press, 2010.

[4] J.R. Abrial, M. Butler and S. Hallerstede, *An open extensible tool environment for Event-B*, ICFEM06, LNCS 4260, Springer, p. 588-605, 2006.

[5] M. Dumas and M.C. Fauvet, *Les services web. intergiciel et construction d'applications reparties*, ICAR, 2006.

[6] L.P. Cabrera, G. Copeland, M. Feingold, R.W. Freund, T. Freund, J. Johnson,S. Joyce, C. Kaler, J. Klein, D. Langworthy, M. Little, A. Nadalin, E. Newcomer, D. Orchard, I. Robinson, J. Shewchuk, and T. Storey. *Web servicescoordination(ws-coordination)*, 2005.

[7] L.P. Cabrera, G. Copeland, M. Feingold, R.W. Freund, T. Freund, J. Johnson,S. Joyce, C. Kaler, J. Klein, D. Langworthy, M. Little, A. Nadalin, E. Newcomer, D. Orchard, I. Robinson, T. Storey, and S. Thatte. *Web services atomic transaction (wsatomictransaction)*, 2003.

[8] L.P. Cabrera, G. Copeland, M. Feingold, R.W. Freund, T. Freund, S. Joyce, J. Klein, D. Langworthy, M. Little, F. Leymann, E. Newcomer, D. Orchard, I. Robinson, T. Storey, and S. Thatte. *Web services business activity framework(ws-businessactivity)*, 2003.

[9] R. Hamadi and B. Benatallah, *A petri net-based model for web service composition*. Fourteenth Australasian Database Conference (ADC2003), 2003.

[10] G. Salaun, A. Ferrara, and A. Chirichiello, *Negotiation among web services using lotos/cadp*. European Conference on Web Services (ECOWS 04), 2004.

[11] M. Rouached, W. Gaaloul, W.M.P. van der Aalst, S. Bhiri, and C. Godart, *Web service mining and verification of properties: An approach based on event calculus*. OTM Confederated International Conferences, 2006.

[12] H. Foster, S. Uchitel, J. Magee, and J. Kramer. *Model-based verification of web service compositions*, IEEE Automated Software Engineering (ASE), 2003.

[13] M. Graiet, R. Maraoui, M. Kmimech, M.T. Bhiri and W. Gaaloul, *Towards an approach of formal verification of mediation protocol based on Web services*, 12th International Conference on Information Integration and Web-based Applications & Services (iiWAS2010), Paris-France, November 2010.

[14] D. Garlan, R. Monroe and D. Wile, *ACME: Architectural Description of Component-Based Systems*. Foundations of Component-Based Systems, Leavens G.T, and Sitaraman M. (Eds.), Cambridge University, Press, 2000.

[15] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1998.

[16] D. Garlan, R. Monroe and D. Wile, *ACME: Architectural Description of Component-Based Systems. Capturing software architecture design expertise with Armani*, Technical Report CMU-CS-98–163, Carnegie Mellon University School of Computer Science, 2001.

[17] I. Ait-Sadoune and Y. Ait-Ameur, *From BPEL to Event-B, International Workshop on Integration of Model-based Methods and Tools* IM FMT'09 at IFM'09 Conference, Dsseldorf Germany, Fevruary , 2009.

[18] A. K. Elmagarmid, Ed., *Database transaction models for advanced applications*, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc., 1992.

[19] W. M. P. van der Aalst and K. M. van Hee, *Workflow Management: models, methods and tools*, ser. Cooperative Information Systems, J. W. S. M. Papazoglou and J. Mylopoulos, Eds. MIT Press, 2002.

[20] W. Gaaloul, S. Bhiri and M. Rouached, *Event-Based Design and Runtime Verification of Composite Service Transactional Behavior*, IEEE Transactions on Services Computing, 02 Feb. 2010, IEEE computer Society Digital Library, IEEE Computer Society.

[21] M. Leuschel and M. Butler, *ProB: A Model Checker for B* , in K. Araki, S. Gnesi, D. Mandrioli (eds), FME 2003: Formal Methods, LNCS 2805, Springer-Verlag, pp. 855-874, 2003.

[22] C. Metayer, J. Abrial, and L. Voisin , *Event-B Language. Technical Report D7*, RODIN Project Deliverable, 2005.

[23] L. Jemni Ben Ayed and F. Siala, *Event-B based Verification of Interaction Properties In Multi-Agent Systems*, in Journal of Software, Vol 4, No 4 (2009), pp.357-364, Jun 2009.

[24] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz, *A transaction model for multidatabase systems*, in ICDCS, pp. 56-63, 1992.

[25] B. Medjahed, B. Benatallah, A. Bouguettaya, A. H. H. Ngu and A. K. Elmagarmid, *Business-to-business interactions: issues and enabling technologies*, The VLDB Journal, vol. 12, no. 1, pp. 59-85, 2003.

[26] W. M. P. van der Aalst, A. P. Barros, A. H. M. ter Hofstede and B. Kiepuszewski, *Advanced Workflow Patterns* in 5th IFCIS Int. Conf. on Cooperative Information Systems (CoopIS'00), ser. LNCS, O. Etzionand P. Scheuermann, Eds., no. 1901. Eilat, Israel: Springer-Verlag, September 6-8, pp. 18-29, 2000.

[27] W. M. P. van der Aalst and A. H. M. ter Hofstede,*Yawl: yet another workflow language* Inf. Syst., vol. 30, no. 4, pp. 245-275, 2005.

[28] S. Bhiri, C. Godart and O. Perrin, *Transactional patterns for reliable web services compositions*, in ICWE, D. Wolber, N. Calder, C. Brooks, and A. Ginige, Eds. ACM, pp. 137-144, 2006.

[29] S. Bhiri, O. Perrin and C. Godart, *Extending workflow patterns with transactional dependencies to define reliable composite web services*, in AICT/ICIW. IEEE Computer Society, p. 145, 2006.