Model-based Verification of Web Service Compositions

Howard Foster, Sebastian Uchitel, Jeff Magee, Jeff Kramer
Department of Computing, Imperial College London, 180 Queen's Gate, SW7 2BZ, UK
{hf1, su2, jnm, jk}@doc.ic.ac.uk

Abstract

In this paper we discuss a model-based approach to verifying web service compositions for web service implementations. The approach supports verification against specification models and assigns semantics to the behavior of implementation models so as to confirm expected results for both the designer and implementer. Specifications of the design are modeled in UML, in the form of Message Sequence Charts (MSCs), and mechanically compiled into the Finite State Process notation (FSP) to concisely describe and reason about the concurrent programs. Implementations are mechanically translated to FSP to allow a trace equivalence verification process to be performed. By providing early design verification, the implementation, testing and deployment of web service compositions can be eased through the understanding of the differences, limitations and undesirable traces allowed by the composition. The approach is supported by a suite of cooperating tools for specification, formal modeling and trace animation of the composition workflow.

1. Introduction

The Business Process Language for Web Services (BPEL4WS)[1] is an emerging standard for specifying and executing workflow specifications for web service composition invocation. The ability to perform such flow invocation for web services is the next step in the evolution of internet distributed computing. The move away from a point-to-point web services framework, which is currently used in an individual and isolated style, will be to a distributed yet coordinated service-oriented framework that can be directly or dynamically composed of web services through workflows [2, 3]. This move however, will also raise the awareness of how these compositions are verified for service use, potential deadlocks that could occur and requiring usability assessment of partnered services and other web service workflows. Whilst the technology alone provides a specification to compose and constrain flows of web service invocation, there is little to support the design process of such distributed service-oriented architectures. Therefore, of clear interest is the need to support such engineering tasks as process verification, partner service usability, and trace checking to verify the roles of web service users and their actions [4]. There is also high value in providing a simulated workflow mechanism to visually compare expected with simulated results of a workflow invocation which can increase expectations of a successful outcome prior to deployment [5].

In this paper, we describe a formal approach to modeling and verifying the compositions of web services workflows using the Finite State Processes (FSP) notation. To illustrate how these compositions are verified, we have constructed workflow scenarios using message sequence charts, together with a model checking tool to interactively verify the workflow behavior. These models can then be used to check BPEL4WS implementations. The paper is written as follows; Section 2 provides a background to the issues and a consideration of what that is required in web service workflow verification, and introduces a framework for our verification process. Section 3 specifies how web service workflows can be represented formally using model based design and the Finite State Processes specification. Sections 4 and 5 describe an example implementation of the workflow in BPEL4WS and how this translates to a FSP specification. Section 6 describes the verification process, using the previous sets of FSP, to illustrate how verification results can be used to correct invalid forms of behavior in composition implementation against the initial design.

2. Background

Web Service workflow languages aim to fulfill the requirement of a coordinated and collaborative service invocation specification to support long running and multi-service transactions. This is seen as an important element of making web services viable for wide spread use, and to provide a closer representation of business transactions in cross-domain enterprises. The effect of using similar earlier architecture styles has been prone to issues of semantic failure and difficulties in providing the necessary compensation handling sequences [6]. This has been attributed greatly to the strict binding of services with specific technologies. Where previously designers of the workflow had to work very closely with the developers of the technical solution, we now have a mechanism to support technology independent workflow service invocation and this provides opportunity for the designers to concentrate on exactly what is required from the workflow without hindrance from limitations of technical possibilities or effort required to implement. As web technology has evolved, the emphasis has been placed on providing ease of design and deployment, with WYSIWYG now the normal rather than the exception for rapidly building web served applications. This is equally applicable to the domain of web services. Even though we are concentrating from a view of systems to systems rather than actual human actors, the concepts are highly related. If we offer a web service, or a web service workflow we are interested in who will use the service, how they will use the service and what they will expect to be invocated when requesting the service. The verification of the workflow therefore needs to be concrete. Of interest to the BPEL4WS engineer is specifically which order the requests are made and replies sent to the services of the workflow, but on a wider scale how the behavior of the implementation differs from that of the specification. Later in the paper we describe an example of a marketplace workflow service. The interaction in this is either by buyer or seller, yet in a web application there is an implied limitation to what these roles can perform in the functionality offered in the web page presented to them. With the web service version of this scenario, the buyer and seller are system requests. The order in which requests are made by these roles is implied in system design, yet not enforced explicitly with the service offered.

Whilst web service architectures aim to provide a technology independent means of integration, the ability to verify workflows is inherently not a technology issue but of state, behavior and identity [7]. With this in mind, we can utilize a UML design of these workflows away from a technical implementation, and evaluate their transitional state and behavior locally before deploying any parts of the workflow, and realizing the true effect of the request flow. If the model does not reflect an intended use, then the model can quickly be corrected, and a repeat test performed. Furthermore, model verification can also be used to identify parts of the process flow that have been implemented incorrectly, or perhaps have unforeseen property results. Whilst there have been other attempts to use model-checking techniques for reliable web service verification [8, 9], they have concentrated on property specifications in domain specific language notations (e.g. Promela, the implementation language of SPIN), whilst we provide verification from an abstract functional specification using MSCs.

2.1. Requirements for Verification Process

To enable modeling to be abstract yet concise to our requirements, we need a specification that can closely resemble workflow specification languages, and a tool to simulate from the specification created in this language. To facilitate the above requirements, we have selected two separate representations (for workflow and web service compositions) in our layers of abstraction, and provided additional layers for a tool for designers to model these representations in, and an engine to invoke the implementation. Figure 1 illustrates our layers of modeling abstraction.

Model Verification Tool
Workflow Independent Notation
Web Service Composition Notation
Web Service Workflow Engine

Figure 1. Layers of modeling abstraction

LTSA + Message Sequence Charts
Finite State Process Notation
BPEL4WS Service Specifics
BPEL4WS Engine

Figure 2. Layer implementations

Figure 2 lists our focus on the implementations of the layers described previously. LTSA, in layer 1, is a tool which provides a means to construct and analyse complex models of finite state process specifications. This tool, which is fully explained in [10], provides us with an opportunity to model workflows prior to implementation and deployment testing, and with the message sequence chart extensions [11] to easily model workflow scenarios, which can increase the expectation that web service workflow invocation will provide the necessary path of invocation in all states specified (e.g. reliably by eliminating deadlock situations). With message sequence chart and animator extensions, the tool can also provide a facilitator in simulating workflow specifications. Finite State Processes (FSP), in layer 2, is a textual notation (technically a process calculus) for concisely describing concurrent programs. FSP is designed to be easily machine readable, and thus provides a preferred language to specify abstract workflows. The constructed FSP can be used to model the exact transition of workflow processes through a modelling tool such as the Labelled Transition System

Analyzer (LTSA), which provides compilation of an FSP into a Labelled Transition System. BPEL4WS in the third and fourth layers, is the result of a series of work carried out by both industry and academia to support an internet based, XML specification for the workflow of web services represented on the internet using universal resource identifiers and service descriptors. The notation is based on XML and the engine designed for Web Service Architecture frameworks, and is defined as being a layer above the Web Services Description Language (WSDL)[12]. The earlier work, IBM's Web Service Flow Language (WSFL)[13] and Microsoft's XLANG[14], have contributed to the specification of BPEL4WS.

The verification process applied against these layers requires the following steps; 1) use LTSA-MSC to capture desired workflow behavior in the form of message sequence charts, 2) write a BPEL4WS implementation and translate BPEL4WS representation in to FSP, 3) perform abstraction mapping to provide activity label matching, 4) use model trace equivalence checks to detect possible additional (implied) scenarios that the model supports but that were left unspecified or undefined by the user, and 5) examine the trace results of the FSP model checking and iterate tracing resolution until no violations or deadlocks are discovered.

2.2. Verification Architecture

The verification architecture is formed from two viewpoints. A specification is created as part of the requirements for the composition. The specification consists of the upper two layers from our layered model specified in section 2.1., being that of MSCs, state modeling tool, and the FSP representation of the composition. Adjoined to these layers is a set representing the abstraction mapping between implementation and specification, which we discuss further later in this paper. The second viewpoint is from that of the implementation of a workflow. The implementation focuses on the lower two layers of the verification model and a translation of BPEL4WS to a FSP representation. With specification and implementation the bridge between these is by unison and comparison of the FSP representations. Figure 3 illustrates the verification architecture and process flow. When the engineer is satisfied that the BPEL4WS implementation meets the criteria for composition design, then a translation from the technically oriented implementation to the abstract notation can take place. Furthermore, by performing model checking on the design and implementation models trace equivalence can be performed using a defined abstraction mapping.

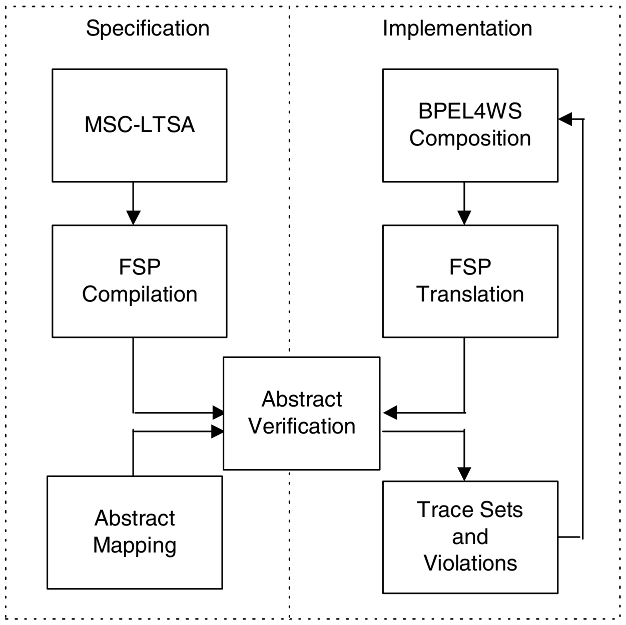

Figure 3. Model-based Verification Architecture

2.3. Composition Example

The example we have used in this paper is based upon a marketplace of buyers and sellers. The marketplace consists of a series of requests and replies, formed by the offering and requesting of products, request and offer of price for the products and confirmation of an iterative negotiation phase, which determines if an agreement of price for product is made.

The marketplace provides three stages to a negotiation. Firstly, a product may be either offered or requested. The message is passed from the seller or buyer role respectively, and is received by the marketplace service. Once a request is received, the marketplace instantiates a new transaction and awaits for either a seller or buyer to offer or request a similar product. This process matches a seller to a buyer. A seller cannot be matched to another seller, and equally a buyer cannot be matched to another buyer. When a match is recognized, the second stage is undertaken. The second stage of the negotiation is to receive initial prices from the partners, for when satisfied, allows the workflow to proceed to the third stage. The third stage provides an iterative negotiation of prices, with each partner able to specify a price and then place agreement as to whether a deal is made or terminated. In our composition we may be interested in verifying various scenarios, for example; to enter a negotiation phase, a seller and a buyer must specify an initial price required; a seller may only agree or disagree once between subsequent buyer agreements or disagreements, and a seller may not change the price requested until the buyer price has been received for each iteration of the negotiation steps. A context diagram of the marketplace composition is illustrated in figure 4.

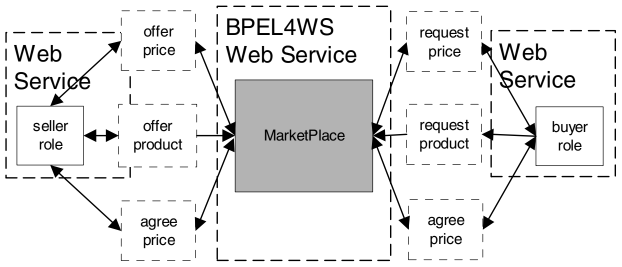

Figure 4. Marketplace Context Diagram

3. Modeling the Composition in MSCs

Modeling the composition in the MSCs is provided by the LTSA tool message sequence chart extensions (LTSA-MSC). The tool implements a framework for synthesizing implementation models for scenario-based specifications. By using this tool, we can guarantee that the resulting model is a model that implements the least unwanted behaviors. Additional features of this framework also allow us to check for implied behaviors. This feature is extremely advantageous for checking web service compositions, and for modeling send and receives message behavior. The goal of using this method of design is to find the differences between decomposition (our view of the system) and the actual system behavior specified in the implementation. To begin this composition modeling, we define a high level specification diagram. The specification diagram for the marketplace service example is illustrated in Figure 5.

The initial scenario (init) is the starting point of the workflow, and indicates that either a seller can offer a product, or a buyer can request a product. If either of these requests are made then the initial requestor must wait for the other to enter the negotiation phase. To specify the design actions for each possible sequence of our service we compose individual scenarios, such as for the actions of requesting or selling a product (figure 6), and seller and buyer agreements (figure 7). When there is a completed set of scenarios, the LTSA tool provides a FSP translation mechanism to convert each of the scenarios generated into a complete FSP specification.

The FSP specification can then be compiled into a state machine and an architecture model built to visually represent the complete possible state sequences allowed.

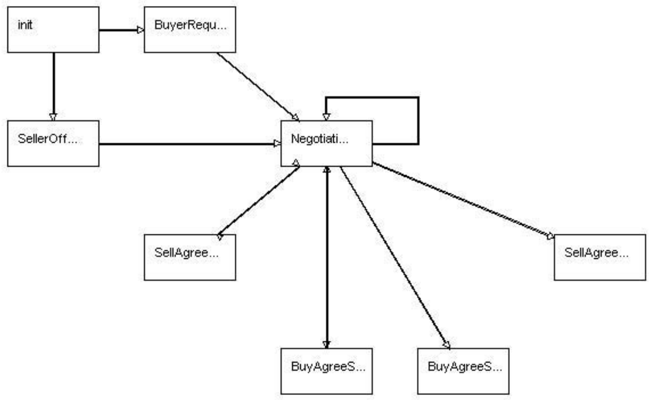

Figure 5. Specification of Marketplace Composition

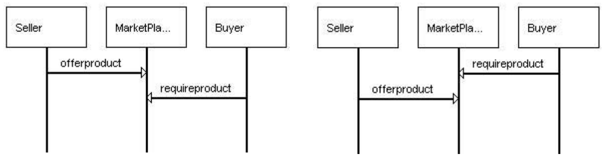

Figure 6. Scenario 1: Buyer or Seller requests

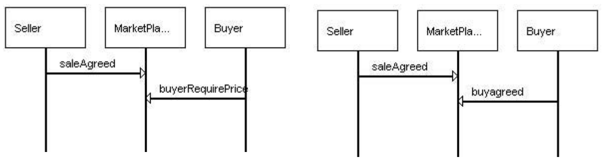

Figure 7. Scenario 2: Buyer or Seller agreements

Figure 8 illustrates the complete model for the marketplace example. The model provides a clear and concise representation of the workflow we intended to put in place for the service. The second step in the verification process is to build the composition service directly in the BPEL4WS notation.

4. Implementation in BPEL4WS

The BPEL4WS representation of the example is constructed from the viewpoint of a process. The process has partners, contains data elements and a series of activities. Here we describe how the design is implemented in BPEL4WS given the same elements used to construct the MSC representation. The BPEL4WS constructs and the corresponding elements of our message sequence charts are listed in table 1. Following this table we discuss each of the constructs and its relationship to concurrent system design and workflow specifications.

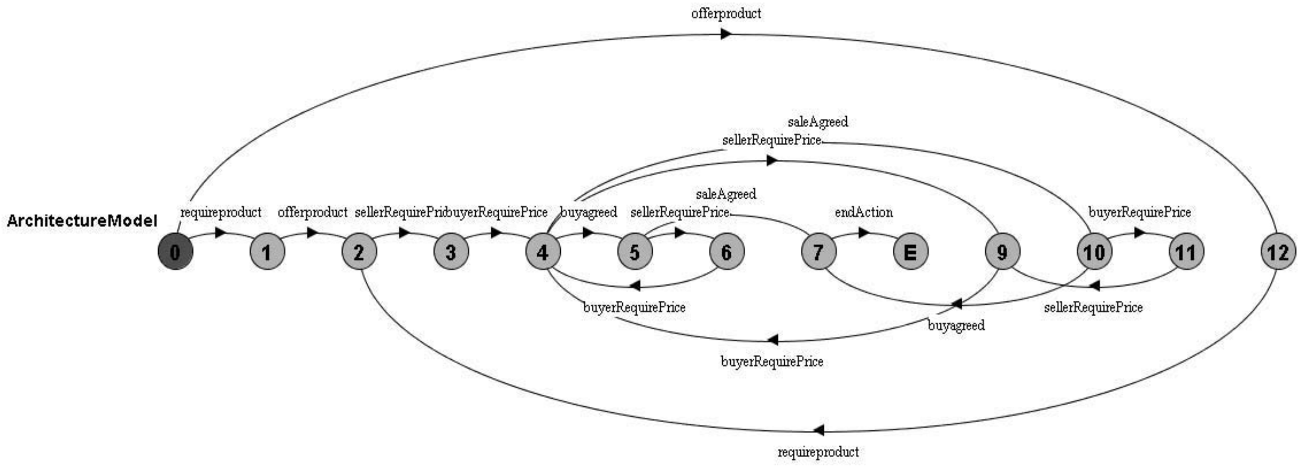

Figure 8. Specification architecture model

Table 1. BPEL4WS Process constructs

Process	Top level message sequence chart
Containers	specify variables for message data
Correlationsets	dependencies between messages
Flow	concurrent message transitions
Assign	messages that require data storage
Switch	test and branch conditions
Sequence	sequential execution of activities
While	scenario iterations
Receive	messages input from services
Reply	messages conveyed back to services

4.1. Process and Partners

The process is given the same name as our specification model. In the marketplace example, the name is marketplace. The additional targetnamespace and xmlns entries also default to this name. The instances in our message sequence charts correspond to the partners defined in the BPEL4WS specification. Our example models the seller and buyer roles, so these are specified as our process partners.

4.2. Containers and Correlationsets

The containers provide data storage variables for the information passed between partner service and the marketplace service. These containers are required for each of the messages in the message sequence charts. For example, the seller product offered is assigned to the sellerInfo container, whilst the buyer product required is assigned to the buyerInfo container. The negotiation phase is modeled on both price and agreements. The correlationsets define the linking of messages between service partners. The correlationsets defined for the marketplace example are for the negotiatedItem (product) and negotiatedPrice (initial price offered and later required).

4.3. Sequences and Loops

Sequences and Loops in BPEL4WS are specified using the <sequence> and <while> constructs. In our marketplace example we have defined several sequences. The sequence construct is used wherever a series of activities need to occur sequentially, although they may be contained one or more times within looping or concurrent construct activities. For example, in our marketplace example, we have defined a loop for the negotiation phase, specified using the <while> construct. Within this activity, we have defined two sequences, each with a set of activities to be performed on price and agreement negotiations.

4.4. Concurrency

Concurrency in BPEL4WS permits us to model the concurrent transitions in the message sequence charts. In BPEL4WS, this is specified using the <flow> construct.

4.5. Receives and Replies

The sending of messages uses two primitive activities in the BPEL4WS specification. <receive> provides a mechanism to wait for a message from a given partner, whilst <reply> allows the workflow to respond to the given request. In this way, this can be seen as the "request-reply" messaging model.

4.6. Assignments and Conditional Branching

The final construct in our BPEL4WS example specifies how we defined the assignment and branching of the workflow based upon values in data containers. The <assign> construct is used to assign a value to a container. The <switch> statement provides us with the conditional branching based upon the comparison of either static or container based data. The marketplace switch used in our example compares the offer price with that of the selling price and performs one of two sequences depending on the comparison result. The assign statement is also used throughout the BPEL4WS implementation to initialize the container values where used in conditional branching and looping.

4.7. Example BPEL4WS for a Market Place

A section of the BPEL4WS constructed for our marketplace is given below, illustrating the concurrent agreement receive requests.

```
<process name="marketplace"
     targetNamespace = "urn:MPService"
        xmlns:tns="urn:MPService"..>
     <flow name="MarketplaceAgree">
     <receive partner="seller"
            portType="tns:sellerPT"
            operation="submit"
            container="sellerAgree"
            createInstance="yes"
            name="SellerAgree">
     </receive>
     <receive partner="buyer"
            portType="tns:buyerPT"
            operation="submit"
            container="buyerAgree"
            createInstance="yes"
            name="BuyerAgree">
     </receive>
     </flow>
</process>
```

5. BPEL4WS Translation to FSP

To enable the verification of the BPEL4WS against the MSC representation, we need to translate the technical dependant BPEL4WS XML notation to the independent but easily machine readable FSP notation. To ease this translation we define the constructs of BPEL4WS as one of four groups. Structured represents the traditional structured design principles of sequence, selection and iteration, Concurrent specifies the parallel activities, whilst Primitive Actions specifies those actions that are atomic. Error or Compensation actions provide mechanisms for fault tolerance in workflow state transition. We have listed in table 2, how the BPEL4WS process tokens expressed in [1], can be grouped together by the construct groups discussed.

Table 2. BPEL4WS Process Token Groups

Structured	Concurrent	Primitive	Recovery
Sequence	Flow	Receive	Scope
Switch		Reply	Compensate
While		Invoke	faultHandler
Pick		Throw	
		Empty	
		Terminate	
		Wait	
		Assign	

We represent the first two of these construct groups in the FSP specification. A list of transitions is presented in table 3 for use as a basis for the BPEL4WS to FSP conversion.

Table 3. BPEL4WS to FSP Translation

BPEL Token, Example and FSP Representation	
<sequence> <receive name="act1"> </receive> <receive name="act2"> </receive> </sequence>	ACT1 = (action1 -> END). ACT2 = (action2 -> END). SEQUENCE = ACT1;ACT2;END.
<switch name= "MPS"> <case condition= "cond1" = "true">…… [act1] <otherwise>…[act2] </switch>	SWITCH = if cond1-true then ACT1;END else if cond2-true then ACT2;END else END.
<while condition = "cond1" = "true"> <sequence>……… </sequence> </while>	WHILE = If condition-true then ACT1;WHILE else END.
<pick name ="pick1"> <onMessage> <invoke ACT1>… <onAlarm> <invoke ACT2>… </pick>	PICK1 = (event1 -> ACT1; END | event2 -> ACT2; END).
<flow name="flow1"> <receive name="act1">… <receive name="act2">… </flow>	||FLOW1 = (ACT1 || ACT2).

5.1. Containers and Value Comparisons

Each container can be translated from BPEL4WS to the FSP process declarations with parameters. Here the value change of the container can be represented using a range parameter, and subsequent references to the container can assign differing values in that range. Write and Read functions are represented in a Pick of possible transitions, as illustrated in the following FSP.

```
CONTA [i:IntRange] = (write[j : IntR]
-> CONTA[j] | read[i]->CONTA[i]),
```

5.2. Sequence

The Sequence token used in BPEL4WS is defined in FSP as: If x is an action and P a process then the action prefix (x->P) describes a process that initially engages in the action x and then behaves exactly as P. An example sequence representation in FSP is:

```
MarketPlace = (transaction -> END |
transaction->OFFERREQUEST;NEGOTIATION).
```

5.3. Switch

The Switch token in BPEL4WS is represented in FSP by the choice definition. This definition is formally given as: If x and y are actions then (x->P | y->Q) escribes a process which initially engages in either of the actions x or y. After the first action has occurred, subsequent behavior is described by P if the first action was x and Q if the first action was y. An example of the switch representation in FSP is:

```
CHECKPRODUCT = if sellerproduct =
buyerproduct then OFFERREQUESTPRICE ;
END else END.
```

5.4. While

The While token in BPEL4WS, is represented in FSP by a combination of both the conditional elements of the Switch token and the sequence operator. For example:

```
MarketPlace = if newtransaction then
MarketPlaceItemFlow;
MarketPlacePriceFlow;MarketPlaceSwitch;
NEGOTIATE_FLOW;WHILE else DEALMADE.
```

5.5. Pick

As with the Switch token in section 5.2, the Pick token is represented in FSP by the choice definition. The BPEL4WS implementation however does not explicitly use the Pick statement, as this is reserved in BPEL4WS for detection of events being triggered.

5.6. Flow

The Flow token in BPEL4WS is represented in FSP by the parallel composition definition. This definition is formally given as: IF P and Q are process then (P || Q) represents the concurrent execution of P and Q. The operator || is the parallel composition operator. In the BPEL4WS implementation of our design, we have used the Flow statement to represent concurrent execution of receiving and replying to requests from seller and buyer. The translation to FSP for the initial price request is:

```
BR =(receive.submit.buyerrequest -> END).
SR = (receive.submit.selleroffer -> END).
||MarketPlaceItemFlow = (BR || SR).
```

5.7. Summary of BPEL4WS FSP

By examining the differences between BPEL4WS and FSP we can see how the technical and abstract notations are positioned from a programmatic deployment approach for BPEL4WS and from a simple yet concise state and process perspective for FSP. With FSP as our middle layer, the abstract notation can be used to specify the composition and workflow without hindrance from technical limitations, and the design model can be constructed and verified independently. Whilst the BPEL4WS specification requires detail of specific implementations (such as referencing the location of the services involved, and who will use them), we can remove these specifics and detail the workflow from an abstract view. Using transition table 3, the BPEL4WS specification of the marketplace example can be translated as follows, although it should be noted that this is just one possible representation of several. For example, in this FSP representation we have replaced the use of conditional variables and converted the BPEL4WS Switch to a series of Pick representations. When compiled in the LTSA tool, a graphical architecture model can again be produced (see Figure 9). At this stage, we have prepared an initial design using the message sequence charts and compiled this into an FSP representation. We have also migrated from design to BPEL4WS and translated the BPEL4WS to an FSP representation. Given these two sets of FSP representations, we can move on to describe how the verification process compares the FSP for the MSCs and the FSP for the BPEL4WS relationships.

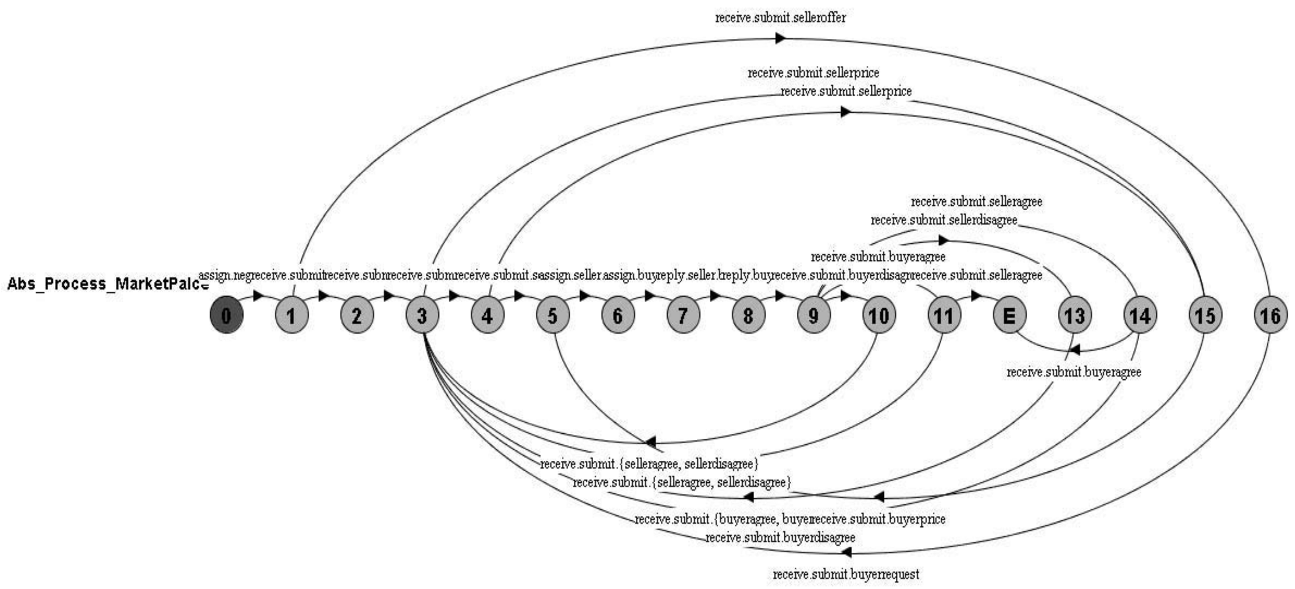
Abs_Process_MarketPalce
0
1
2
3
4
5
6
7
8
9
10
11
E
13
14
15
16
receive.submit.selleroffer
receive.submit.sellerprice
receive.submit.sellerprice
receive.submit.selleragree
receive.submit.sellerdisagree
receive.submit.buyeragree
receive.submit.buyeragree
receive.submit.{selleragree, sellerdisagree}
receive.submit.{selleragree, sellerdisagree}
receive.submit.{buyeragree, buyereceive.submit.buyerprice
receive.submit.buyerdisagree
receive.submit.buyerrequest

Figure 9. Implementation Architecture Model

6. Verification Process and Results

The verification process for our model-based design is focused on composing joint sets of labeled transition systems using the FSP specifications constructed earlier in the modeling process. The essence of the verification mechanism is to check trace equivalence. Any violations exhibit traces of actions that could occur from the state machine generated from one FSP specification, over that of the other. The verification process is broken down into several sections. Firstly, we need to identify and match labels that have the same semantics and relabeled where necessary to join these as a single label represented in both specifications. If any actions have been included in the BPEL4WS specification that are dependent on actions undertaken for BPEL4WS implementation specifics, then these must be hidden as a secondary step, so as not to be included in the safety checks and traces. The verification process used to undertake this is illustrated in figure 10.

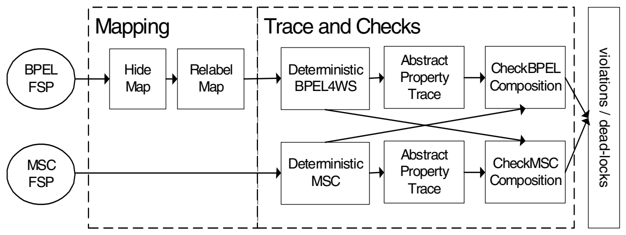

Figure 10. Verification Process Map

6.1. Abstraction Mapping

As part of the verification process, the BPEL engineer must map the activities specified in the BPEL4WS implementation to those represented in the MSC specification. To achieve this we need to address re-labeling differing action naming conventions. This however, limits us in future work to provide an automated verification mechanism, as it is anticipated that designer and implementer may use different conventions. As an example of the relabeled transition process specification, the marketplace example gives the following operation for the BPEL4WS specification:

```
/{requireproduct/receive.submit
.buyerrequest, offerproduct/
receive.submit.selleroffer}.
```

In addition to re-labeling, we must also consider actions in the BPEL4WS that are solely for implementation and do not represent an action in the specification. This will typically be actions in the BPEL4WS that are concerned with assignments, switch conditions, end actions and initiators. To specify this in FSP, the \ operator is used. The hiding for the marketplace example was given as follows:

```
\ {receive.submit.buyerdisagree,
receive.submit.sellerdisagree,
assign.negotiation.false,
assign.seller.price,
assign.buyer.price,
reply.seller.buyerprice,
reply.buyer.sellerprice,transaction}.
```

6.2. Deterministic and Trace Compositions

To perform trace equivalence of the prepared FSP specifications, two checks are performed. The first checks if the BPEL4WS FSP provides possible traces that the MSC FSP cannot. The second checks if the MSC FSP provides possible traces that the BPEL4WS FSP cannot. Both trace inclusion checks are performed as follows. If a model A is to be checked against a model B for trace inclusion, then B is made deterministic while preserving trace equivalence using the "deterministic" keyword of LTSA. Then a property is declared using the deterministic version of B using the keyword. This keyword adds to the model an error state and transitions that make actions that are not enabled in states to transition to the error state. Thus, the resulting model, if composed with A does not constrain its behavior and will have a reachable error state if A can perform traces that B cannot. The FSP code that follows shows how the BPEL4WS FSP is checked for trace inclusion against the ArchitectureModel. Process DetA is the deterministic version of the MSC FSP model, A is the property resulting from DetA and CheckBPEL is the process on which reachability of the error state is performed.

```
deterministic ||DetA =
      ArchitectureModel \{endAction}.
property ||A = DetA.
||CheckBPEL=A||Abs_Process_MarketPlace).
```

6.3. Assessing and Resolving Violations

The LTSA tool provides a “safety check” feature that, provided with a composition identifier, will perform a reachability analysis for a given specification. Here, if a violation is discovered, we can follow through the transitions and determine how the violation occurred given our FSP specification. This counter example represents a possibility that cannot occur in our MSC but could occur in the BPEL4WS FSP specification. Two such traces are listed in Table 4, as checks on our marketplace composition.

Table 4. Traces from BPEL safety check

1st Trace	2nd Trace
offerProduct	offerProduct
requireProduct	requireProduct
buyerRequirePrice	sellerRequirePrice
	buyerRequirePrice
	buyAgreed
	recieve.submit.sellerDisagree

From the first trace of the CheckBPEL property check, we are able to determine that a sequence of requests made by the clients of our web service composition in the order of offerproduct, requireproduct and buyerRequirePrice is allowed in our BPEL composition but not allowed in the MSC specification. To correct this we revisit the BPEL and locate the buyerRequirePrice receive. In our BPEL implementation we had used a FLOW for price requests. Either buyer or seller may have requested first. Therefore, we change this to a SEQUENCE so that the seller price must be received before buyer price can be received. From the second trace, we can observe that through the sequence of requests, the buyer has agreed, yet the seller submits a disagreement. This is a breach of our property which if a buyer agrees to a price then the seller cannot subsequently disagree. Again, we change this in our BPEL to reflect the buyeragreed action can only transition to a successful outcome of a negotiation.

From these traces, we are able to determine if there is a breach of the behavior specified in our FSPs. This violation can be resolved by either changing our BPEL4WS specification and FSP, or if it is a violation which needs validation from users, it may be something which is subsequently corrected in the MSC specification. The nature of this “trace and fix” iterative process is part of a wider web service engineering lifecycle, and fits conveniently in with a proposed lifecycle of web service composition engineering. One such lifecycle is suggested in [15]. This lifecycle encompasses DAML-S (a service description language) which provides a process model for service discovery, description and selection. The authors of this paper also discuss how the principle of the service interaction may also allow for participating services to enquire at a greater depth in to another service’s process model. It is perhaps the result of our verification process, that trace and other models could be added to an advertised service description, and thus allow for greater metadata inspection.

6.4. Verification Process Automation

To supplement the process described in this paper, we have developed a plug-in module for the LTSA tool which facilitates writing BPEL4WS specifications and using an integrated development environment, the BPEL4WS implementation can be translated to FSP and compiled into a model. This eases the translation process and also provides a useful mechanism for implementers to visually realize how the BPEL4WS is represented. The automation is limited however, by the manual requirement of the abstract mapping between

BPEL4WS actions and MSC labels, whereby semantic detail must be applied by knowledge outside that given in the automated technical solution. For further information and to download the plug-in with LTSA, please refer to LTSA homepage at http://www.doc.ic.ac.uk/ltsa.

7. Conclusions

BPEL4WS provides an initial work for forming an XML specification language for defining and implementing business process workflows for web services. The use of this technology provides an example of how distributed system computing using web services will be specified for web service workflow invocations, yet it is important to compose the service workflow correctly for all service actors and more importantly, verify this flow before actual implementation and deployment is undertaken. Technically it is evident that BPEL4WS lacks in verification of service provider and client use, yet with a verification process such as the one described in this paper, value can be added to the development process by early verification through a model checking process.

Our contribution is to address these issues by modeling these workflows in an accessible and concise notation, which can then be used to verify, not only web service workflows but any workflow processes involved. In this paper we presented an approach, which specifically addresses adding semantic representation to the BPEL4WS notation, and have identified a tool for verification of implementations against abstract functional specifications. Furthermore, by automating the process specified in this paper, a framework can be built to support modeling, verification and implementation in the notation of choice. We have chosen BPEL4WS to use as an example, yet other web service workflow specifications may be introduced. The approach also provides scope to enhance the verification of BPEL4WS implementations by analyzing and determining the impact of fault tolerance and compensation actions, addressing the concerns of availability and reliability. With this in mind, we foresee the findings presented here as an initial work that will provide a foundation for further ideas and contributions on these wider issues.

8. Acknowledgements

The authors would like to acknowledge that this research was supported, in part, by the STATUS ESPIRIT project (IST-2001-32298) and by the EPSRC READS project (GR/S03270/01).

9. References

[1] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana, "Business Process Execution Language For Web Services, Version 1.0," 2002.

[2] D. Chakraborty and A. Joshi, "Dynamic Service Composition: State-of-the-Art and Research Directions," University Of Maryland, Baltimore, Technical Report TR-CS-01-19, December 19 2001.

[3] P. Wohed, W. M. P. v. d. Aalst, M. Dumas, and A. H. M. t. Hofstede, "Pattern Based Analysis of BPEL4WS," Queensland University of Technology, Brisbane, Technical Report 2002.

[4] R. Akkiraju, D. Flaxer, H. Chang, T. Chao, L.-J. Zhang, F. Wu, and J.-J. Jeng, "A Framework for Facilitating Dynamic e-Business via Web Services," presented at OOPSLA 2001 - Workshop on Object-Oriented Web Services, Tampa, FL, 2001.

[5] C. Karamanolis, D.Giannakopoulou, J.Magee, and S.Wheater, "Modelling and Analysis of Workflow Processes," Imperial College of Science, Technology and Medicine, London 1999.

[6] O. Bukhres and C.J.Crawley, "Failure Handling in Transactional Workflows Utilizing CORBA 2.0," presented at 10th ERCIM Database Research Group Workshop on Heterogeneous Information Management, Prague, 1996.

[7] P. Hruby, "Specification of Workflow Management Systems with UML," presented at OOPSLA Workshop on Implementation and Application of Object-oriented Workflow Management Systems 1998, Vancouver, BC.

[8] S. Nakajima, "Model-Checking Verification for Reliable Web Service," presented at OOPSLA 2002 Workshop on Object-Oriented Web Services, Seattle, Washington, 2002.

[9] S. Narayanan and S. A. Mcllraith, "Simulation, Verification and Automated Composition of Web Services," presented at Eleventh International World Wide Web Conference (WWW-11), Honolulu, Hawaii, 2002.

[10] J. Magee and J. Kramer, *Concurrency - State Models and Java Programs*: John Wiley, 1999.

[11] S.Uchitel and J. Kramer, "A Workbench for Synthesising Behaviour Models from Scenarios," presented at the 23rd IEEE International Conference on Software Engineering (ICSE'01), Toronto, Canada, 2001.

[12] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL) 1.2," W3C, 2003.

[13] F. Leymann, "Web Services Flow Language (WSFL 1.0)," IBM Academy Of Technology 2001.

[14] S. Thatte, "XLANG - Web Services For Business Process Design," Microsoft Corporation 2001.

[15] S. A. Mcllraith and D. L. Martin, "Bringing Semantics to Web Services," *IEEE Intelligent Systems*, 2003.