Annotated FSM for Modeling Verification and Composition of Web services

Deepak Chenthati1, Avula Damodaram2

1 Teradata(R&D) India Pvt Ltd, Hyderabad, India.
2 School of Information Technology, JNTUH, Hyderabad, India.
1 chvcdeepak@gmail.com

Keywords

web service, modeling, verification, structural, behavioural

Abstract

Web services are now an integral part of many of the applications we use daily. Model-based service generation would be required to make these services immediately accessible. Nevertheless, precise model specification is required for automated code generation, as inaccurate model specification results in errors and bugs in the generated code. Consequently, model verification is essential for eliminating issues even during the design phase. In this paper, we present a method for integrated model verification that can detect structural flaws during the design phase and behavioral flaws during the execution phase. A finite state machine with annotations was employed to identify structural flaws such as unreachability, deadlock, and temporal inconsistencies. In addition, context data is incorporated to assist in the detection of behavioral flaws by validating context sequence, context co-occurrence, and context timeliness. Web Services have the potential to improve Business-to-Business collaboration by combining compatible services into a service with a higher level of functionality. Two services are only composable if their conversation protocols are compatible. For modeling and matching conversations, the applicability of formal structures such as Finite State Machine (FSM) has been extensively researched. However, storing, matching, and retrieving FSMs using graph structure is not scalable for representing conversation protocol for the tens of thousands of web services that will soon be deployed. We also propose a relational model to store FSMs and a query system to retrieve services that match a given service in order to meet the challenge as an extension to juddi. The proposed integrated framework serves composition and verification of services.

This work is licensed under a Creative Commons Attribution Non-Commercial 4.0 International License.
Deepak Chenthati¹, Avula Damodaram²

1. Introduction

Model-driven software development methodology for systems is commonly accepted because it promotes modelling, transformation, scalability, portability, and improvement. Web-based services are easily accessible, and modelling of web services is a popular study topic. We’ve selected Finite State Machine FSM for modelling since it can model discrete events, communication and corresponding actions. Also an FSM model may be employed for system verification and validation. We have presented a collection of state patterns for web service workflow and communication modelling, and demonstrated how these patterns can be translated into BPEL and WSDL primitives[1]. States in these patterns are marked with information useful for obtaining BPEL and WSDL implementations hence called AFSM.

Given the importance of the situation, maintaining software consistency is always a top priority. Verification is performed throughout the software development lifecycle, beginning with requirements engineering and ending with ongoing maintenance. This exemplifies the importance of developing high-quality systems for users’ happiness and the reputation of the developers who made them. The ultimate success of a business service is predicated on its faultless execution. This highlights the significance of web service verification. Verification becomes more challenging when dealing with composite services due to the aggregation of the behaviour of the individuals involved.

Web Services offer the potential to improve B2B collaboration by merging compatible services into a higher-dimensional service. Matching conversation protocols of participating services is essential for service composition. This can give rise to new business standards that enable all types of organisations (especially small and medium-sized businesses) to reach out to clients by putting service information on the web and by forming a consortium to provide sophisticated services.

Web services can be published and discovered via repositories like UDDI (Universal Description, Discovery, and Integration) [4] to improve service composition.

By orchestrating current services into a new service that implements a business process, Service Composition can be achieved. A business process consists of a series of message exchanges (i.e., the flow of messages) between services composing the process (choreography). Search of services must therefore not only assure the exchange of specified messages, but also conform to the sequence of message transactions among the services that compose the search. This requires the modelling of business process communications and business logic. Diverse techniques, including EPC, Petrinet, and UML, have been proposed for defining web services and their compositions. We discovered that FSM is intensively investigated for modelling online services, particularly web choreography to ensure that a web service is communicating effectively with its collaborating services.

2. Related Work

Modeling, verification, and composition are the three research categories that are examined. These areas are investigated in order to locate relevant literature and are discussed here.

2.1. Service Modeling

Various techniques based on EPC [2][3], petrinet [4], processalgebra [5], CPN [7] and UML [6][8][9][10] have been proposed for modelling web services. Introduction of Event Process Chain (EPC) for business processes is presented in [11] modelled as graph composed of functions, events, and the control flow between them. [12] describes the translation of EPC process models to a technical language such as BPEL.

2.2. Service Verification

According to [5], the control flow of a BPEL process can be expressed in a process algebra BPEL-calculus, and then the process’ properties, such as reachability and deadlock freedom, can be analysed. In [9], we see the presentation of a framework named VERBUS for BPEL process verification. VERBUS incorporates the SPIN specification language and the SMV verification tool in addition to the BPEL4WS specification language. [2] gives a method for judging how well the choreography works with each partner.

The composition of Web services can be verified with the help of CPN tools, as proposed in [7], which involves converting WSBPEL to a CPN model. Reachability, boundness, dead marking, dead transition, liveness, home, fairness, and dialogue can all be assessed with the use of CPN tools [10] thanks to the wide range of formal analytic approaches they provide. In [11] these attributes were used to verify Web service composition using the formal technique TLA (Temporal Logic Action) to describe, compose, select, and verify Web services.
Using Event Calculus formalism to explain and test the transactional behaviour consistency of service composition, [2] presents a seminal work on this topic. In another work, WS Timed Automata are employed for the same purpose [3].

The concept of timed automata is extended to emulate WS transactional behaviour, which, unlike process state transition, requires time to travel between states.

Uppaal, a model checker for WS Timed Automata, is used to perform formal verification on the BPEL4WS application once it has been converted to this format. Traces generated by algorithms like flow, switch, pick, and while. It is confirmed that Invoke, Throw, and other service execution paths are valid.

In a separate work [4], extended finite state machines are used to model service dialogues. It verifies a protocol that is precisely and concisely stated in Service-Oriented Description Language (SODL) for the message processing logic in web service protocol implementations. It is also used to develop and deploy service protocols automatically.

According to [5], there is an additional model-based method for behavioural verification. Control verification and operational behaviour verification are two means by which the issue is distinguished. Essentially, it determines whether or not these two types of behaviour are compatible. Message forwarding facilitates this compatibility by employing a functional behaviour. Temporal logic is employed to specify control aspects and evaluate functional performance.

### 2.3. Service Composition

Conversation protocols have been modelled and matched using FSM[13][14]. Nonetheless, we believe that storing, matching, and retrieving FSMs using graph structure is not scalable for representing conversation protocol for the thousands of web services that will soon be deployed.

We propose a relational model to store FSMs and a query system to retrieve services that match a given service in order to meet the challenge.

Differentiating our work from [15] is the consideration of message transactions. For the functionality of a composed service, it is crucial that their message communications match. In spite of matching input and output, a composition will fail to match if their communication is mismatched. When matching, we consider both the order and significance of messages by defining types such as mandatory and optional. To make the technique scalable, we have utilised a relational database to store communication details, similar to [15]. Each service's message communications are stored in databases, and matching between services is accomplished by joining two tables. The join operation is relaxed to provide Full Match, Partial Match, and No Match matching results. We have also discussed implementation options for the proposed method. We believe that the combination of our method and the method proposed in [7] will result in a service composition that is precisely realisable.

### 3. AFSM for Modelling Web services

Professional organisations such as OMG, BPMI, W3C, and WfMC, as well as numerous businesses and academic institutions, are developing numerous techniques for the specification and standardisation of business processes. Diverse approaches, such as EPC, PetriNet, UML, and FSM, have been proposed for specifying a business process with a focus on various aspects, such as business documentation, formal analysis, orchestration, and choreography. Reviewing recent works on various modelling approaches, we find that FSM is being actively studied for modelling web services, particularly web choreography to ensure that a web service is successfully communicating with its collaborating services. Using finite state machines, the behaviour of Web services is formalised (FSMs). Conceptually, a business process can be viewed as an FSM because it executes and transitions from one state to the next.

For modelling web services, the AFSM definition includes the constructs choice, flow, and synchronisation. However, improper application of modelling components leads to structural and behavioural issues. Given that structural errors are the primary causes of behavioural errors, [16] identifies and recommends a method for avoiding them. Several structural flaws have been identified, including unreachable activities in a service, deadlock issues caused by improper synchronisation usage, incorrect temporal restrictions, and missing option pathways.
The Annotated Finite State Machine (AFSM) was proposed as a means of representing business processes, with more complex processes being broken down into their component parts. External service interactions and internal business logic are both described using states and edges. The transition from one state to another is described using the syntax of the labelled edge: `<ServiceProvider>#<consumer>#<operationName> #<Context> #<SubContext>`.

For firm internals, consumer and service provider are same, whereas for message forwarding, the consumer is an external service provider. Here, context information is extracted from the operation and sub context from the service.

We specify a AFSM as a tuple `<S, T, A, d, s, F>` where

- **S**: set of states
- **A**: set of activities
- **d**: set of transitions
- **s**: Start state
- **F**: set of final states
- **T**: set of time constraints
- **d Î A # A # OP # C** where
  - **OP**: set of operations
  - **C**: set of Contexts and
  - **P**: set of service provider entities.

In addition to simple activities such as messaging, assignment, etc., the proposed specification can also be used to describe organised and complex activities such as parallel execution, synchronisation, and temporal specifications. Timed FSM was created by modifying the concept of basic FSM in order to specify temporal constraints. We extend AFSM to specify the behavioural characteristics of a web service. During execution, the executional traces of a web service are compared to its behaviour specification.

Here are some examples of temporal constraints that can be defined over transitions involving a temporal variable t.

- **Condition 1**: The activity's clock is initialised with (t:=0)
- **Condition 2**: An activity with (t=#time) waits until the system clock exhibits a specific time.
- **Condition 3**: The action must be completed prior to the system clock displaying the specified time. (t<#time)

This AFSM has been augmented with additional information that provides context for its use. Each transition is denoted using context. For transition labelling, the following syntax is utilised: `Service Provider># Consumer># operation># <Context><Subcontext>`, where sub context information is derived from the service and context information is derived from the operation.

### 4. Service Matching and Composition

The business logic of a service is driven by business communication. In [3], we demonstrated that both can be modelled using FSM. Here, we are only concerned with modelling business communications according to FSM, as depicted in Figure 1.

Each node identifies a state and a communication edge. A label on the edge follows the format sender#recipient#message.

Figure 1 identifies two services, Book store and Shipping, and the sequence of events that occur from the time a book is ordered until it is shipped.
Consider the example of the Ordering a Book procedure in an online bookstore. The following sequence of events will transpire:

1) A customer requests a book from an e-book store (bs).
2) If the book is available, the bookstore sends a confirmation of its availability to the customer. If the book is unavailable, the customer is notified that it is unavailable.
3) The customer places a book order.
4) The Book Store responds with a message requesting payment.
5) The client makes payment.
6) The Book Store transmits the payment confirmation.
7) bs interacts with the shipping service provider (s) via a shipping message request.
8) The shipper responds to bs with availability information if shipping to the specified location is possible.
9) Then, bs displays the message Deliver the book.
10) Upon receiving the message Delivered, bs selects one of the payment methods (Credit Card) CCPayment or (Debit Card) DCPayment.
11) Upon receipt of the payment, the shipper sends a payment confirmation message.
12) Bookstore sends customer a Book Delivered message

This research aims to identify service partners whose message exchange sequence corresponds with that of the requesting service. According to the diagram, the Book Store service communicates with the consumer and shipper. In this case, shipper is the service collaborator with which Book Store collaborates. The states that are shaded involve shipper-to-shipper communication. The states that are shaded involve shipper-to-shipper communication. The current objective is to identify the shipper service whose message flow corresponds to that of the Book Store service.

Fig.1 depicts the AFSM representation of the Book Ordering process. Here, the mandatory messages are bs#c# Avail Conf, c# bs#Order Book, bs# c # Make Payment, bs# c # Pay Conf, c# bs # Book Delivered, and c#bs#Payment. The messages bs #s#CCPayment and bs#s#DCPayment are optional.

A match between two FSMs is considered to have occurred if and only if their intersection is not empty. For non-empty intersection, we will first perform the intersection operation on two FSMs, followed by an emptiness test on the resultant intersection FSM. If the emptiness test fails (i.e., the FSM accepts a string of messages), then there is a path from the start.
state to the final state in the intersected FSM. Therefore, two FSMs are matched. FSM-based matchmaking has the disadvantage that storing, matching, and retrieving FSMs using graph structure is not scalable when there are a large number of web services. For this reason, we look for a relational repository.

**4.1 RDBMS Approach for Scalability**

We have proposed an RDBMS schema for storing both messages and the communications sequence. The E-R diagram of the proposed schema is depicted in Figure 2. A single BusinessEntity can provide multiple BusinessServices, and each BusinessService has a corresponding ServiceMessageFlow description, as depicted in the diagram. Additionally, we have proposed the RequestorMessageFlow table, which specifies the format in which a client must send his or her request.

**BusinessEntity:** A BusinessKey uniquely identifies each instance of a BusinessEntity. The BusinessEntity's name and contact information (e-mail, WebURL) provide simple textual information.

**BusinessServices:** The ServiceKey identifies a BusinessServices entity uniquely. The BusinessKey feature uniquely identifies the BusinessEntity that is the BusinessService provider. Name, brief service description, and Category provide simple textual information about BusinessServices.

![Figure 2 ER Diagram](image)

A ServiceMessageFlow object is uniquely recognized by its ServiceKey, message, and followedByMsg properties. This table stores the flow of messages between services. The type field specifies whether a message is optional or mandatory. The snd rcv field indicates whether the message was sent or received by the customer.

**RequestorMessageFlow:** A RequestorMessageFlow instance is uniquely recognized by its msg and followedByMsg properties. The table contains the message flow of a requested service, which may be provided from the outside or produced from the ServiceMessageFlow table.

**Following is a summary of the tables in the proposed RDBMS structure.**

**BusinessEntity**
(BusinessKey, BusinessName, Phone, Fax, e-mail, URL)

**Provides**
(BusinessKey, ServiceKey)

**BusinessServices**
(ServiceKey, BusinessKey, ServiceName, Description, Category)

**Describes**
(ServiceKey, msg, followedByMsg)

**ServiceMessageFlow**
(ServiceKey, msg, followedByMsg, snd rcv, type, CPartner)

**RequestorMessageFlow**
(msg, followedByMsg, snd rcv, type)

Regarding the identical instance Book Ordering from Section 4. The FSM is shown with annotations in Figure 1. Table I displays the schema representation of the Book Store FSM corresponding to ServiceKey (SK) = 1. We suggest 6 column table; Where msg represents the communication message ByMsg is the previous message that was received, snd rcv
indicates if the message was sent or received, there are two sorts of messages, 'mandatory' (man) and 'optional' (op), and Communication Partner (CPartner) specifies the details of the service's partner. sk2 and sk3 are two versions of ServiceKey's shipping processes.

Let's begin with the top row. In this case, followedByMsg is NULL because there is no message that came before the Req Book message. And the rcv value is receive because customer(c) type is set to mandatory and BookStore(bs) is getting that message from customer(c).

In the same way, all the other rows in that table are added. The number of messages in FSM is equal to the number of rows in a table. CCPayment and DCPayment are put into the table as optional messages because, by default, if an unannotated FSM state sends out two or more messages, those messages are treated as optional.

Using the example table presented in Section 4, we have observed that a single service's message flow spans across numerous rows in a relational table.

<table>
<thead>
<tr>
<th>sk</th>
<th>msg</th>
<th>followedByMsg</th>
<th>snd_rec</th>
<th>type</th>
<th>CPart</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Req_Book</td>
<td>NULL</td>
<td>receive</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>1</td>
<td>Avail_Conf</td>
<td>Req_Book</td>
<td>send</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>1</td>
<td>Order_Book</td>
<td>Avail_Conf</td>
<td>receive</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>1</td>
<td>Make_Payment</td>
<td>Order_Book</td>
<td>send</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>1</td>
<td>Payment</td>
<td>Make_Payment</td>
<td>receive</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>1</td>
<td>Pay_Conf</td>
<td>Payment</td>
<td>receive</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>1</td>
<td>Req_Shipping</td>
<td>Pay_Conf</td>
<td>send</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>1</td>
<td>Ship_Avail</td>
<td>Req_Shipping</td>
<td>receive</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>1</td>
<td>Deliver_Item</td>
<td>Ship_Avail</td>
<td>send</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>1</td>
<td>Delivered</td>
<td>Deliver_Item</td>
<td>receive</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>1</td>
<td>CCPayment</td>
<td>Delivered</td>
<td>send</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>1</td>
<td>DCPayment</td>
<td>Delivered</td>
<td>send</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>1</td>
<td>Ship_Pay_Conf</td>
<td>DCPayment</td>
<td>receive</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>1</td>
<td>Ship_Pay_Conf</td>
<td>CCPayment</td>
<td>receive</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>1</td>
<td>Book_Delivered</td>
<td>Ship_Pay_Conf</td>
<td>send</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>2</td>
<td>Req_Shipping</td>
<td>NULL</td>
<td>receive</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>2</td>
<td>Ship_Avail</td>
<td>Req_Shipping</td>
<td>send</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>2</td>
<td>Deliver_Item</td>
<td>Ship_Avail</td>
<td>receive</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>2</td>
<td>DCPayment</td>
<td>Deliver_Item</td>
<td>receive</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>2</td>
<td>Ship_Pay_Conf</td>
<td>DCPayment</td>
<td>send</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>3</td>
<td>Req_Shipping</td>
<td>NULL</td>
<td>receive</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>3</td>
<td>Deliver_Item</td>
<td>Ship_Avail</td>
<td>receive</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>3</td>
<td>DCPayment</td>
<td>Delivered</td>
<td>receive</td>
<td>man</td>
<td>cust</td>
</tr>
<tr>
<td>3</td>
<td>Ship_Pay_Conf</td>
<td>DCPayment</td>
<td>send</td>
<td>opt</td>
<td>cust</td>
</tr>
</tbody>
</table>

Table I ServiceMessageFlow

Therefore, it is not possible for any type of query to perform match in RDBMS. Operation between two such messages spanning multiple rows flow sequences. We have therefore proposed an algorithm for Web service matching based on previously defined tables. Additionally, this algorithm is implemented [17] as a stored procedure in MySQL. Following the completion of the matching process, a service falls into one of the following three categories: Exact match, Partial match, or No match.

In the case of an exact match, the message flow sequences of both the service provider and the requestor are identical, or some optional messages from the requestor's message flow sequence may be absent from the service provider's message flow sequence. In the case of a Partial match, certain messages of the service provider's message flow sequence do not have matching messages in the requestor's message flow sequence. In the event of a mismatch, neither the service provider's nor the requestor's messages match, or certain required messages in the requestor's message flow sequence lack corresponding messages in the service provider's message flow sequence.

Tables 1 and 2 show two tables: the ServiceMessageFlow table, which contains three services with their message flows, and the RequestorMessageFlow table. Assume that only these three services are available and that they all fall into the same category. Let's look at the matching process between the service 2 message flow and the requestor message flow. Table III shows the full outer join table of the service 2 and RequestorMessageFlow tables.
The first three messages (rows) of service sk2 were identical to the requestor messages. According to the join table in Table III, there is a mismatch in the fourth message of sk2, and the join table shows that the followedByMsg's are different while the msg's are the same. Because there is a pause in the message exchange, both services can negotiate. As a result, it is a Partial Match. A few messages specified in the provider service may not be present at the requestor's end, in which case both the requestor and the provider can start negotiating and agree on the missing message. sk3 matches exactly with requestors. Hence the services can be matched.

5. Service Verification

For modeling web services, we proposed Timed FSM with the newly described choice, flow, and flow elements in addition to synchronisation. However, improper utilization of Modeling constructs leads to structural and behavioral errors. Possible errors that could be represented by the proposed modeling technique include:

- **Unreachable Activities**
- **Deadlock**
- **Improper temporal constraints**
- **Missing choice paths**

5.1 Unreachable Activities

Inappropriate use of synchronization constructs prevents the attainment of a state.
Fig 3 An example of unreachable activities

Considering the situational actions depicted in Figure 3, two groups of activities are in flow in the example. The second set's activity a6 can only occur after activities a1 and a2 have been completed. However, the FSM indicates that activities a1 and a2 are optional, meaning that, depending on the circumstances, only one of the pathways will be chosen and executed at any given time. Therefore, because the activities are synchronized, it is impossible to perform tasks a1 and a2 concurrently, and activities a6 and a8 cannot be completed. As structural flaws are the leading cause of behavioral errors, we are especially interested in discovering them. During service executions, contexts and their sequence of changes are extracted and compared to the specification labels that have been augmented with their respective AFSMs. The comparisons fall into three categories:

- Co-Context Checking
- Context Sequence Checking
- Context Timeliness Checking

Co-Context Checking

In Co Context checking, runtime contexts acquired from a service are compared to the desired contexts. The desired context is a collection of context information extracted from the AFSM model. Specification-based variations in the desired contexts for each service exist. For context checking, the contexts obtained from the service under execution are searched in the desired context; if the context is found in DC, the service is executing within the scope of the desired context. Otherwise, the service's behavior is not compliant with the specification.

Run-Time Context □ Desired Context

Context Sequence Checking

First, the context sequence is verified, followed by the subcontext sequence. Contexts received from a service are verified for their sequence using a contexts-specific behavioral FSM. This FSM is derived from context-annotated FSM parsing (C-AFSM). The states of the FSM are formed by the extracted contexts from C-AFSM, and the timing information along with the edges is preserved. Figure 4 depicts the behavioral FSM (BFSM) of contexts for the travel booking agent (tba). This BFSM is derived by parsing the context-annotated FSM of the travel agent.
Searching (S), booking (B), payment (P), and confirmation are the contexts here (C). Flight Searching (FS), Hotel Searching (HS), and Cab Searching are the sub-contexts of searching (CS).

This behavioral FSM generates all possible paths, and the obtained context sequences determine the desired context order (DC). This order includes the sequence of valid sub-contexts within a given context. The sequence of contexts generated by a service at runtime is represented as the order of runtime context (RTC). The representation of this context sequence verification is as follows.

\[ \text{Order} (\text{RTC}) \leq \text{Order} (\text{DC}) \]

Context Timeliness Checking
Checking the time-related information or time constraints defined in the AFSM is the purpose of context-time checking. The behavioral FSM is annotated with timing information extracted from the FSM service model. Timing information may be associated with context or sub-context. During runtime verification, when a service reaches a state where timing activity is mentioned in the FSM, a timer is started and initialized with transaction time obtained from the AFSM. If the service reaches the specified state within the allotted time, the behavior is valid; otherwise, a timeout exception is thrown.

\[ \text{Time}(c, \text{RTC}) \leq \text{Time}(c', \text{DC}) \]

The algorithm to verify the context time line checking is presented in [16].

These are the key steps in this algorithm:

**Step1:** Extraction of Time and Contextual Information
**Step2:** Verifying the Sequence in its Context and Subcontext
**Step3:** Checking Time-line
**Step 4:** Saving context.

The events generated by the service being executed are one of the algorithm's inputs. The remaining inputs, namely desired context and desired context orders, are obtained from the service specification in AFSM. An event detail comprises context, sequence number, and time; subcontext data is extracted from context. The algorithm executes for each event and continues until all events associated with a service instance are validated or until an event associated with context or sub-context is executed out of order.

Here we illustrate behavioral verification using the example of the travel booking agent. Search context, Booking context, Payment context, and Confirmation context are the four contexts that Travel Booking Agent (TBA) will use to process user requests. The context shown in Figure 5 begins with the Search context and concludes with the Confirmation context. As
depicted in Figure 4, the context-annotated AFSM is parsed and behavioral AFSM is generated. In this process, the contexts viz. Search, Booking, Payment, and Confirmation are identified namely D.contexts. Contexts and all possible paths are generated, and these paths comprise the D.Context order, which is one of the algorithm 1’s inputs. The service being performed will always be in one of the states depicted in Fig. 5. On reaching each of these states, the service generates context- and sub-context-specific events. This context data is fed into the algorithm for verification purposes. For example, if the context thrown is PFP, meaning the context is payment context and the sub context is Flight payment (Step1 of Algorithm), then the sub context is Flight payment (Step1 of Algorithm). Next, in Step2, the obtained context, i.e. Payment, is validated in the desired context set, and the context order is validated according to Fig. 4, sub scontext Flight. In addition, payment is examined for sequence compatibility with the context order (D.Context order) extracted from BFSM. Later, the timing information contained in AFSM (refer to Fig.1) is monitored in Step3 by determining whether the payment is processed within 10 minutes. If the service's behavior does not conform to the specification at any step, the verification algorithm would throw an error indicating why the behavior is invalid.

6. Integrated Framework

In this section, the integrated framework for structural and behavioral webservice verification is presented. From Figure 6 The initial model for the services is an AFSM, which indicates that all state changes are an integral part of business operations. The model is then structurally validated, and a valid model serves as input to the Generate Code module, which generates the executable BPEL and WSDL codes. These codes are also deployed on an executable engine, where the service monitor module will record and add all generated events to an EventsQueue. Timing and context checks are used to determine whether or not these occurrences comply with the specified behavior. If the contexts are not properly sequenced or the timing is off, an error report is generated to identify the cause of the error.

Fig 6 An integrated framework for service composition and Verification

7. Conclusion

In this paper Annotated FSM is proposed for service modelling, composition and verification. Modelling service communication in FSM and then extracting protocol details for storage in RDBMS and querying for finding matches (of different degrees). A new matching algorithm and message flow databases are added to the open-source jUDDI that aid composition.

A variety of error-prone behaviours that may occur when modelling business processes have been discussed. In addition, we proposed methods for automating model verification in order to detect errors. As a business process is executed in a temporal context, the significance of temporal constraints has also been taken into account. In addition, Web service behavioural verification is essential for ensuring that services are executed in accordance with their specifications. As a result, it facilitates service monitoring and failure case analysis. Each service accomplishes its tasks in one of two ways: either by delivering messages or by executing business logic in a context with associated sub-contexts. Contexts are extracted from the specification model (AFSM) and used in two ways: first, to model context-based behavioural FSM, and second, to provide test stubs that can be inserted into service code to report context changes while a service is executing. Throughout service execution, contexts are recorded at runtime and validated using behavioural AFSM. Context verification, context sequence verification, and timeliness verification are all components of the algorithm for behavioural verification. Runtime verification increases execution time because it is time-consuming. Therefore, we have proposed a method that, while not exhaustive, satisfies both user and design goals. Contexts and the order in which they change are utilized to develop this policy. Recent studies advocate constraint modelling in model-driven system development as a method for ensuring process constraint
conformance throughout the system lifecycle, which supports the concept we have proposed.

References


