A Proof-based Approach for Verifying Composite Service Transactional Behavior

Lazhar Hamel
MIRACL, ISIMS, TUNISIA
lazhar.hamel@gmail.com

Mohamed Graiet
MIRACL, ISIMS, TUNISIA
mohamed.graiet@imag.fr

Mourad Kmimech
MIRACL, ISIMS, TUNISIA
mkmimech@gmail.com

Mohamed Tahar Bhiri
MIRACL, ISIMS, TUNISIA
Tahar_bhiri@yahoo.fr

Walid Gaaloul
Computer Science Department Télécom SudParis
walid.gaaloul@it-sudparis.eu

***Abstract*— Web services are software components accessible via Internet. Web services are defined independently from any execution context. A key challenge of Web service compositions 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. In this work, we propose an approach to formalize a model of Web services composition to check and ensure reliable execution. To achieve this, we propose a proof oriented approach for the formalization and verification of transactional behavior of web services composition using Event-B.**

Keywords-web service composition; Event-B; transactional web service; proof; verification.

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. The problem, which we are interested in, is how to ensure reliable web services compositions. By reliable, we mean a composition where all the executions are correct.

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. 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 the goal of which is to cancel the effect of other transaction that failed to be successfully completed. several transactions models previously proposed in databases, distributed systems, collaborative environments. In order to manage with this focus many specifications proposed to response to this aspects. WS-Coordination [1], WS-AtomicTransaction [2] and WS-BusinessActivity [3]. Many research in this field aiming for instance to guarantee that an activity is cancellable 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 [4], process algebras [5], or the temporal theories [6].

Our work deal 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 [7] used in the RODIN platform [8]. 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 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-

 ISBN: 978-1-61208-165-6

B method are used to explore it and in section 5 we discuss this approach.

This paper is organized as follows. In Section 2, we introduce a motivating example. Section 3 presents the Event- B method, its formal semantics and its proof procedure and introduces our transactional CS model. In Section 4, 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 5.

II. 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.

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.

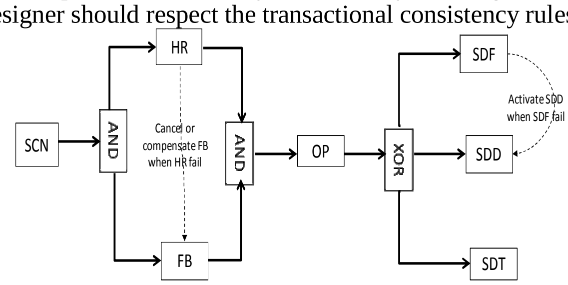

Figure 1. Motivating example

III. 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 [9], 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.

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.

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 modeled. 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[10].

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

An Event-B model M with invariants I is well-formed, denoted by M |= 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)}[11].

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 *TWS* is defined in the SETS clause which represents all transactional web services.

```
CONTEXT ServiceContext
SETS
SWT
STATES

AXIOMS
Axm1: STATES= {active, initial, aborted, cancelled, failed, completed, compensated}
```

The service state which is represented by a functional relation *service_state* defined in VARIABLES clause gives the current state of such a service. 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.

```
MACHINE ServiceMachine
SEES ServiceContext
VARIABLES
Service_state
SWT_C
SWT_P
SWT_R
INVARIANTS
Inv1: service_state∈SWT→STATES
Inv2: SWT_C ⊆ SWT
Inv3: SWT_R ⊆ SWT
Inv4: SWT_P ⊆ SWT
```

In our model, transitions are described by the event. For instance the event activate changes the status of a service and pass it from initial status to active. The event compensate enables to compensate semantically the work of a service and pass it from completed status to compensate. The event retry changes the status of a service and activate it after his failure and pass it from failed status to active. The event complete enables to finite the execution of a service with success and pass it from active status to completed.

```
Activate ≙ ANY s WHERE
grd1 : s∈SWT
grd2 : service_state (s) =initial
THEN
act1 : service_state (s):=active
END
Compensate ≙ANY s WHERE
grd1 : s∈SWT_C
grd2 : service_state (s) =completed
THEN
act1 : service_state (s):=compensated
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 [12]. 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*...).

```
MACHINE CompositionMachine
REFINES ServiceMachine
SEES CompositionContext
```

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.

```
Axm1 : depA∈SWT↔SWT
Axm2 : depAL∈SWT↔SWT
…
```

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

 ISBN: 978-1-61208-165-6

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. Alternative dependencies allow us to define forward recovery mechanisms. A compensation dependency allows us to define a backward recovery mechanism by compensation. A cancellation dependency allows us to signal a service execution failure to other service(s) being executed in parallel by canceling their execution. Activation dependencies 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 in 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.

```
Compensate ≙ REFINES Compensate
…
grd4:∃s0·s0∈SWT∧s0↦s∈depCOMP⇒((service_state(s0)=failed)
∨ (service_state(s0) = compensated))
THEN
act1  :   service_state (s) ≔compensated
END
```

IV. TRANSACTIONAL SERVICE PATTERNS

The use of workflow patterns [13] 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). 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 [14]. 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.

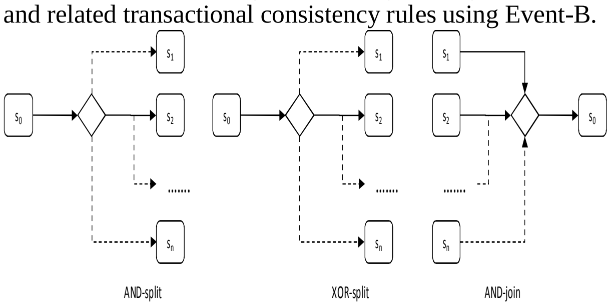

Figure 2. Studied patterns

Our model introduces a new context *And-patternContext* which extends the context *Composition-Context* and a machine transactional patterns which refines the machine *CompositionMachine*. 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.

```
AND-split ≙
ANY
S0
SWToutside
WHERE
grd1  :   SWToutside⊆SWT_AS
grd2  :   S0∈SWT_AS
grd3  :   S0∉SWToutside
grd4  :   service_state(S0)=complete
THEN
act1  :   stateSWTout≔activated
END
```

The SWToutside represent the set of services ($s_1,\ldots,s_n$) and s0 is represented by sAS.

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 [14] 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 *sAS*

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

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  ≙
ANY
S0
SWToutside
WHERE
grd1  :   SWToutside⊆SWT_AJ
grd2  :   S0∈SWT_AJ
grd3  :   S0∉SWToutside
grd4  :   ∀s•s∈SWToutside⇒service_state(s)=complete
THEN
act1  :   service_state(S0) ≔ active
END
```

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.

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

- Inv 24: $\forall s.s \in SWToutsideAJ \Rightarrow s \mapsto sAJ \notin depCOMP$

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: $\forall s.s \in SWToutsideAJ \Rightarrow s \mapsto sAJ \notin depANL$

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  ≙
ANY
S0
SWToutside
sw
WHERE
grd1  :   SWToutside⊆SWT_XS
grd2  :   S0∈SWT_XS
grd3  :   S0∉SWToutside
grd4  :   service_state(S0)=complete
grd5  :   sw∈SWToutside
THEN
act1  :   service_state(sw)≔active
END
```

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: $\forall s.s \in SWT\_XS \setminus \{sXS\} \Rightarrow s \mapsto sXS \in depCOMP$

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

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

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.

V. VALIDATION

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[15]. 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 3). 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). In our case shown in Figure 3 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.

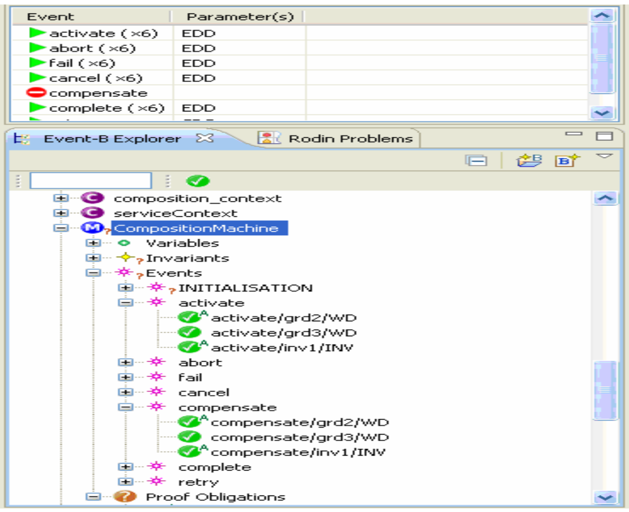

Figure 3. Proof obligations and animation

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 4). 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 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.

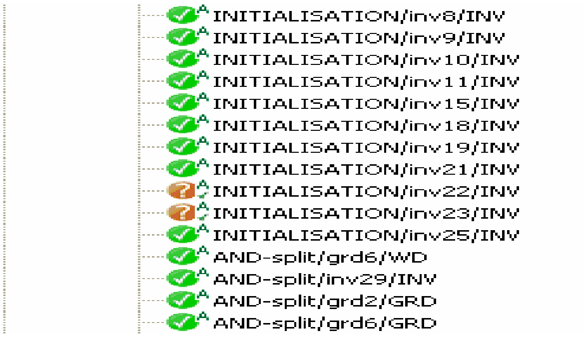

Figure 4. A red logo indicates that the proof obligations is not discharges

In the development of our model some proof obligations are not discharged but the specifications is correct according to our work in [6] 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.

VI. CONCLUSION AND FUTURE WORKS

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 presents 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

 ISBN: 978-1-61208-165-6

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.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

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

[5] G. Sala¨un, A. Ferrara, and A. Chirichiello, “Negotiation among web services using lotos/cadp,” European Conference on Web Services (ECOWS 04), 2004.

[6] 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.

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

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

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

[10] C. Metayer, J. Abrial, and L. Voisin , “Event-B Language. Technical Report D7,” z RODIN Project Deliverable, 2005.

[11] S. Mehrotra, R. Rastogi, H. F. Korth, and A. Silberschatz, “A transaction model for multidatabase systems.” in ICDCS, 1992, pp. 56–63.

[12] 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.

[13] 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, 2000, pp. 18–29.

[14] 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, 2006, pp. 137–144.

[15] 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.