Symbolic model checking composite Web services using operational and control behaviors

Jamal Bentahar a,b,⁎, Hamdi Yahyaoui c, Melissa Kova d, Zakaria Maamar e

a Concordia University, Concordia Institute for Information Systems Engineering, Montreal, Canada
b Khalifa University of Science, Technology and Research, College of Engineering, Abu Dhabi, United Arab Emirates
c Kuwait University, Computer Science Department, Kuwait
d Ubisoft Divertissement, Montreal, Canada
e Zayed University, College of Information Technology, Dubai, United Arab Emirates

A R T I C L E  I N F O

Keywords:
Composite Web service
Behavior
Soundness
Completeness
Symbolic model checking
NuSMV

A B S T R A C T

This paper addresses the issue of verifying if composite Web services design meets some desirable properties in terms of deadlock freedom, safety (something bad never happens), and reachability (something good will eventually happen). Composite Web services are modeled based on a separation of concerns between business and control aspects of Web services. This separation is achieved through the design of an operational behavior, which defines the composition functioning according to the Web services’ business logic, and a control behavior, which identifies the valid sequences of actions that the operational behavior should follow. These two behaviors are formally defined using automata-based techniques. The proposed approach is model checking-based where the operational behavior is the model to be checked against properties defined in the control behavior. The paper proves that the proposed technique allows checking the soundness and completeness of the design model with respect to the operational and control behaviors. Moreover, automatic translation procedures from the design models to the NuSMV model checker’s code and a verification tool are reported in the paper.

© 2012 Elsevier Ltd. All rights reserved.

1. Introduction

Web services are widely used for developing Business-to-Business applications whose performance spreads by default over organizations’ boundaries. Indeed, Web services rely on a set of platform-independent and vendor-neutral specifications that offer the necessary means for their description, discovery, invocation, and mainly composition (Yang, Zhang, & Lan, 2007). Despite the great interest of the research and industry communities in Web services, some challenging issues remain pending and hence, hinder the adoption of Web services to develop robust, dynamic, and safe business applications. Examples of issues include verification and model checking (Lomuscio, Qu, & Solanki, 2008, 2011; Rouached, Fdhila, & Godart, 2010; Yeung, 2011), reliability (Bhiri, Perrin, & Godart, 2005; Zhang & Zhang, 2005), security (Yahyaoui, 2012), transaction handling (Zhao, Kart, Moser, & Melliar-Smith, 2008), service discovery (Fardin, Naser, & Ali, 2012; Tian & Huang, 2012), and last but not least context awareness of interaction management (Handorean, Sen, Hackmann, & Roman, 2006). The severity of these issues intensifies when several component Web services are put together to form composite Web services (Benslimane, Maamar, & Ghedira, 2006; Maamar, Benslimane, Cherida, & Mrissa, 2005).

In this paper, we address the “thorny” issue of verifying the design of composite Web services. Developing business applications that end-users trust requires a deep investigation of the different and independent operations and behaviors that the component Web services in a composition execute and exhibit, respectively. For instance, having a deadlock in a Web service-based business transaction is a simply disaster for all stakeholders. Although software vendors can guarantee the safety of their Web services, the development, testing, and verification of these Web services are done independently from other vendors’ peers, which means a serious lack of how these Web services behave when put together. To tackle this problem, we use model checking, a powerful formal and fully automatic technique for the verification of system models against specified properties in an early stage in the system lifecycle. Compared to other verification techniques such as code reviewing and testing, model checking is the only technique that provides a formal proof and thus a guarantee that the system is sound with respect to the checked properties. Using this technique, the Web services composition model, which captures the behavior

* Corresponding author. Address: Concordia University, Concordia Institute for Information Systems Engineering, 1515 Ste-Catherine Street West, EV7.640, Montreal, Quebec, Canada H3G 2W1.
E-mail addresses: bentahar@ciise.concordia.ca (J. Bentahar), hamdi@sci.kuniv.edu.kw (H. Yahyaoui), melissa.kova@ubisoft.com (M. Kova), zakaria.maamar@zu.acae (Z. Maamar).
0957-4174/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.eswa.2012.07.069
of this composition, can be investigated to check if it satisfies desirable properties such as reachability and safety, and does not satisfy undesirable properties such as deadlock. The problem of model checking is formally denoted by $M \models \varphi$, where $M$ is the system model, $\varphi$ a property, and $\models$ the satisfaction symbol, meaning that the model $M$ satisfies the property $\varphi$. In model checking approaches (Cimatti et al., 2002; Clarke, Grumberg, & Peled, 1999), $M$ is represented in a formal language capturing the system's dynamics and $\varphi$, against which the model is checked, is expressed in a given logic such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) (Emerson, 1990). The verification technique examines all possible executions of the model. In this paper, we focus on a particular technique of model checking referred to as symbolic. In symbolic model checking, the model and properties are not explicitly represented as automata, but implicitly and symbolically encoded using Boolean functions and binary decision diagrams, which alleviates the state explosion problem, the main drawback of automata-based techniques (Clarke et al., 1999).

As pointed out in many proposals, for instance (Meng & Arbab, 2007; Peltz, 2003; Zhou, Huang, & Wang, 2007), two composition approaches dominate service-oriented computing: choreography and orchestration. On the one hand, choreography identifies the set of acceptable conversations for a composite Web service without allocating a specific Web service to monitor the composition progress. It captures the global view of the composition by focusing on the coordination and multi-participant perspectives. On the other hand, orchestration is an executable specification that identifies the steps of execution for the peers where one particular party monitors the execution of such a specification. It captures the local view of the composition from one participant perspective. Although the approach we propose in this paper is choreography-oriented because it stresses the coordination and conversation among Web services in a global perspective, it is still general enough and can be applied to orchestration as well, as far as the design model, which is needed for model checking, is fully available.

To better model compositions, we build upon the idea of separating concerns that model, at different levels of abstractions, two different behaviors of Web services: operational and control (Yahyaoui et al., 2010). From the perspective of single Web services, the operational behavior is application-dependent; it defines the business logic underpinning a Web service operation at a low level of abstraction by specifying the functions the Web service should perform, and the control behavior is application-independent; it defines at a high level of abstraction a general design pattern of the Web service and guides and monitors the execution progress of the operational behavior by identifying the actions to take and constraints, if any, to satisfy. Compared to design patterns, the control behavior is not implementable, so not directly transformable to executable code, but helps design the operational behavior, which is implementable. These two behaviors implement the separation of concerns between the business logic details and the abstract control logic of a Web service. The motivation behind such a separation is to enhance the current design practices of Web services that are prone to errors since the architectural and operational perspectives are treated at the same level, which increases the design and management complexity. From the design pattern perspective, the control behavior, when synchronized with the operational behavior, guarantees that design requirements and constraints are considered and good practices are being used. Dissociating the operational behavior from the control behavior permits to assess at design time the impact of any change in the business logic on the control behavior as shown in Sheng et al. (2010). Thus, any change in the operational behavior that does not impact the synchronization with the control behavior is allowable as it reflects the continuous fulfillment of the design pattern. The control behavior controls the flexibility of the business logic and serves as a tool for analyzing the sensibility of the Web service to design changes. For example, if a new business rule has to be incorporated in the Web service's functionality, this will not affect the control behavior. The main advantage of such a practice is to provide service engineers with a means to monitor the execution of the Web service and identify and address design problems at an early stage.

In a previous work (Yahyaoui et al., 2010), the control and operational behaviors have been thoroughly investigated in the context of individual Web services, i.e., isolated from other peers in the same composition. The authors generalized this separation to the composite scenarios extending the separation framework proposed in Kova, Bentahar, Maamar, and Yahyaoui (2009). Unlike the control behavior of isolated Web services that focuses on the internal behavior of an individual Web service, the control behavior of a composition focuses on the interactions among different Web services. The control behavior of a composition shapes the design pattern of this composition; it is application independent and hence, applicable to a wide range of compositions of Web services. Its objective is to frame, control, and monitor the business logic execution as it provides the guidelines for an appropriate composition behavior. Based on the principle of separation of concerns (Kambayashi & Ledgard, 2004), this behavior is designed without updating the operational behavior that captures the business logic of the composition. This behavior facilitates the reusability of composition scenarios as it is independent from any specific business case. The operational behavior describes the business logic of a specific composite scenario by identifying the functions to perform and messages to exchange among the components, while the control behavior identifies, in an application-independent perspective, the valid actions and sequences that the operational behavior should follow. Similar to the case of isolated Web services, if a new composition constraint has to be added to the composition business logic, the control behavior will not be affected, but used to enable or disable such a constraint. In this paper, the formal machinery of this composition-oriented separation of behaviors is defined in order to be automatically verified. This automatic verification raises fundamental challenges. For instance, the control behavior of a Web service can be in interaction with the operational behavior of another one, which is involved in the same composition. This leads to the need of specifying valid interactions. We map the control behavior onto the operational behavior in order to verify the soundness and completeness of the composition process if the two behaviors are synchronized. We use symbolic model checking technique to implement this verification by checking if the model of the operational behavior satisfies the properties generated from the control behavior.

The contributions of this work are twofold:

1. Proposal of an automata-theoretic approach for modeling composite Web services based on control and operational behaviors. Both are linked together to check that the conversation sequences of the operational behavior, which implement the business logic, are synchronized with the valid sequences, called executions, specified by the control behavior (soundness checking) and vice versa (completeness checking). We use statecharts enhanced with additional syntax to facilitate the mapping process between both behaviors.

2. Formal and automatic verification of the mapping procedure using symbolic model checking techniques. The implementation is done using a Java-based translation procedure, which is proven to be sound, and NuSMV model checker (Cimatti et al., 2002).
Paper Overview. The paper is organized as follows. In Section 2, we discuss the modeling and formalization of Web services composition. In Section 3, we introduce the control and operational behaviors, specify their formal machinery, and define the concepts of soundness and completeness of Web services composition. A concrete example is also provided, which helps understand the proposed verification framework. In Section 4, we present our model checking-based approach and prove that the soundness and completeness can be checked automatically using this approach. Furthermore, theoretical results concerning the translation of the design model into a verifiable formal model are provided. Furthermore, verification results of the considered example using the NuSMV model checker are discussed. In Section 5, we describe the Java-based implemented tool and show relevant statistical results. Section 6 compares our proposal with related work and finally, Section 7 concludes the paper.

2. Modeling and formalizing Web services composition

When a user’s request cannot be satisfied by any single, available Web service, the composition of Web services is the alternative. The automatic composition of Web services has been studied using different techniques such as logical inference (Tang, Jiang, & Zhou, 2011), Petri nets (Yoo, Jeong, & Cho, 2010), and event calculus (Okutan & Cicelki, 2010). In this paper, we define a composition of Web services and a component Web service as follows.

Definition 1 (Composition). The composition of a set of component Web services is a 5-tuple: \( CW = \langle W, w_0, F_w, L_w, T_w \rangle \) where:

- \( W \) is the set of component Web services that take part in the composition.
- \( w_0 \) is the initial Web service (from which the composition starts).
- \( F_w \subseteq W \) is the set of final states.
- \( L_w \) is the set of labels used to name the transitions including the empty label \( \varepsilon \).
- \( T_w \subseteq W \times L_w \times W \) is the set of labeled transitions between the component Web services.

Each component Web service \( w_i \in W \) is defined as follows:

Definition 2 (Component Web service). A component Web service \( w_i \in W \) is a 5-tuple: \( w_i = \langle S_i, S_0, F_i, L_i, T_i \rangle \) where:

- \( S_i \) is the set of possible states of the Web service.
- \( S_0 \) is the initial state of the Web service.
- \( F_i \subseteq S_i \) is the set of final states of the Web service.
- \( L_i \) is the set of labels used for the internal transitions.
- \( T_i \subseteq S_i \times L_i \times S_i \) is the set of internal labeled transitions within the Web service.

Basically, a composition is a set of individual Web services that interact with each other using messages. A well-formed sequence of messages leads into a conversation (Bultan, Su, & Fu, 2006; Sheng et al., 2010). Formally, a conversation between \( n + 1 \) Web services \( w_0, \ldots, w_n \) using messages \( a_0, \ldots, a_{n-1} \) is represented with a finite path (i.e., a finite sequence of states and transitions) as follows: \( w_0 \xrightarrow{a_0} w_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-2}} w_{n-1} \xrightarrow{a_{n-1}} w_n \) where \( v \in [0, \ldots, n-1] \) and \( (w_i, a_i, w_{i+1}) \in T_i \) and \( w_n \in F_i \). It is worth noticing that Definition 1 defines compositions by capturing all possible conversations that achieve a final state even if some conversations are not complete (which is the case where a final state is achieved but still messages can be sent in a given composition). In fact, it is the verification part that should check if a conversation is valid by verifying if it satisfies some desirable properties in terms of deadlock freedom (no deadlock in the conversation), safety (something bad never happens), and reachability (or liveness) (something good will eventually happen). In this paper, we demonstrate how model checking, using the operational and control behaviors, is used to check these properties, which are expressed in a logical language. Precisely, we use symbolic model checking (Clarke et al., 1999) and express properties in LTL and CTL. Before introducing our model checking-based verification approach, we define in the next section the conversations between Web services and the synchronization (i.e., the mapping) of their operational and control behaviors. This synchronization characterizes the soundness and completeness of Web services composition.

3. Control and operational behaviors of composite Web services

To define soundness and completeness of a composition, we first discuss the separation of the composition behavior into control and operational. This division is reported in Kova et al. (2009) and Yahyaoui et al. (2010). In this paper, new formal definitions of these behaviors are provided using automata-theoretic concepts. In fact, previous definitions only capture a restricted form of properties, namely those expressed as linear temporal formulas. The new definitions are given so that branching temporal logic formulas can be expressed and verified, as well.

3.1. Control behavior

As a design pattern, the control behavior of a composition of Web services shows the valid execution progress over a set of component Web services. As stressed earlier, the control behavior extracts the properties that a composition, modeled with an operational behavior, should satisfy. Because these properties are expressed in a temporal logic, we formally define the control behavior using alternating Büchi automata (Vardi, 1995). Two reasons motivate the choice of this class of automata. First, alternating Büchi automata are finite structures capturing infinite runs, which are compatible with continuous invocation of Web services. Second, alternating Büchi automata can represent both LTL and CTL formulas, since they consider not only universal choices needed for LTL but also existential choices. Existential and universal choices are captured in alternating Büchi automata because they run on both infinite words (compatible with LTL) and infinite trees (compatible with CTL). Thus, these automata represent properties expressed in linear and branching temporal logics, which hold in some or all computations. Examples of those properties are: (1) in the future of all possible computations starting from the initial state, the user’s request about a given service will be processed; and (2) in all states of some computation, if the service is suspended, there is always possibility of processing the service in some future. Before giving the formal definition of the control behavior, we should introduce the technique that allows representing existential and universal choices.

Let \( S \) be a set of states and \( B(S) \) be the set of all positive Boolean formulas built from elements in \( S \) using the Boolean connectives \( \land \) (and) and \( \lor \) (or). For example, if \( S_1, S_2, S_3 \), and \( S_4 \) are four states in \( S \), \( (S_1 \land S_2) \lor (S_3 \lor S_4) \) is an element of \( B(S) \). The disjunction in this Boolean formula represents the existential choice whereas the conjunction captures the universal choice.

Definition 3 (Control behavior). The control behavior of a composition is a 6-tuple \( CB = \langle M_{cb}, S_0, S_f, F_{cb}, \rho, \nu \rangle \), where:

- \( M_{cb} \) is a finite nonempty set of messages (i.e., the alphabet).
- \( S_0 \) is a finite nonempty set of states.
- \( S_0 \in S_f \) is the initial state.
• $F_Q \subseteq S_Q$ is the set of accepting states.

• $\rho : S_Q \times M_Q \rightarrow B(S_Q)$ is a partial transition function representing universal and existential choices.

• $\zeta : B(S_Q) \rightarrow 2^{|s|}$ is a valuation function associating a set of states with each Boolean formula such that the Boolean formula is evaluated to true if the elements of the associated set of states are evaluated to true.

Let us consider an example. Let $s$ be a state in $S_Q$ and $m_1$ be a message in $M_Q$ such that $\rho(s,m_1) = (s_1 \vee s_2) \wedge (s_2 \vee s_3)$. Then, we have: $\forall s_1 \vee s_2 \vee s_3 = \{s_1, s_2, s_3\}$. Intuitively, the transition $\rho(s,m_1)$ is existentially connecting state $s$ to each element of the state.

The control behavior is defined as a structure that captures valid executions, which are characterized by reaching accepting states from the initial state. Furthermore, as we are interested in capturing and verifying continuous behaviors, we define valid execution in the control behavior over acceptance states $F_Q$ as follows.

**Definition 4 (Valid Execution).** Let $CB$ be a control behavior over $M_Q$. An execution in $CB$ over a sequence of messages $m_0, m_1, \ldots$ from $M_Q$ is a labeled tree (i.e., directed graph) where $s_0$ is the root and each node $s$ at level $i$ has $k$ children $s_1, \ldots, s_k$ if $V(s,m_i) = \{s_1, \ldots, s_k\}$. The execution is said to be valid if it visits infinitely often an element of $F_Q$, the set of accepting states.

This definition is then a generalization of the definition proposed in Yahyaoui et al. (2010) in which a valid execution is a finite branch, which means a finite sequence of states and transitions $s_0 = s_0 \leftarrow s_1 \leftarrow \ldots \leftarrow s_k = s_0$, where $s_0$ is the initial state, $m_0, \ldots, m_k$ are messages and $s_k$ is a final state. We generalize this definition from two perspectives: (1) We consider not only branches but also trees, which means not only existential choices, but also universal ones, are considered; and (2) executions are considered infinite so that not only finite behaviors, but also infinite executions are considered. Consequently, infinite branches having the form $s_0 = s_0 \leftarrow s_1 \leftarrow \ldots \leftarrow s_k = s_0, \ldots$ where an accepting state is being visited infinitely often becomes a special case in our generalized definition.

Fig. 1 depicts the statechart representing the Büchi automaton associated with a control behavior of a composition with two accepting states: $F_Q = \{\text{Done,Compensated}\}$. Initially the composition process is $s_0 = \text{not activated}$. When a user's request is received, the process moves to the received state to reflect the receipt of the request by a component Web service. In case of failure, two options exist: either the process is suspended, which leads into a retrial, or the process is aborted, which leads into a rolling back. This transition is formally represented by $\rho(\text{Received}, \text{Failure}) = (\text{Suspended} \vee \text{Aborted})$. If Suspended is reached, then the following transition takes place: $\rho(\text{Susended}, \text{Retrial}) = (\text{Received} \vee \text{Activated})$, in which case $V(\rho(\text{Suspended}, \text{Retrial})) = \{\text{Received}\}$. When the composition process reaches the received state and no failure occurs, it can invoke a certain component Web service, so that the process moves to the activated state and the execution thread of the composite Web service starts processing. The formal representation of this transition is as follows: $\rho(\text{Received}, \text{Invoke}) = (\text{Processed} \wedge \text{Activated})$ and thus, $V(\rho(\text{Received}, \text{Invoke})) = \{\text{Received, Invoke}\}$. The results are then communicated back from the component Web service. Upon response delivery within an agreed-upon time frame, the composition process resumes to move to the received state. Other requests can be sent to other component Web services and so on. At the end of this process, a component Web service receives the final information and it has to communicate the result back to the client. Thus, when the composition is finalized, the process moves to the done state, which is an accepting state. Compensated state, the second accepting state, could be reached either after failed retrials or after finalizing the composition. The same figure illustrates then a typical composition behavior independently of any specific case study. However, the approach is flexible, so other states and transitions could be added if we are interested in more complicated composition scenarios, including for example transaction and session handling (Handorean et al., 2006; Zhao et al., 2008).

The principles that a designer should follow to define and conceptualize a control behavior of a composition depends on the type and objectives of such a composition. For instance, if the composition should consider transaction handling (Zhao et al., 2008), states reflecting transaction status and transitions capturing the change of this state should be added. Another example is compositions where Web services are allowed to be substituted or replaced if some problems occur. For this type of composition, states representing the status of Web services (for instance, active, asking to be replaced, replaced, resumed, etc.) together with transitions are to be added. If the QoS could be negotiated prior or during the composition, again states and transitions should be added modeling the negotiation status and the established QoS. As those control behaviors aim to solve general problems, the principles governing their design are similar to those used for design patterns in software engineering.

### 3.2. Operational behavior

The operational behavior of a composition of Web services illustrates the business logic implementing the functioning of a given (either orchestration or choreography) composition. The operational behavior is application dependent and is overseen by the control behavior. Formally, we define the operational behavior as follows.

**Definition 5 (Operational Behavior).** The operational behavior of a composition is a 7-tuple $OB = \langle M_{Ob}, S_{Ob}, S^1, S^2, s_0, F_{Ob}, \rightarrow \rangle$, where:

• $M_{Ob}$ is a finite set of messages including the empty message $\epsilon$.

• $S_{Ob}$ is a finite nonempty set of states.

• $S^1$ is a set of sequential states.

• $S^2$ is a set of and-states.

• $s_0 \in S_{Ob}$ is the initial state.

• $F_{Ob} \subseteq S_{Ob}$ is the set of final states.

• $\rightarrow \subseteq S_{Ob} \cup S^1 \cup S^2 \times M_{Ob} \times S_{Ob} \cup S^1 \cup S^2$ is a transition relation.

The operational behavior is described by three types of states: normal states (from $S_{Ob}$), sequential states (from $S^1$), and and-states (from $S^2$). Sequential and and-states are composite states that respectively, capture the sequence of functionalities and concurrency. Sequential and and-states are formally defined as follows.

**Definition 6 (Sequential state).** Each sequential state $s^1 \in S^1$ is a tuple $s^1 = \langle S^1, S^1(0), R^1 \rangle$, where:

• $S^1$ is a finite set of states, such that each state is an operational behavior.

• $S^1(0) \in S^1$ is the initial state.

• $R^1 \subseteq S^1 \times S^1$ is a strict order over $S^1$.

Intuitively, two states $s^1_1, s^1_2 \in S^1$ such that $(s^1_1, s^1_2) \in R^1$ means that the operational behavior of $s^1_1$ will be executed before the operational behavior of $s^1_2$.

**Definition 7 (And-state).** Each and-state $s^2 \in S^2$ is a tuple $s^2 = \langle S^2, S^2(0), R^2 \rangle$, where:
Definition 8. Let OB be an operational behavior over \( M_{ob} \). A conversation in OB over a sequence of messages \( m_0, m_1, \ldots, m_n \) from \( M_{ob} \) is a sequence \( s_0 = s_{i_0} \xrightarrow{m_0} s_{i_1} \cdots \xrightarrow{m_{n-1}} s_{i_n} \), such that \( \forall i \geq 0 \) one of the six following conditions holds:

1. \( s_{i_0}, s_{i+1} \in S_{ob} \) and \( (s_{i_0}, m_i, s_{i+1}) \in \rightarrow \).
2. \( s_i \in S_{ob} \) and \( \exists s' \in S : s' = (s_i, s_i', \delta_i', i') \wedge s_i' \in s_{i+1} \).
3. \( s_i \in S_{ob} \) and \( \exists s' \in S : s' = (s_i, s_i', \delta_i', i') \wedge s_i' \in s_{i+1} \).
4. \( \exists s'' \in S' \cup S'' \) and \( s_i \) is a state in the operational behavior associated with a state in \( s'' \) and \( (s_i, m_i, s_{i+1}) \) is a transition in this operational behavior.
5. \( \exists s' \in S' \) and \( s_{i+1} \) is an initial state of the operational behavior associated with \( s' \) and \( (s_i, m_i, s_{i+1}) \) is a transition relating the two operational behaviors; or
6. \( s_{i+1} \) is a final state in the operational behavior of a state in \( s'' \).

The first condition in this definition captures transitions between normal states. The second and third conditions represent transitions connecting normal states (as source states) to sequential and and-states (as target states) respectively. In these two conditions, the initial state of the operational behavior associated with the first state (in the case of sequential states) or any state (in the case of and-states) are used. The fourth and fifth conditions capture the internal transitions within the operational behaviors associated with sequential or and-states. The fourth condition focuses on inter-transitions within the same internal operational behavior, while the fifth condition stresses intra-transitions between sequential sub-states of the same sequential state. Finally, the sixth condition captures the last option, which is linking an internal state of an operational behavior associated with a sequential or and-state (as source state) to a normal state (as target state).

Because the control behavior specifies the properties of supposedly valid and correct executions, the conformance of the operational behavior of a composition of Web services to this control behavior is a proof that such a composition is well designed. We define such a conformance in terms of correspondence between the conversations of Web services in the operational behavior of a composition and the control behavior executions. We distinguish then two types of conformance: weak and strong. The former means each conversation \( c \) in the operational behavior (Definition 8) is simulated by an execution \( e \) in the control behavior (Definition 4), which we denote by \( c \sim e \). This implies that each conversation among Web Services according to the business logic is valid and correct since it is recognized and authorized by the control behavior that frames and monitors the executions. This simulation represents the soundness property. The latter means that each valid execution \( e \) in the control behavior is simulated and...
represented by a possible conversation $c$ in the operational behavior ($e \equiv c$), which guarantees the completeness property since all valid executions are implemented in the business logic. Intuitively, this means all the desired properties a composition should satisfy are implemented in the business logic captured by the operational behavior. We have then the following definition:

**Definition 9 (Soundness and Completeness).** Let $CW$ be a composition of Web services, $C$ the set of conversations in the operational behavior of $CW$, and $E$ the set of executions in the control behavior.

- $CW$ is sound if $\forall c \in C \exists e \in E: c \equiv e$.
- $CW$ is complete if $\forall e \in E \exists c \in C: e \equiv c$.

To illustrate the control and operational behaviors synchronization, we use a composition of Web services for ticket reservation whose operational behavior is shown in Fig. 2. The statechart used in the figure consists of:

- Filled circle, pointing to the initial state.
- Hollow circle containing a smaller filled circle, indicating the final state (if any).
- Rounded rectangle, denoting a state. This rectangle contains the name of the state.
- Arrow, denoting transition. The name of the event causing this transition labels the arrow.

Initially, the execution process is in the “Itinerary Received” state upon itinerary receipt. Then the process invokes the airline reservation Web service. If the airline reservation system is done without faults, vehicle and hotel reservation Web services will be invoked concurrently. If a time-out or fault occurs, the process will end with errors (cancellation). Otherwise, the invocation of these Web services is done correctly. The process moves then to the “Itinerary Modified” state. At the end, when the submission is done, the process moves to the “Itinerary Returned” state, which is a final state and so until the complete itinerary is returned to the client. Otherwise, the process moves to another final state representing a failure (“Itinerary Canceled”).

In this example, Conversation1 in the operational behavior is mapped onto Execution1 in the control behavior, where $*$ means repetition (or loop) and:

**Conversation1** = (Itinerary Received $\rightarrow$ = Airline Invoked Reservation Copied $\rightarrow$ = Hotel Invoked Reservation Copied $\rightarrow$ = Vehicle Invoked Reservation Copied $\rightarrow$ Submission $\rightarrow$ Itinerary Returned)

**Execution1** = (Not Activated $\rightarrow$ = Received Put on Hold $\rightarrow$ = Received $\rightarrow$ Received $\rightarrow$ Received $\rightarrow$ Done)

This mapping shows that Conversation1 is a correct composition. It corresponds to a valid execution of the composition. So, to check that the composition in Fig. 2 is sound, we need to check that the designed conversations can be mapped onto valid executions in Fig. 1. Similarly, to show that the composition is complete, we need to verify that all the valid executions in Fig. 1 could be mapped onto conversations in Fig. 2. In general, the mapping is not always one-to-one (state-to-state and transition-to-transition), but it could be more complicated. In fact, as the control behavior is supposed to be application-independent and more abstract, it should model many classes or scenarios of compositions of Web services. Thus, a state in this control behavior could be mapped onto a finite path (i.e., conversation) in the operational behavior. For instance, vehicle and hotel reservation Web services in the operational behavior are mapped onto received and put on hold or received and aborted states in the control behavior even if more states and transitions are added to the business logic of these two Web services. Verifying this conformance manually is then tedious even when the number of states in both behaviors is small. The number of possible conversations and executions grows exponentially with the number of total states (states of control and operational behaviors). We propose then, in the next section, an automatic technique to perform this verification based on model checking, and we prove that this technique can be used to check the soundness and completeness of composite Web services.

4. Model checking approach

4.1. Soundness and completeness

To verify the conformance of the operational behavior with the control behavior of a composition of Web services, we convert the former behavior into a system model $M$, represented as a Kripke structure (Definition 10), and extract from the latter behavior all
the properties $\varphi$ that this model has to satisfy, which are deadlock freedom, safety, and liveness (or reachability). In this section, we illustrate how these properties are defined in the control behavior using the control behavior of Fig. 1 as example. These properties are then represented in a temporal logic format. Checking conformance is now a model checking problem, which consists of verifying whether $M \models \varphi$ or not.

**Definition 10 (Kripke Structure).** A Kripke structure, a nondeterministic finite state machine, is a 4-tuple $K = \langle D, I, R, V \rangle$, where:

- $D$ is a finite set of states.
- $I$ is a set of initial states.
- $R \subseteq D \times D$ where $\forall d \in D \exists d' \in D$ such that $(d, d') \in R$ is a total transition relation.
- $V : D \times \Phi \rightarrow \{\text{True}, \text{False}\}$ is a valuation function associating true or false with atomic propositions from the set $\Phi$ in each state from $D$.

Beyond being formal and fully automatic, model checking permits to automatically guarantee the soundness and completeness as defined in Definition 9, thanks to the following theorem:

**Theorem 1.** Let CW be a composition of Web services and $M$ the Kripke model associated with its operational behavior. CW is sound and complete iff $M \models \varphi$ for all properties $\varphi$ satisfied in the control behavior.

**Proof.** Let us first prove the soundness. According to Definition 9, a composition of Web services is sound if for every conversation $c$ in the operational behavior, there exists an execution $e$ in the control behavior such that $c \models e$. According to model checking algorithms, $M \models \varphi$ implies that in all possible computations of $M$, the property $\varphi$ is satisfied. Because $\varphi$ is a property satisfied in the control behavior, there is an execution $e$ in this behavior through which $\varphi$ is satisfied. Because each computation in $M$ is a conversation in the operational behavior, it follows that each conversation can be mapped onto an execution, so we are done.

Let us now prove the completeness. According to Definition 9, a composite Web service is complete if for every execution $e$ in the control behavior, there exists a conversation $c$ in the operational behavior such that $c \models e$. According to model checking algorithms, $M \models \varphi$ implies that there is no possible computation in $M$ violating $\varphi$. Therefore, there is no conversation in the operational behavior violating properties satisfied in the control behavior. In other words, each property in the control behavior is satisfied by a given conversation in the operational behavior. Because every execution in the control behavior can be associated with a given property, it follows that each execution is mapped onto a given conversation, whence the result. \hfill \square

Having proved the importance of the proposed model checking for composite Web services, let us explain how it could be used by explaining how the properties to check are expressed and the model $M$ is obtained from the operational behavior of such Web services.

### 4.2. Properties to check

The properties to check in the Kripke structure $M$ associated with the operational behavior of composite Web services can be written in propositional LTL and CTL formats (Emerson, 1990). These two formal languages, widely used in model checking, are not equivalent since some properties can be expressed in LTL but not in CTL and vice versa. Here we introduce the syntax of these two languages. Let $\Phi$ be the set of atomic propositions and $p \in \Phi$. LTL syntax in BNF grammar is as follows:

$$
\Phi := \text{True} \mid p \mid \neg \Phi \mid \Phi \land \Phi \mid \Phi \lor \Phi \mid \Phi \rightarrow \Phi \mid \Phi \mid X \Phi \mid \Phi \mid \Phi \mid \Phi
$$

CTL syntax is as follows:

$$
\Phi := \text{True} \mid \neg \Phi \mid R \Phi \mid \Phi \land \Phi \mid \Phi \lor \Phi \mid \Phi \rightarrow \Phi \mid A \Phi \mid \Phi
$$

$X \Phi$ means in the next state $\Phi$ is true. $\Phi U \Psi$ means $\Phi$ is true until $\Psi$ becomes true. $E$ and $A$ are the existential and universal quantifiers over paths. $F$ (in a state in the future) and $G$ (globally, i.e. in all states) are abbreviations, i.e., $F \Phi \equiv \text{True} \land \Phi$ and $G \Phi \equiv \neg F \neg \Phi$.

A path is an infinite ordered sequence of states such that each state is followed by another state via a transition. Given a path $x$, $x(i)$ refers to the state $i$ in $x$ and $x_i$ refers to the suffix of $x$ starting from $x(i)$. LTL semantics is given as usual using a Kripke structure $M$ (Definition 10). In LTL, formulas are evaluated over paths, so $M, x \models \Phi$ means $\Phi$ is satisfied in the model $M$ along the path $x$ (as abbreviation, we will use $x \models \Phi$). LTL semantics is as follows (we also give the semantics of some abbreviations for convenience purposes):

- $x \models p$ iff $V(x(0), p) = \text{True}$, where $p \in \Phi$.
- $x \models \neg \Phi$ iff $x \not\models \Phi$.
- $x \models \Phi \land \Psi$ iff $x \models \Phi$ and $x \models \Psi$.
- $x \models \Phi \lor \Psi$ iff $x \models \Phi$ or $x \models \Psi$.
- $x \models X \Phi$ iff $x_{i+1} \models \Phi$.
- $x \models G \Phi$ iff for all $i \geq 0, x_i \models \Phi$.
- $x \models F \Phi$ iff there exists an $i \geq 0$ such that $x_i \models \Phi$.
- $x \models \Phi U \Psi$ iff there exists an $i \geq 0$ such that $x_i \models \Psi$ and for all $0 \leq j < i$, $x(j) \not\models \Phi$.

In LTL, formulas are evaluated in states, so $M, s \models \Phi$ means $\Phi$ is satisfied in the model $M$ in the state $s$ (as abbreviation we use $s \models \Phi$). A path starting at $s$ (also denoted by $s(0)$) is an infinite sequence of states $s(0), s(1), \ldots$ linked by transitions. Given a state $s$ in the Kripke structure, CTL semantics is as follows:

- $s \models p$ iff $V(s, p) = \text{True}$, where $p \in \Phi$.
- $s \models \neg \Phi$ iff $s \not\models \Phi$.
- $s \models \Phi \land \Psi$ iff $s \models \Phi$ and $s \models \Psi$.
- $s \models \Phi \lor \Psi$ iff $s \models \Phi$ or $s \models \Psi$.
- $s \models AX \Phi$ iff there exists a path $s(0), s(1), \ldots$ such that $s(1) \models \Phi$.
- $s \models EX \Phi$ iff there exists a path $s(0), s(1), \ldots$ such that for all $i \geq 0$, $s(i) \models \Phi$.
- $s \models EG \Phi$ iff there exists a path $s(0), s(1), \ldots$ such that for all $i \geq 0$, $s(i) \models \Phi$.
- $s \models AF \Phi$ iff for all paths $s(0), s(1), \ldots$, all $i \geq 0$, $s(i) \not\models \Phi$.
- $s \models EF \Phi$ iff there exists a path $s(0), s(1), \ldots$ such that there exists an $i \geq 0$ and $s(i) \models \Phi$.
- $s \models A F \Phi$ iff for all paths $s(0), s(1), \ldots$, there exists an $i \geq 0$, such that, $s(i) \models \Phi$.
- $s \models E F \Phi$ iff there exists a path $s(0), s(1), \ldots$ such that, there exists an $i \geq 0$ such that $s(i) \models \Psi$ and for all $0 \leq j < i$, $s(j) \models \Phi$.
- $s \models A (F U \Psi)$ iff for all paths $s(0), s(1), \ldots$, there exists an $i \geq 0$ such that $s(i) \models \Psi$ and for all $0 \leq j < i$, $s(j) \models \Phi$.

In LTL, the default path quantifier is $A$, so a state satisfies a formula if it is satisfied in all paths starting by this state. As already mentioned, the reason behind using two different languages LTL and CTL to specify the properties is because they are not equivalent. There are properties that can be expressed in LTL but cannot be expressed in CTL (e.g., $AF(p \land Xp)$) and vice versa (e.g., $AGE(p)$). We also notice that we consider fair LTL and CTL (Emerson, 1990), which means in any computation, some states called fair states, should be reached. In the control behavior of Fig. 1, done and aborted are two fair states. Thus, in any execution, done or aborted should be reached.

To specify the properties $\Phi$ we aim to check in the Kripke model $M$ associated with the operational behavior using the control
behavior in which those properties are satisfied, we will consider
the following initials (Fig. 1): Not activated: Na; Received: Re; Put
on hold: Ph; Suspended: Su; Aborted: Ab; Processed: Pr; Compensa-
ted: Co; Done: Do; and End: En. Let $\rightarrow$ be the logical implication.
Examples of these properties expressed in fair LTL include:

- $\varphi_1 = G(\neg Su \land XDo))$.
- $\varphi_2 = G((\neg Ph U Co))$.
- $\varphi_3 = G(Na \rightarrow XRe)$.
- $\varphi_4 = G(Re \rightarrow XF(Ph \lor \neg Ab \lor Su \lor Do))$.
- $\varphi_5 = G(Do \rightarrow XFEn)$.
- $\varphi_6 = G((Do \lor Ab) \rightarrow XFEn)$.
- $\varphi_7 = G((Ph \lor \neg Pr \lor Re \lor Su))$.
- $\varphi_8 = G(Do \lor (Ab \lor Pr \lor Re \lor Su))$.

The two first properties having the form $\varphi = G(\neg \varphi)$ are safety
properties. Property L1 states that in all possible computations, it is
not the case that the composition process achieves state done
right away after being suspended. Property L2 states that in all possible
combinations, it is not the case that the composition process is
conturally put on hold until its compensation. The properties L3
to L8 having the form $\varphi = G(\varphi \lor XW)$ or $\varphi = G(\varphi \lor FP)$ are live-
ness properties. For illustration purposes, let us explain some of
them. Property L3 states that always the composition process will
be in received state if the previous state is not-activated state. Prop-
ter L4 states that there is always put on hold, aborted, suspended, or
done state in the future after received state. Finally, property L7
states that there is always end state after aborted or done states.
Property L9 is the deadlock freedom property stating that from
each state computation is not stopped.

Examples of CTL properties from the control behavior include:

- $\varphi_1 = AG(\neg Su \land EXDo))$.
- $\varphi_2 = AG(\neg Su \land AXDo))$.
- $\varphi_3 = AG(\neg E(Ph U Co))$.
- $\varphi_4 = AG(Na \rightarrow AXRe)$.
- $\varphi_5 = AG(Re \rightarrow AXAF(Ph \lor Ab \lor Su \lor Do))$.
- $\varphi_6 = AG(Do \rightarrow AXAFN)$.
- $\varphi_7 = AG((Do \lor Ab) \rightarrow AXAFEn)$.
- $\varphi_8 = AG((Do \lor Ab) \rightarrow AXAFEn)$.
- $\varphi_9 = AG((Ph \lor AXAF(Ph \lor Ab \lor Su \lor Do)))$.
- $\varphi_{10} = AG(Do \lor (Ab \lor Pr \lor Re \lor Su))$.

Notice that the first two properties C1 and C2 cannot be
expressed in LTL. The first one (with existential quantifier E) is
weaker, while the second one (with universal quantifier A) is stron-
ger than L1 (see (Baier & Katoen, 2008), Ch. 6 for a proof on similar
formulas). Consequently, L1 cannot be expressed in CTL as the two
candidates (universal (C2) and existential (C1)) are not equivalent
to it. Property C3 cannot be expressed in LTL as it uses the existen-
tial quantifier. Other examples of properties in CTL that cannot be
expressed in LTL are:

- $\varphi_{11} = AGER(En)$.
- $\varphi_{12} = AGER(Re \rightarrow EXPH)$.
- $\varphi_{13} = EF(Re \lor Ph) U Do)$.
- $\varphi_{14} = AGER(Ph \rightarrow EX(Pr \lor Re))$.
- $\varphi_{15} = AGER(Pr \rightarrow EX(Re \lor Ab))$.

Capturing reachability, property C11 is stating that in all paths
there always exists a path that will reach end state. Property C12 is
liveness stating that globally in all paths there always exists a path
where achieving received state implies the existence of a path so that
the next state on that path is put on hold state. Property C13 is an-
other reachability indicating the existence of a computation so that
in its future done state will be reached after a sequence of received
and put on hold states. Property C14 indicates that globally in all
computations there exists a path through which if put on hold
state is reached then there is a path so that the next state will be
processed or received. Finally property C15 can be explained in a
similar way.

4.3. System models

After extracting the properties $\varphi$ to be checked in the opera-
tional behavior from the alternating Büchi automaton of the con-
trol behavior, the second step in the verification is to build the
Kripke model $M$ from this operational behavior. The resulting mod-
el is the one we use to automatically generate the SMV code used
by the NuSMV model checker (Cimatti et al., 2002). This translation
is automatic as per the following algorithm. Each state $s_{op}$ in
the operational behavior is translated into a set of states and transi-
tions in the Kripke structure $M$ and each transition is translated
into one or many transitions. If $s_{op}$ is a simple state, it is translated
into one state in $M$ with the same content. If $s_{op}$ is a statechart, then
two cases are possible: (1) the state is a sequential state; (2) the
state is an and-state. In both cases, each simple state is translated
into one state with the same content and all the end states are
translated into one end-state. In the first case, the connector is re-
placed by the next state if this state is simple, or by the first state of
the next sequential state or and-state. In the second case, the and-
states are simply considered as sequential and the sequence order is
selected randomly. The reason is that in an and-state, all states
should be considered but the order of this consideration is not
important. Only the last state in the selected order is related to
the next state by a transition. The number of possible Kripke struc-
tures generated from a given operational behavior depends then on
the number of sub-states in and-states in this behavior as each or-
der of these sub-states bears a possible Kripke structure. For exam-
ple, if the operational behavior includes 2 and-states, each of which
includes 2 sub-states, then 4 order choices are possible, making the
number of Kripke structures that can be generated equal to 4.
However, all the executions of these possible structures are equiva-
elent as the order between the sub-states of an and-state is not
important because these states are independent. Consequently,
only one structure among all the possible ones should be consid-
ered. The conditional selections are simply ignored as they are cap-
tured by deterministic transitions because transitions are labeled.
Transitions between simple states are translated into transitions
between the corresponding states in the Kripke structure. Transi-
tions between simple and sequential states or and-states are trans-
lated into transitions between the corresponding states of the
simple state and the corresponding state of the first state of the
sequential state or and-state according to the selected order. We
have then the following proposition and the proof is straightfor-
ward from the explanation above:

**Proposition 1.** The translation procedure from the operational
behavior into Kripke models explained above terminates and is sound
and complete.

Termination means that the translation procedure always ends
after a finite number of iterations. Soundness and completeness
mean all the executions (i.e., sequences of states and transitions)
in the operational behavior are captured in the associated Kripke
model and there are no additional executions in the Kripke model
other than those of the operational behavior.

Fig. 3 shows the Kripke model obtained after translating the
operational behavior in Fig. 2. In Fig. 3, after an airline Web service
is invoked, the action could be committed directly, or vehicle and
hotel Web services could also be invoked depending on the initial
client request. At any time the reservation could be canceled and
the process is \textit{aborted} in that case. The atomic propositions that are true in the obtained states using the evaluation function $V$ are those used in the control behavior. Fig. 4 shows the same Kripke model where states are simply renamed using the state names of the control behavior, which makes the verification clearer as the properties to be checked are extracted from the control behavior. For instance, \textit{idle} state corresponds to non-activated state and \textit{airline reservation canceled} state corresponds to aborted state.

### 4.4. Model checking technique

The model checking technique $M \models \varphi$ consists of computing whether or not a formal model $M$ representing the system satisfies a logical formula $\varphi$ describing a certain property. The computation is usually automatic for finite models. The approach used in this paper is called symbolic model checking. This approach avoids building or exploring the state space corresponding to the models explicitly. Instead, a symbolic representation is used based on ordered binary decision diagrams (OBDDs) or propositional satisfiability (SAT) solvers (Clarke et al., 1999).

The model checker we use is NuSMV (Cimatti et al., 2002). It is a software tool for the formal verification of finite state systems based on symbolic model checking. NuSMV allows checking finite state systems against specifications in the temporal logics LTL and CTL. The input language of NuSMV allows the description of finite state systems that range from completely synchronous to completely asynchronous. The basic purpose of the NuSMV language is to describe the transition relations of a finite Kripke structure. It supports modular hierarchical descriptions and definition of reusable components. This tool has been designed as an open architecture for model checking. It is aimed at reliable verification of industrially sized designs, for use as a backend for other verification tools and as a research tool for formal verification techniques.

The advantage of NuSMV is the flexibility in the use. However when used by non experts, there is still a risk of providing inconsistent code of the system behavior. To manage this inconsistency issue, we provide an automatic translation from the Kripke structure obtained by our translation procedure from the operational behavior into the SMV code. The deadlock freedom, safety, and reachability properties to check are, also, extracted from the control behavior and translated into LTL and/or CTL, which are supported by alternating Büchi automata used to model the control behaviors (we call this extraction and translation: \textit{process instantiation}). The obtained SMV program and LTL and CTL formulas are then submitted to the NuSMV model checker for verification. The approach of our model checking-based technique is described in Fig. 5.

### 4.5. Reduction algorithm

Before translating the Kripke model of the operational behavior into an SMV program, we optimize this model by reducing its size. This permits to reduce the verification time and complexity. The intuition behind our reduction algorithm is similar to the one behind the reduction of OBDDs to obtain Reduced OBDDs (ROBDDs) (Clarke et al., 1999), which is based on removing or merging redundant and irrelevant states. The reduction algorithm reduces the number of states and transitions based on the fact that two states labeled with the same atomic propositions using the valuation function $V$ are equivalent, so they can be combined into one state. The transitions are then reduced as follows:

For all $s_1$ and $s_2$, if $s_2$ is reduced to $s_1$, which means $s_1$ and $s_2$ are equivalent, then:
can be generated. In this case, the new
transition in TK
(1) adding one state; and (2) adding one transition. By the
completeness still holds if

Theorem 2. Let K be a Kripke model and K
be the reduced model obtained using the reduction algorithm. K and K' are semantically equivalent.

Proof. Let T_K be the set of transitions in K and T_{K'} be the set of transitions in K'. To prove the result, we should prove that for each transition in T_{K'}, there is a semantically corresponding transition in T_K (soundness) and vice versa (completeness). By doing this, all the runs will be captured.

The soundness is simply proved by construction as all the transitions in K' are constructed from the transitions in K, so we cannot obtain additional transitions in K'.

We prove completeness by induction on the number of states and transitions of the Kripke model K (i.e. |K| the size of K).

Base case: if |K| = 1 (only one state), then K' is exactly K, so the completeness holds. The same result holds if |K| = 2 (one state and one loop transition).

Induction case: Suppose that the completeness holds for |K| < n (n 2), and consider |K| = n. When |K| = n – 1, two cases are possible: (1) adding one state; and (2) adding one transition. By the induction hypothesis, the completeness holds for |K| = n – 1, and because adding one state will not generate additional runs in K, the completeness still holds if |K|n. By adding one transition, new runs can be generated. In this case, the new K' will be obtained from the previous one by executing one of the fourth rules (a), (b), (c), or (d).

The proof is then by deduction on the reduction rules. For the first rule, the removed transitions from T_{K'} are semantically captured by the loop transition in T_K as the two states s_1 and s_2 are equivalent. For the second rule, the removed transition is captured by staying in the state. In fact, here we have (s_1, s_2) ∈ T_{K'} and s_1 and s_2 are equivalent, so one state and the transition are redundant. For the third and fourth rules, the removed transitions are captured by the replaced transitions because (s_0, s_1) and (s_0, s_2) are equivalent and (s_1, s_2) and (s_2, s_0) are equivalent since s_1 and s_2 are equivalent.

The reduction algorithm preserves then the semantics of the original Kripke model and is automatically performed. Fig. 6 depicts the result of reducing the Kripke model presented in Fig. 5.

(a) If (s_1, s_2) and (s_2, s_1) are two transitions, then they are replaced by one loop transition (s_1, s_1).
(b) If only one of the two transitions does exist, then it is removed as it cannot be replaced by a loop and staying in the same state simulates the removed transition.
(c) For all x, if (s_0, s_2) is a transition, then it is removed and replaced by the transition (s_0, s_1) if such a transition does not exist.
(d) For all y, if (s_2, s_0) is a transition, then it is removed and replaced by the transition (s_1, s_0) if such a transition does not exist.

Definition 11 (Equivalence). Two Kripke models K and K' are semantically equivalent if and only if each run in K (sequence of states and transitions linking the initial state to a final state) can be simulated by a run in K' and visa versa.

Then, the reduced model is automatically translated into the SMV code by the NuSMV model checker. SMV code mainly describes the transition relation of the Kripke model (Appendix A).

To check the properties described in Section 4.2, the following commands are used:

NuSMV > read_model –i TRS.smv (TRS.smv is the name of the smv file we created)
NuSMV > flatten_hierarchy
NuSMV > encode_variables
NuSMV > build_model
NuSMV > check_ltlspec (to check ltl specifications)
NuSMV > check_ctlspec (to check ctl specifications)

First, we have to read the SMV program then flatten the hierarchy, encode the variables, and build the model. The specifications are checked afterwards.

5. Implementation

In the previous section, we explained our model checking-based approach to verify composite Web services through the following steps:

(a) Translate the statechart of the operational behavior into a Kripke model.
(b) Reduce the obtained Kripke model.
(c) Translate the reduced model into SMV code; and
(d) Verify the properties extracted from the alternating Büchi automaton of the control behavior.

In order to make the translation easy and automatic for the user, we implemented a translation procedure in Java and a tool (SMV converter) equipped with an interface where the user can enter the different states and transitions as indicated in the original statechart. The user can, also, add LTL and CTL properties and then the SMV file is automatically created.

Our SMV converter is responsible for converting statecharts into SMV code to automatically help the user in the verification process.

Fig. 5. Model checking-based technique of composite Web services.

Fig. 6. Reduced Kripke model of ticket reservation system.
This converter consists of four areas. The first one (1) is for state-charts. The second and third areas (2 & 3) are for LTL and CTL specifications, respectively. Finally, the fourth area (4) displays the SMV code that will be put in the “.smv” file (Fig. 7).

The most important phase is the translation from the statechart of the operational behavior to the SMV syntax. From the state-charts, we extract the different transitions and each transition is identified by indicating the state from which the transition starts (called from state) and the state to which the transition ends (called to state). We assign types to these states from the states used in the control behavior to help in the SMV translation procedure. The types are: Not Activated, Received, Put on Hold, Processed, Aborted, Compensated, Suspended, Done, and End. Therefore, in the first area, we have 4 fields:

1. FROM STATE field: the user enters in this field the state from which the transition begins.
2. FROM TYPE field: the user chooses, from the drop-down list of types, the type of the FROM STATE.
3. TO STATE field: the user enters in this field the state where the transition ends; and
4. TO TYPE field: the user chooses, from the drop-down list of types, the type of the TO STATE.

To facilitate the comprehension of this procedure, we list in Table 1 the different FROM STATE, FROM TYPE, TO STATE, and TO TYPE of the ticket reservation system statechart (Fig. 2). This table corresponds to the different transitions of this system. The records in Table 1 are entered in the appropriate fields of the first area of our SMV converter. At the end, when all the information is entered.

<table>
<thead>
<tr>
<th>FROM STATE (FS)</th>
<th>TYPE OF FS</th>
<th>TO STATE (TS)</th>
<th>TYPE OF TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIAL</td>
<td>NOT ACTIVATED</td>
<td>ITINERARY RECEIVED</td>
<td>RECEIVE</td>
</tr>
<tr>
<td>ITINERY RECEIVED</td>
<td>RECEIVE</td>
<td>AIRLINE INVOKED</td>
<td>PUT ON HOLD</td>
</tr>
<tr>
<td>AIRLINE INVOKED</td>
<td>PUT ON HOLD</td>
<td>AIRLINE RESERVATION COPIED</td>
<td>ABORTED</td>
</tr>
<tr>
<td>AIRLINE RESERVATION COPIED</td>
<td>PROCESSING</td>
<td>HOTEL INVOKED</td>
<td>PUT ON HOLD</td>
</tr>
<tr>
<td>HOTEL INVOKED</td>
<td>PUT ON HOLD</td>
<td>HOTEL RESERVATION COPIED</td>
<td>ABORTED</td>
</tr>
<tr>
<td>HOTEL RESERVATION COPIED</td>
<td>PROCESSING</td>
<td>VEHICLE INVOKED</td>
<td>PUT ON HOLD</td>
</tr>
<tr>
<td>VEHICLE INVOKED</td>
<td>PUT ON HOLD</td>
<td>VEHICLE RESERVATION COPIED</td>
<td>ABORTED</td>
</tr>
<tr>
<td>VEHICLE RESERVATION COPIED</td>
<td>PROCESSING</td>
<td>ITINERARY MODIFIED</td>
<td>PROCESSING</td>
</tr>
<tr>
<td>ITINERY MODIFIED</td>
<td>PROCESSING</td>
<td>ITINERARY RETURNED</td>
<td>RECEIVE</td>
</tr>
<tr>
<td>ITINERY RETURNED</td>
<td>RECEIVE</td>
<td>DONE</td>
<td>DONE</td>
</tr>
<tr>
<td>CANCELED</td>
<td>ABORTED</td>
<td>END</td>
<td>END</td>
</tr>
</tbody>
</table>

Fig. 7. SMV converter tool.
the user can require the automatic conversion, which is responsible for converting all these transitions into the first part of the SMV code (Appendix A). These transitions will appear in area 4, namely SMV PROGRAM. LTL and CTL specifications are extracted from the control behavior as stated before. To help the user in writing LTL properties in SMV syntax, the second area of the implemented tool includes buttons such as LTLSPEC, G, F, etc. After the user enters one or several properties, he can click on the ADD LTL SPEC button available in this area. These specifications are then automatically added to SMV program we construct in area 4 (see Fig. 7).

Moreover, the third area concerns the CTL properties. In this area, there are buttons that are specific to the CTL syntax such as SPEC, A, E, etc. that could be used by the user to write a CTL property in an SMV syntax. Like in the LTL area, after the user inputs one or many properties, he can click on the ADD CTL SPEC button available in this area. These specifications are then added to the SMV program of area 4 (see Fig. 7).

After adding as many properties as needed, the user can finalize the SMV code by clicking on the button CREATE SMV FILE, so the syntax of the program is shown in area 4 (Fig. 7) and the SMV file is getting created. This file is then automatically given to the NuSMV model checker to verify the composite Web services. Table 2 shows relevant statistics of this verification for the example of ticket reservation system.

### Table 2

<table>
<thead>
<tr>
<th>Memory in use (byte)</th>
<th>4,764,980</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of BDD variables</td>
<td>13</td>
</tr>
<tr>
<td>Number of variables sifted</td>
<td>1000</td>
</tr>
<tr>
<td>Number of variable swaps</td>
<td>2,000,000</td>
</tr>
<tr>
<td>Total number of nodes</td>
<td>4339</td>
</tr>
<tr>
<td>Reordering time (s)</td>
<td>0.20</td>
</tr>
<tr>
<td>Execution time (s)</td>
<td>1.20</td>
</tr>
</tbody>
</table>

6. Relevant related work

The concept of control and operational behaviors was previously studied in Maamar, Sheng, Yahyaoui, Bentahar, and Boukadi (2009), Yahyaoui, Maamar, and Boukadi (2009) and Sheng et al. (2010) where the control behavior illustrates the business logic that underpins the functioning of an isolated Web service, and the operational behavior regulates the execution progress of this control behavior by stating the actions to carry out and the constraints to put on this progress. However, the verification aspects were not investigated. The composition issue from a formal perspective and the tools used were, also, stated in some papers. In Hull and Su (2005), the authors describe concepts and assumptions on current proposals on composition. They present several composition models including semantic Web services, the Roman model, and the Meaty conversation model. They also give techniques for analyzing Web services such as translating them into formalisms that are suitable for analysis, for example state machines, extended mealy machines, and process algebra. However, synchronization between behaviors and verification of composition design were not analyzed.

Other projects that use model checking techniques for BPEL composite Web services verification were done. In Foster, Uchitel, Magee, and Kramer (2003), the authors verify mediated composite services specified in BPEL against the design specified using Message Sequence Chart and Finite State Process notations. Unlike our proposal, the focus is on the control flow logic and not on the conversations between the composite services. Also, the proposed verification method has not been implemented. In Fu, Bultan, and Su (2004a), the tool presented can be used to check that composite Web services satisfy LTL properties. The input of the tool is BPEL specifications and XML messages that are translated into guarded automata. These automata are then translated into the Promela language to check them in the SPIN model checker. This allows verifying designs at a more detailed level and to check properties about message content. Although the verification approach is similar to ours, there are many fundamental differences between the two proposals. In our work, the verification is based on separating and synchronizing behaviors reflecting two different abstractions and not only on BPEL composition flow, which only focuses on the operational aspect. Another important difference is the way the composition behavior is modeled. In Fu et al. (2004a), only sequential conversations i.e., the global sequence of messages recorded in the order in which they are sent are considered, while in the present paper alternations are considered as well. This difference has significant consequences on the technical level because capturing sequential conversations means only universal choices through all possible executions, captured by guarded automata are considered, which means only LTL is supported. However, in this paper, both universal and existential choices, captured by the control and operational behaviors are included. This implies restrictions on the properties that can be checked in Fu et al. (2004a) as a property has to be verified in all possible computations, otherwise it is violated. In this paper, in addition to universal properties, a property can still be satisfied if it holds in some specific computations. The second fundamental difference is related to specification of the properties to check. In Fu et al. (2004a), properties are supposed to be given. In this paper, properties are formalized through the control behavior. This difference has also a significant consequence as in our work soundness and completeness are guaranteed with respect to the operational and control behaviors, which has not been considered by Fu and his colleagues.

In Yeung (2011), a formal approach for web services composition for B2B applications and conformance verification based on WS-CDL and WS-BPEL is introduced. This approach makes use of model checking as an automated means of verifying choreography conformance. Like our approach, the paper focuses on the choreography-based composition of Web services. However, unlike our proposal that investigates the soundness and completeness of the design model with regard to the operational and control behaviors, Yeung’s work addresses the conformance problem considering only one type of behavior, namely the contract negotiation process.

In Su, Bultan, and Fu (2005), the authors show the importance of asynchronous messaging in sharing information and resources in the form of Web processes. Web service interaction models are formalized into a conversation concept with ordering constraints on messages. FIFO queues are considered in the design of message passing between services. In terms of verification, only some abstract strategies of model checking service composition for both bottom-up and top-down design approaches are outlined. However, neither analysis nor implementation of these strategies was provided.

Model checking of composite Web services has been studied also in Ding and Zhang (2007), Fu, Bultan, and Su (2004b), Hong, Lee, and Sokolsky (2001) and Lomuscio et al. (2008) using different model checkers. The main difference with this work concerns the properties to verify and the underlying technique. To the best of our knowledge, this work is the first investigation on separating concerns in composite scenarios and automatically verifying the operational behavior against the control specification using both LTL and CTL languages thanks to the way the two behaviors are modeled using the formalism of alternating Büchi automata. The technique is based on analyzing the two behaviors and extracting
properties from the general control behavior to be verified in the model represented by the operational behavior of the system. This enables us to control the composition process of Web services and guarantee its soundness and completeness.

In terms of Web service interactions, some researchers studied feature interactions in order to model and monitor undesirable interactions (Su et al., 2005; Weiss & Esfandiari, 2005; Weiss, Esfandiari, & Luo, 2007). Feature interactions for Web services are described as the situations where the requirements of services are inconsistent (Amyot, Gray, Liscano, Logrippo, & Sincennes, 2005). Feature interactions are often seen as the result of complex behavior interleaving for the state machines that represent the features. In Boutrous Saab et al. (2009), the authors studied the issue of generating an appropriate client given a BPEL-based description of the interaction protocol of a Web service. They achieved this using a formal semantics for BPEL via process algebra, which yields an algorithm which decides whether such a client exists and synthesizes the description of this client as a timed automaton. This work does not separate between the different behaviors of a composite Web service. In Layouni, Logrippo, and Turner (2008), a first-order logic model checking tool called Alloy is used for automated detection of feature interactions. Our work is different since we consider not only undesirable interactions, but all possible interactions that can be extracted from the control behavior (completeness property). The model checking technique we use is also different from the first order model checking. In Rouached et al. (2010), the authors mapped BPEL constructs onto the EC algebra. The authors further the semantic mapping to include Web service composition interactions through modeling Web service conversations and their choreography. The verification and validation techniques are also exposed while automated induction-based theorem provers are used as verification back-end. Unlike our proposal, this work does not discuss the soundness and completeness issues of composite Web services. Furthermore, our approach is based on model checking rather than theorem proving, and it has been proved that theorem proving is computationally more complicated than model checking, particularly for expressive logics, because instead of proving a formula given a knowledge base, we only check that the formula holds in a Kripke model (Halpern & Vardi, 1991). Consequently, model checking is more suitable for Web service settings as the verification time is usually very short (see Table 2).

Model checking Web services has benefited from model checking business processes, which has been largely investigated in the past years. Janssen, Mateescu, Mawu, Fennema, and van der Stappen (1999) demonstrated the benefits of using model checking for business modeling and analysis by people having limited expertise and knowledge in formal techniques. A business model of an insurance company has been used as a case study and Spin has been exploited as the model checker underlying a graphical modeling language. The business requirements to be verified are specified using business requirements patterns, which are translated to LTL. Model checking techniques are also used to automatically verify elementary CTL requirements of a business process such as the termination and reachability of states where the business model functionalities are specified using automata (Koehler, Tirenni, & Kumaran, 2002). Fisteus, Fernández, and Kloos (2005) applied model checking to distributed business processes among several organizations using the Spin and SMV model checkers. A modular and extensible framework for the verification of these processes, called VERBUS, has been developed aiming to assist the designer localize specification errors at design time. In Anderson, Hansen, Lowry, and Summers (2005), the authors used model checking to help design and evaluate business processes that satisfy some business logic properties such as the money atomically in distributed, concurrent and parallel settings. The authors extended existing frameworks on model checking electronic commerce protocols such as NetBill protocol (Heintze, Tygar, & Wong, 1996) and e-processes (Wang, Hidvegi, Bailey, & Whinston, 2000). In Anderson, Hansen, Lowry, and Summers (2006), the authors addressed the problem of the integrity and reliability of the transaction process focusing on fair-exchange and atomicity assurance. They showed how atomicity assurance in these transactions can be dealt with using model checking as a technique to verify the correctness of the implementation of e-commerce protocols. Rossetti (2009) presented a semantic timed model checking algorithm for business processes. The key idea of this approach is to separate the procedural description of business processes from their ontological descriptions, and to link the two through semantic annotations. The approach includes three steps: (1) a representation of business processes based on semantically annotated timed transition systems; (2) a representation of specifications based on a semantically annotated version of timed computation tree logic (TCTL); and (3) a model checking algorithm integrating the traditional timed model checking techniques with semantic reasoning. This algorithm has been proved to be sound and complete and PSPACE-Complete.

7. Conclusion and future work

In this paper, we discussed how composite Web services could be designed, modeled, and verified based on their control and operational behaviors. The operational behavior illustrates the business logic a composite Web service implements. The control behavior illustrates and states the constraints the operational behavior should satisfy and be in. Synchronizing both behaviors is a key issue in designing sound and complete compositions. The verification approach used in this paper suggests translating the operational behavior, formally modeled using an extended version of finite state machine, into a Kripke model, which is afterwards translated into the SMV code, so that symbolic model checking can be utilized. The properties to check are taken from the control behavior formally modeled using alternating Büchi automata, and are automatically verified in the different operational scenarios. Using these two formalisms to model the operational and control behaviors allows specifying and checking a wide range of properties expressed in both LTL and CTL computational logics.

The verification technique proposed is mainly based on the reduction of model checking compositions to model checking TLT and CTL by verifying the operational behavior against the properties specified by the control behavior, which is assumed to be correctly designed (as a design pattern). Two extensions of this approach are to be explored: (1) defining a specific model checking algorithm for the control behavior itself by considering different classes of this behavior, which would lead into some optimized results in terms of verifying the conformance of compositions; and (2) developing dedicated model checking algorithms for the whole approach and analyzing them from theoretical and computational perspectives, which would lead into a gain over the reduction overload. Moreover, in this paper, we only considered static composition processes. As future work, we plan to extend this approach for dynamic compositions. Considering dynamic compositions where new Web services can be involved is a particularly challenging issue. Consequently, we need the control and operational behaviors that correspond to dynamic processes. For instance, fault handling is easier in static compositions as the execution is controlled, which is not the case with dynamic compositions. Web services can be easily and transparently replaced in case of static compositions, whereas it will be difficult in the dynamic case.
Appendix A. SMV code for the NuSMV model checker

MODULE main

VAR

state : (Na,Re,Ph,Ab,Pr,Po,Do,En,Co,Su);

ASSIGN

init (state) = Na;

next (state) = case

(state = Na) : (Re); 
(state = Re) : (Ph,Do); 
(state = Ph) : (Pr,Ab,Re); 
(state = Ab) : (En); 
(state = Pr) : (Re); 
(state = Do) : (En); 
(state = En) : (Na);

1: state;

esac;

– LTL Specifications

LTLSPEC G ((state = Su & X state = Do))
LTLSPEC G ((state = Ph U state = Co))
LTLSPEC G ((state = Na & X state = Re))
LTLSPEC G ((state = Re & X F state = Ph))
LTLSPEC G ((state = Do & X F state = En))
LTLSPEC G ((state = Do & X F state = En))
LTLSPEC G ((state = Do & X F state = En))

– CTL Specifications

SPEC AG (((state = Su & EX state = Do))
SPEC AG (((state = Ph & EX state = Co)))
SPEC AG (((state = Na & AX state = Re)))
SPEC AG (((state = Re & AX state = Ph))
SPEC AG (((state = Co & AX state = Na)))
SPEC AG (((state = Do & AX state = En)))
SPEC AG (((state = Do & AX state = En)))
SPEC AG (((state = Re & AX AF state = Ph)))
SPEC AG (((state = Do & AX AF state = Na)))
SPEC AG (((state = Do & AX AF state = Na)))
SPEC AG (((state = Re & AX AF state = En)))
SPEC AG (((state = Do & AX AF state = En)))

References


