

EAST WEST UNIVERSITY

Service Oriented Product Line: Analysis and Verification

Submitted by

Yasir Taher

and

Amit Sarker

Supervised by

Dr. Shamim H. Ripon

A project submitted in partial fulfillment for the degree of
B.Sc. in Computer Science and Engineering

In the

Faculty of Science and Engineering

Department of Computer Science and Engineering

May 2015

Declaration

We hereby declare that, this submission is our own work and that to the best of our knowledge and belief it contains neither material nor facts previously published or written by another person. Further, it does contain material or facts which to a substantial extent has been accepted for the award of any degree of a university or any other institution of territory education except where an acknowledgement.

(Amit Sarker)

(Yasir Taher)

Letter of acceptance

The project entitled “**Service Oriented Product Line: Analysis and Verification**” submitted by Amit Sarker (2011-1-60-033) and Yasir Taher (2011-1-60-011), to the Department of Computer Science and Engineering, East West University, Dhaka, Bangladesh is accepted by the department in partial fulfillment of requirements for Award of the degree of Bachelor of Science in Computer Science and Engineering on May, 2015.

Board of Examiners

Dr. Shamim H. Ripon

Associate Professor and Chairperson,

Department of Computer Science and Engineering

East West University, Dhaka, Bangladesh

ABSTRACT

In computer science and software engineering, re-usability is the use of existing assets in some form within the software product development process. A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. A service-oriented architecture is essentially a collection of services. These services communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity. Some means of connecting services to each other is needed. Web Service technology provides a platform on which we can develop distributed services. The interoperability among these services is achieved by various standard protocols. Web service composition originated from the necessity to achieve a predetermined goal that cannot be realized by a standalone service. Internally, in a composition, services can interact with each other to exchange parameters, for example a service's result could be another service's input parameter. We need verification of service composition to check whether there is any errors/deadlock in the composite service and are there any difficulties in the composite service? There are two types of service composition and we focused only on service choreography. To verify the service composition we used process algebraic technique, FPS. We model the service choreography in FSP and then use LTSA tool to animate the transitions and verify the composition.

Contents

Acknowledgements	VIII
1. Introduction	01
1.1. Introduction and Motivation	01
1.2. Objectives	03
1.3. Contribution	03
1.4. Outline	04
2. Background	05
2.1. Web Service Composition	05
2.2. Choreography	05
2.3. Orchestration	06
2.4. Comparison	06
2.5. Petri net	07
2.6. Atomic Process	07
2.7. Sequence Process	08
2.8. Split Process	09
2.9. The Split-Join Process	09
2.10 Label Transition System (LTS)	10
2.11 Finite State Process (FSP)	12
2.12 Primitive Processes	12
2.13 Composite Processes	17
3 Software Product Line Feature Analysis	19
3.11 Software Product Line	19
3.12 Feature Model	19
4 Service Orientation	22
4.11 Services and Comparison with features Comparison with feature	22
4.12 Service Orientated Product line	25

5	Web Service Choreography	28
5.11	Introduction.....	28
5.12	Buyer.....	28
5.13	Lender Web Service.....	29
5.14	Supplier Web Service	29
5.15	Broker Web Service	30
6	Choreography Analysis	33
6.11	Petri Net Representation	33
6.12	FSP Representation.....	35
6.13	Model.....	35
6.14	LTS Analysis	37
7	Conclusion	45
7.11	Summary	45
7.12	Future work.....	45
	Appendix.....	46
	References.....	54

List of Figures

2.1: Orchestration vs Choreography	06
2.2: Atomic process model	08
2.3: Sequence model	08
2.4: Split model.....	09
2.5: Split-Join model.....	10
2.6: Example of LTS.....	11
2.7: LTS of Action prefix.....	13
2.8: LTS of deterministic choice.....	13
2.9: LTS of non-deterministic choice	14
2.10: LTS of STOP	14
2.11: LTS of Indexing.....	15
2.12: LTS of Conditional	15
2.13: LTS of Guard.....	17
2.14: LTS of Parallel Composition	18
2.15: LTS of Relabeling	18
3.1: Feature tree of a broker system.....	21
5.1: Architectural view of Buyer	28
5.2: Architectural view of Lender web service	29
5.3: The Architectural view of supplier web service	30
5.4: The Architectural view of Broker web service.....	31
5.5: Architectural view of Broker System	32
6.1: The Petri Nets Representation of Broker system web service.....	34
6.2: LTS of Buyer	37
6.3: LTS of BROKER.....	38
6.4: LTS of SUPPLIER.....	41
6.5: LTS of Loan.....	42
6.6: LTS of Broker System	43

List of Tables

Table 2.1: Different types of operations7

Acknowledgement

First of all Thanks to ALLAH for the uncountable blessings on me. Thanks to my Supervisor **Dr. Shamim H. Ripon** for providing me this opportunity to test our skills in the best possible manner. He enlightened, encouraged and provided us with ingenuity to transform our vision into reality.

Chapter 1

Introduction

1.1 Introduction and Motivation

In most engineering disciplines, systems are designed by composition that means building a system out of components that have been used in other systems. To achieve better software quality, more quickly, at lower costs, software engineers are beginning to adopt systematic reuse as a design process.

There are many types of software reuse process:

- **Application system reuse:** The whole of an application system may be reused either by incorporating it. Without changing into other systems (COTS reuse) or by developing application families.
- **Component reuse:** Components of an application from sub-systems to single objects may be reused.
- **Object and function reuse:** Software components that implement a single well-defined object or function may be reused.

The benefits of software reuse are to increase dependability, reduced process risk, effective use of specialists, Standards compliance, and Accelerated development.

A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. Software product lines are emerging as a viable and important development paradigm allowing companies to realize order-of-magnitude improvements in time to market, cost, productivity, quality, and other business drivers. Software product line engineering can also enable rapid market entry and flexible response, and provide a capability for mass customization.

SOA is considered to be the new paradigm for developing flexible and dynamic software solutions by using loose coupling of services with diverse operating systems, programming languages and other components of various platforms. The integration of these interoperable services around a business process allows systems development in environments that have to continuously change and adapt to new circumstances. [11] A service-oriented architecture (SOA) is essentially a collection of services. These services communicate with each other. It mainly used to create architecture based upon the use those services. The communication can involve either simple data passing or it could involve two or more services coordinating some activity. Some means of connecting services to each other is needed.

The term 'web services' describes a standardized way of integrating web-based applications using open standards (e.g., XML, SOAP and UDDI) over an internet protocol backbone. Used primarily as a means for businesses to communicate with each other and with clients, web services allow organizations to communicate data without intimate knowledge of each other's IT system behind the firewall. Instead of providing GUIs to users, web services share business logic, data and processes through a programmable interface across the network. In order to create a cross-organizational component in web services, flexible methods are needed to handle web service interfaces. To describe the composition of web services as well as the process flow, two terms are widely used: Orchestration and Choreography. [12]

A *service composition* is an aggregate of services collectively composed to automate a particular task or business process. To qualify as a composition, at least two participating services plus one composition initiator need to be present. Otherwise, the service interaction only represents a point-to-point exchange. Service compositions can be classified into primitive and complex variations. In early service-oriented solutions, simple logic was generally implemented via point-to-point exchanges or primitive compositions. As the surrounding technology matured, complex compositions became more common. Much of the service-orientation design paradigm revolves around preparing services for effective participation in numerous complex compositions. So much so that the Service compos ability design principle exists, dedicated solely to ensuring that services are designed in support of repeatable composition.

1.2 Objectives

Inspired by the growing interest in Service Composition of web services. So, here is the objectives of our project:

- Reuse of Service in Service Composition of Web service.
- Composition of Web Service in Service Oriented Product line.
- To analyze the composition of Web service.
- To verify the Composition of Web service.

1.3 Contribution

We have made the following contribution in this project:

- We have analyzed the feature of **Car Broker** web service by modeling Broker system product line. We first identified various components of the web service as well as the composition among the services.

- After analyzing various services of the Car Broker, We compared features with services and define service oriented product line.
- First we have modeled car broker web service choreography by using Petri net. We then have modeled the broker web service by using Finite State Process (FSP).
- We verified the choreography represented in Finite state process by using Labelled Transition System Analyzer (LTSA). We have checked dead lock/errors in the service composition and check the traces of the animation.

1.4 Outline

The report is organized as follows:

Chapter 2 gives a brief overview of web service composition, Petri Net, Finite State Process and Labelled Transition System. We describe different type of Compositions and comparison between them. We also describe different type of processes in Petri Net and different types of operation and processes of LTS and how they work within the examples.

Chapter 3 gives an overview of the Car Broker web service product line and describe its feature analysis.

Chapter 4 gives an overview of service and gives a brief idea about difference between services and features. Discusses about product line and finally describe all the services and their composition.

Chapter 5 gives a brief overview of all Web services of Car Broker web service product line.

In Chapter 6 we represent our model in Petri Net and Finite State Process then analysis the data and describe them.

Finally, in Chapter 7 we give a summary of the project and outline our future plans.

Chapter 2

Background

2.1 Web Service Composition

Service composition aims at providing effective and efficient means for creating, running, adapting, and maintaining services that rely on other services in some way. Web Service Compositions are formed from a singular problem domain which is “local” to problem domain owner. In structured system domains, the design proceeds from a specification, and there is a single entity that is ultimately responsible for the design and implementation of the system. Web service compositions, as compositions can be built locally yet aim to conform to global compositional constraints through the use of choreography and orchestration rules. This rule base is significantly involved in standard ways of both communication and domain understanding. The composition of web services could be static or dynamic. Both static and dynamic web service compositions can collectively be described as; *“A web service composition consists of orchestrated web services through a local process, itself potentially a service. Static web service compositions are known at design time and too limited to a composition at design time. Dynamic web service compositions are one or many compositions in which web services are known at run time, and which are discovered or their properties resolved based upon a criteria process set at design time.”* [10]

2.1.1 Choreography

The term choreography which is more collaborative and allows each involved party to describe its part in the interaction. Choreography tracks the message sequences among multiple parties and sources typically the public message exchanges that occur between web services rather than a specific business process that a single party executes.

2.1.2 Orchestration

Orchestration always represents control from one party's perspective. The term orchestration refers to an executable business process that can interact with both internal and external web services. The interactions occur at the message level. They include business logic and task execution order, and they can span applications and organizations to define a long-lived, transactional, multistep process model.

2.1.3 Comparison

The terms orchestration and choreography describe two aspects of emerging standards for creating business processes from multiple Web services. The two terms overlap somewhat, but orchestration refers to an executable business process that can interact with both internal and external Web services. Orchestration always represents control from one party's perspective. This distinguishes it from choreography, which is more collaborative and allows each involved party to describe its part in the interaction. Proposed orchestration and choreography standards must meet several technical requirements that address the language for describing the process workflow and the supporting infrastructure.

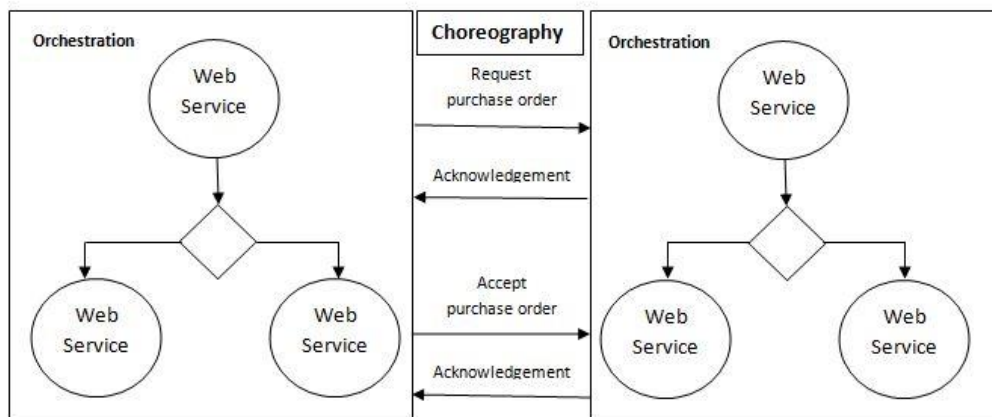


Figure 2.1: Orchestration vs Choreography

2.2 Petri net [1]

A Petri net is one of several mathematical modeling languages for the description of distributed systems. Petri nets is a graphical and mathematical modeling tool applicable to many systems. A Petri net is a directed bipartite graph, in which the nodes represent transitions and places. It is very useful tool for specifying information processing systems, which are describe as being asynchronous, parallel, concurrent, distributed or dynamical.




Places		some type of resource
Transitions		consume and produce resources
Tokens		One unity of a certain resource. The token tells us the state of the process

Table 2.1: Different types of operations

2.2.1 Atomic Process

The Petri nets model for an atomic process. Petri nets $PN = \langle S, T, I, O \rangle$ as shown in Figure 2.1, Where

$$S = \{s\};$$

It represents the service to be run when s includes.

Tokens.

$$T = \{ts, te\},$$

Where t_s represent beginning of the service and t_e represents ending of the service. $I(t_s, s)$ and $O(s, t_e)$ represent Input and Output respectively.

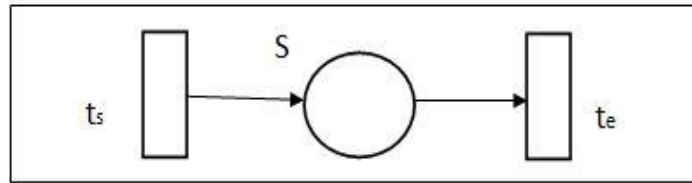


Figure 2.2: Atomic process model

2.2.2 Sequence Process

The petri nets model for sequence composition of web services is a hierarchical Petri nets $SPN = \langle S, PN, I, O \rangle$ as shown in Figure 2.2, where

$$S = \{s_1, s_2, s_3, \dots, s_k\}$$

$$PN = \{PN_1, PN_2, PN_3, \dots, PN_k\},$$

Where PN_i is a subset of i -th service.

$$I = \{I(PN_i, s_i), O(s_i, PN_{i+1}) \mid i = 1, 2, 3, 4, \dots, k-1\}.$$

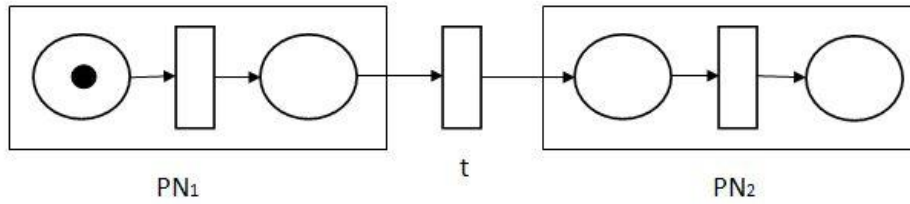


Figure 2.3: Sequence model

2.2.3 Split Process

The Petri Nets model for split composition of web service is a hierarchical Petri nets $SPN = \langle S, PN, I, O \rangle$ as show inFigure 2.3, where,

$$S = \{sin\}$$

$$T = \{t1\}$$

$$PN = \{PN1, PN2, PN3, \dots, PNk\},$$

Where PNi is a subset of i -th service.

$$I = \{I(sin, t1) \cup O(t1, PNi) \mid i= 1,2,3,4, \dots, k\}$$

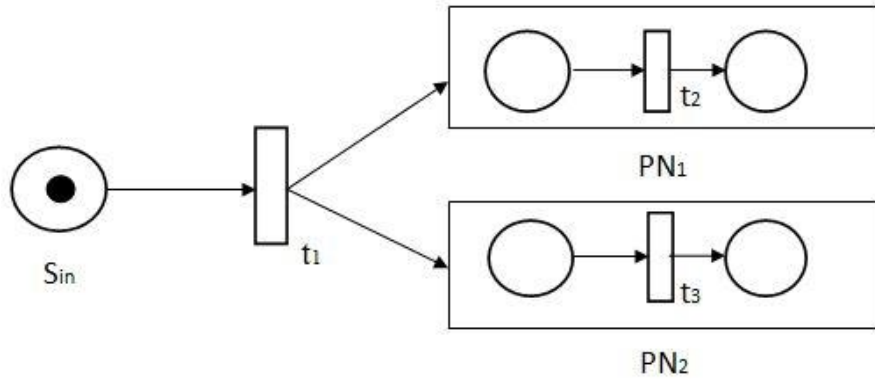


Figure 2.4: Split model

2.2.4 The Split-Join Process

The Petri Nets model for split-join composition of web services is a hierarchical Petri nets $SJPN = \langle S, PN, I, O \rangle$ as shown in Figure 2.4, where,

$$S = \{Sin, Sout\}$$

$$T = \{t1, t2, t3, t4\}$$

$$PN = \{PN1, PN2, PN3, \dots, PNk\},$$

Where PNi is a subset of i -th service.

$$I = I(sin, t1) \cup I(t1, PN_i) \cup I(PN_i, t2) \cup I(t1, sout),$$

$$i = 1, 2, 3 \dots k$$

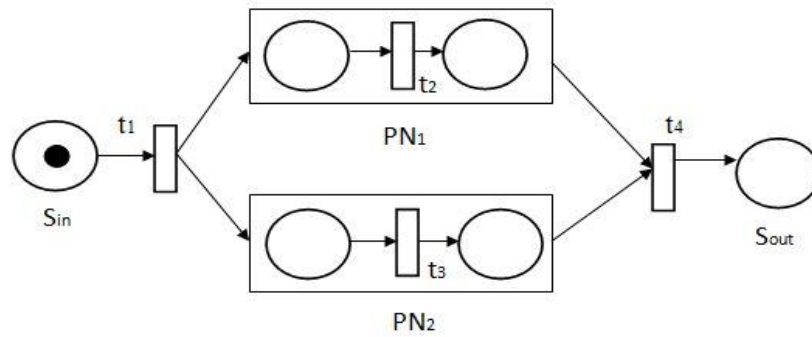


Figure 2.5: Split-Join model

2.3 Label Transition System (LTS)

A labelled transition system (LTS) comprises some number of states, with arcs between them labelled by activities of the system. Labelled transition systems are suitable for modelling discrete state systems that change through action of some kind. Certain states may be distinguished: a start state, perhaps one or more final states.

A labelled transition system is specified by:

- A set S of states;
- A set L of labels or actions;
- A set of transitions $T \subseteq S \times L \times S$.

Transitions are given as triples (start, label, end). The set of states may be finite or infinite; the set of labels is usually finite.

$$S = \{I, R, A\}$$

$$L = \{\text{alert, relax, on, off}\}$$

$$T = \{(I, \text{alert}, R), (R, \text{relax}, I), \\ (R, \text{on}, A), (A, \text{off}, I)\}$$

A *run* of a labelled transition system is a list of transitions that proceed from one state to another:

$$(s_1, l_1, s_2), (s_2, l_2, s_3) \dots (s_k, l_k, s_{k+1})$$

The *trace* of a run is the series of labels from these transitions:

$$l_1 l_2 l_3 \dots l_k$$

Both runs and traces may be finite or infinite. For any given system, we are generally interested in the set of all traces from a given initial state. For example, all these are traces of the LTS on the below:

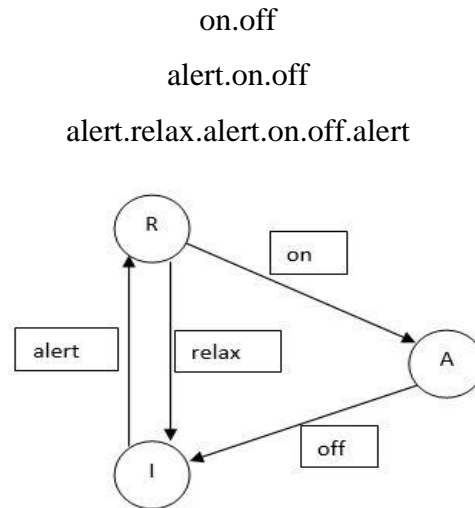


Figure: 2.6: Example of LTS

This is a finite LTS, but its complete set of traces is infinite. The set of traces for an LTS gives some information about the behavior of the system: but it is not enough to reconstruct the LTS itself. Labelled transitions systems, runs and traces are closely related to a range of similar notions in computer science:

- Finite-state machines / Automata
- Regular languages and regular Expressions
- Moore and Mealy machines

2.4 Finite State Process (FSP)

FSP stands for Finite State Process. It is the notation designed to be easily machine readable, and thus provides a preferred language to specify abstract workflows. FSP is a textual notation for

concisely describing and reasoning about concurrent programs. 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 converts an FSP into a state machine and provides a resulting LTS. Syntax and semantics of FSP owe much to both Hoare's CSP and Milner's CCS. FSP behavior specifications contain two sorts of process definitions: *primitive processes* and *composite processes*. Safety properties are specified using property automata.[2]

2.4.1 Primitive Processes [3]

Action Prefix "->":

($a \rightarrow P$) describes a process which engages in the action a and then behaves as described by P . More operationally, action prefix defines a transition between states. The following recursive definition describes the process *CLOCK* which repeatedly engages in the action *tick*.

$CLOCK = (tick \rightarrow CLOCK) .$

The LTS corresponding to the definition above is:



Figure 2.7: LTS of Action prefix

In FSP action labels must always start with a lowercase character (e.g. *tick*) and process names must begin with a capital letter (e.g. *CLOCK*). A primitive process definition is terminated by a full stop(.).

Choice "|":

($a \rightarrow P \mid b \rightarrow Q$) describes a process which initially engages in either of the actions a or b . After the first action has been performed, subsequent behavior is described by P if the first event

was a , or by Q if the first event was b . The LTS corresponding to this process has two possible transitions a and b out of the initial state. The example describes the behavior of a dispensing machine which dispenses coffee if the black button is pressed and tea if the white button is pressed.

`DRINKS = (black -> coffee -> DRINKS | white -> tea -> DRINKS).`

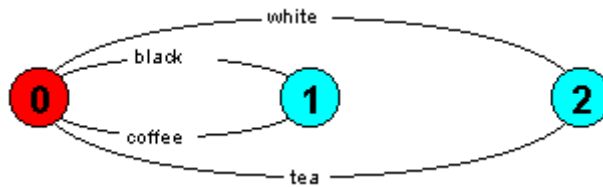


Figure 2.8: LTS of deterministic choice

In non-deterministic choice is simply expressed by having the same action leading to two different succeeding behaviors as shown in the example which describes tossing a coin.

`COIN = (toss -> heads -> COIN | toss -> tails -> COIN).`

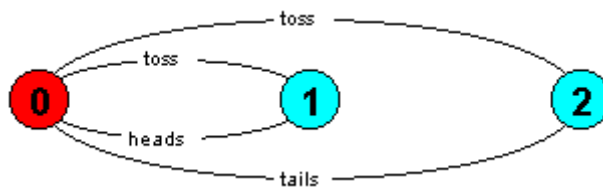


Figure 2.9: LTS of non-deterministic choice

STOP:

It is sometimes necessary to specify a primitive process which terminates. Consequently, a local process `STOP` is predefined which engages in no further actions. In *LTS* terms, `STOP` defines a state with no outgoing transitions. The example is a process which does some *init* action and then terminates.

```
STARTUP = (init -> STOP).
```

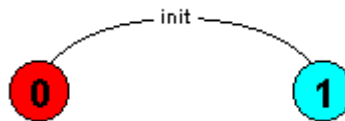


Figure 2.10: LTS of `STOP`

It should be noted that `STOP` means that the primitive process within which it is declared takes no further actions. It does not mean that the global system composed of potentially many primitive processes can take no further actions (which would be a deadlock).

Indexing:

Both local process names and action names may be indexed. This greatly increases the expressive power of FSP as demonstrated by the following examples.

In this example is a single cell buffer which stores integer values in the range 0 to 2.

```
BUFFER = EMPTY,
EMPTY  = (in[x:0..2] -> FULL[x]),
FULL[x:0..2] = (out[x] -> EMPTY).
```


Initially the buffer is empty until an *in* action to store a value occurs. This action may be to store the value 0, 1 or 2 as indicated by the range $[x:0..2]$. The process moves into a state in which it stores the appropriate value $FULL[0]$, $FULL[1]$ or $FULL[2]$. Outputting the value returns the buffer to the *EMPTY* state. The scope of the range variable x in each case is the local process in which it is declared. The LTS for the process is depicted below:

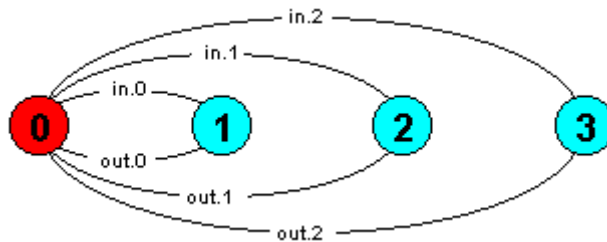


Figure 2.11: LTS of Indexing

The buffer process can be expressed more succinctly as below. BUF is exactly equivalent to BUF and generates an identical LTS.

$$BUF = (in[x:0..2] \rightarrow out[x] \rightarrow BUF) .$$

When modelling behavior of concurrent systems, it is frequently convenient to think of an action with a range as inputting a value in that range and an action indexed with a variable as outputting the value of that variable. Ranges may be explicitly declared as shown below:

$$\text{range } T = 0..2$$

$$BUF = (in[x:T] \rightarrow out[x] \rightarrow BUF) .$$

Which will produce the exactly same LTS like above.

Conditional:

A conditional takes the form: **if** *expr* **then** *local_process* **else** *local_process*. FSP supports only integer expressions. A non-zero expression value causes the conditional to behave as the local process of the **then** part; a zero value causes it to behave as the local process of the **else** part. The **else** part is optional, if omitted and *expr* evaluates to zero the conditional becomes the STOP process.

Example:

```
LEVEL = (read[x:0..3] ->
  if x>=2 then
    (high -> LEVEL)
  Else
    (low -> LEVEL)).
```

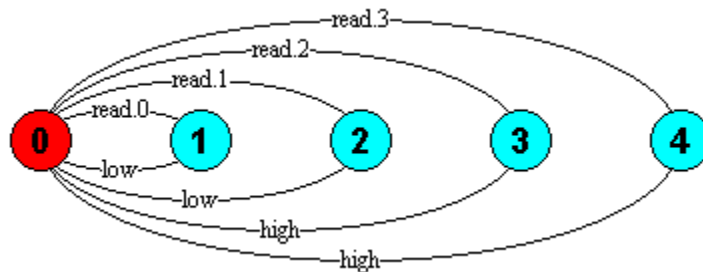


Figure2.12: LTS of Conditional

Guards:

A guarded transition takes the form (**when** *B* *a* -> *P*) which means that the action *a* is eligible when the guard *B* is true, otherwise *a* cannot be chosen for execution. The following example uses guards to define a bounded semaphore:

```
const Max = 4
```

```
rangeInt = 0..Max
```

```
BSEMA(Init=0) = BSEMA[Init],
BSEMA[v:Int] = (when (v<Max) up -> BSEMA[v+1]
|when (v>0) down ->BSEMA[v-1]
).
```

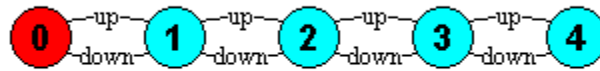


Figure 2.13: LTS of Guard

2.4.2 Composite Processes [3]

Parallel Composition "||":

$(P \parallel Q)$ expresses the parallel composition of the processes P and Q . It constructs an LTS which allows all the possible interleaving's of the actions of the two processes. Actions, which occur in the alphabets of both P and Q , constrain the interleaving since these actions must be carried out by both of the processes at the same time. These shared actions synchronize the execution of the two processes. If the processes contain no shared actions then the composite state machine will describe all interleaving's. In the following example, x is an action shared by the processes A and B .

```
A = (a -> x -> A) .
B = (b -> x -> B) .
||SYS = (A || B) .
```

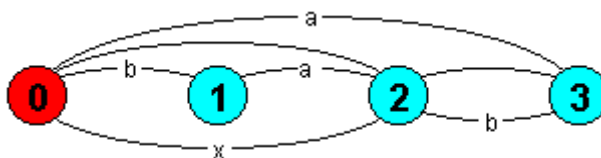


Figure 2.14: LTS of Parallel Composition

The diagram depicts the LTS for the composite process *SYS*. It can be easily seen that, in this simple example, the two possible execution traces are $\langle a, b, x \rangle$ and $\langle b, a, x \rangle$. That is the actions *a* and *b* can occur in any order.

Relabeling "/":

Relabeling functions are applied to processes and change the names of action labels. This is usually done to ensure that composed processes synchronize on the correct actions. A relabeling function can be applied to both primitive and composite processes. However, it is generally more used in composition. The general form of the relabeling function is $\{newlabel_1/oldlabel_1, \dots, newlabel_n/oldlabel_n\}$. The example shows the composition of two binary semaphore processes to give a semaphore which can be incremented twice (by *up*). The diagram shows how the *down* action of the first *SEMA* process is associated with the *up* action of the next semaphore by relabeling both to *mid*.

```
SEMA = (up -> down -> SEMA) .
||SEMA2 = (SEMA/{mid/down} || SEMA/{mid/up} ) .
```

The alphabet of *SEMA2* is $\{up, down, mid\}$ and its LTS is depicted below:

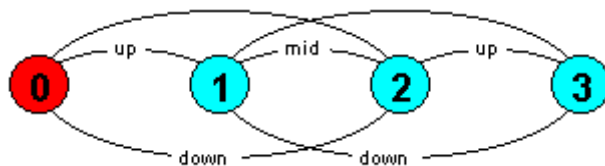


Figure 2.15: LTS of Relabeling

Chapter 3

Software Product Line Feature Analysis

Software Product Line

Software product line engineering is concerned with capturing the commonalities, universal and shared attributes of a set of software-intensive applications for a specific problem domain. It allows for the rapid development of variants of a domain specific application through various configurations of a common set of reusable assets often known as *core assets*, which support the management of commonality as well as variability. On the one hand, *commonality* is supported by providing domain analysts with both the ability and the required tools for capturing all of the shared conceptual information within the applications of a given domain. On the other hand, *variability* is addressed by allowing the domain analysts to include application-specific attributes and features within their unified model but at the same time, restrict their use; this way, commonality and variability are handled simultaneously. In the context of software product lines, *feature modeling* is one of the important techniques for modeling the attributes of a family of systems. This modeling language is important in that it provides means for capturing variability in software product lines.

Feature Model

Features are important distinguishing aspects, qualities, or characteristics of a family of systems. They are used for depicting the shared structure and behavior of a set of similar systems. To form a product family, all the various features of a set of similar/related systems are composed into a feature model. A feature model is a means for representing the possible configuration space of all the products of a system product family in terms of its features. For this reason, it is important

that feature models be able to capture variability and commonality between the features of the different applications available in a given domain. As we will see in the following paragraphs, feature models provide suitable means for modeling commonality, by allowing the domain modelers to form a common feature model representation for multiple applications, as well as variability by providing means to capture competing features of different applications under one unified umbrella. An example of capturing commonality is when a similar feature, which exists in multiple applications is represented as a unique feature in the overall domain representation, while an example of variability is when one notion is viewed differently by separate applications and is therefore modeled using competing features.

Feature models can be represented both formally and graphically; however, the graphical notation depicted through a tree structure is more favored due to its visual appeal and easier understanding. More specifically, graphical feature models are in the form of a tree whose root node represents a domain concept, e.g., a domain application, and the other nodes and leafs illustrate the features. In this context, a feature is a concept property related to a user-visible functional or nonfunctional requirement, e.g., domain application task, modeled in a way to capture commonalities or possibly differentiate among product family variants.

In a feature model, features are hierarchically organized and can typically be classified as:

- *Mandatory*, the feature must be included in the description of its parent feature;
- *Optional*, the feature may or may not be included in its parent description given the situation;
- *Alternative feature group*, one and only one of the features from the feature group can be included in the parent description;
- *Or feature group*, one or more features from the feature group can be included in the description of the parent feature.

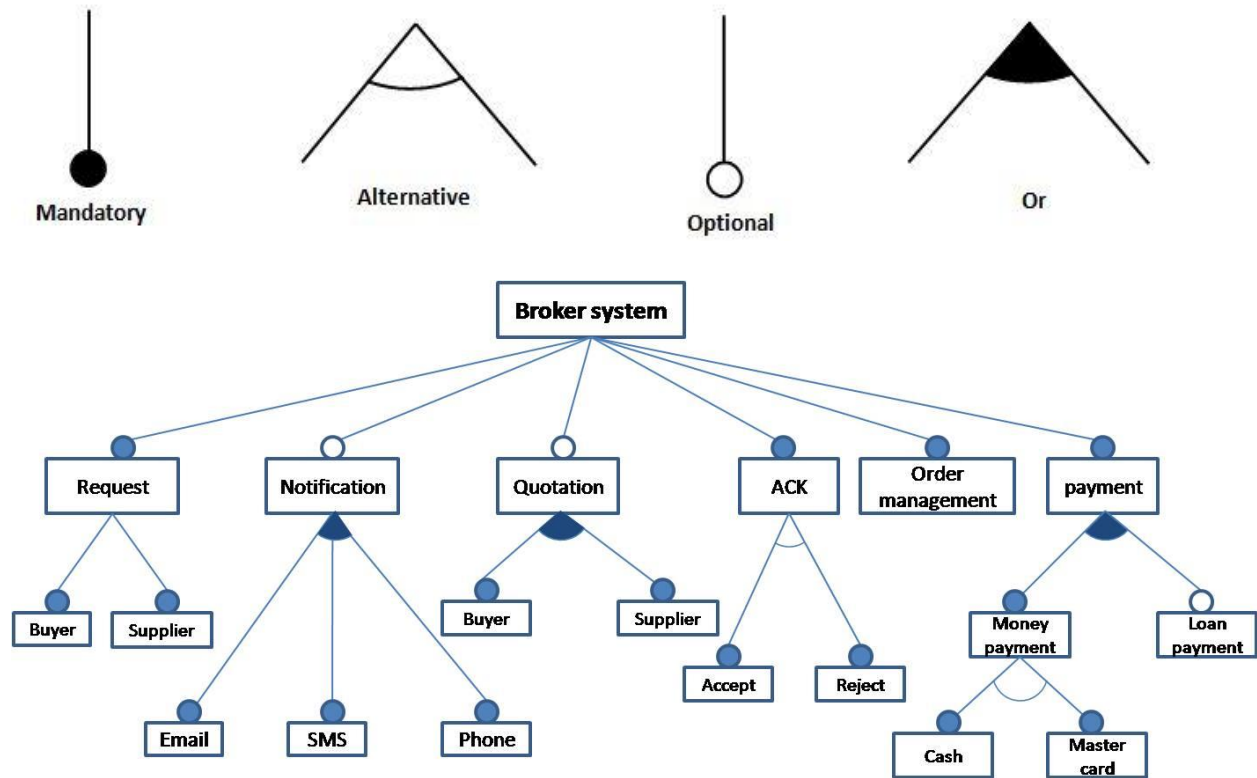


Figure 3.1: Feature tree of a broker system

In Broker System there are in total six features, four of them are mandatory; Request, ACK, Order management and Payment and two are optional features; Notification, Quotation. Request has two features in *or* operation, ACK has another two features and those are *alternative*. Either one can be selected. Also optional feature Notification has three features and those are mandatory but any (one or two or all) of them can be selected. In case of Payment, payment itself it is mandatory and it has two features one of them is mandatory and other one is optional also can both of them or only mandatory one can be selected.

Chapter 4

Service Orientation

4.1 Services and Comparison with features

A service is an abstract resource that represents a capability of performing tasks that represents a coherent functionality from the point of view of provider entities and requester entities. To be used, a service must be realized by a provider agent. This provider agent is the concrete piece of software (or hardware) that sends and receives messages, while the service is the resource characterized by the abstract set of functionality that is provided. Service identification is to select related resources. [5]

In service oriented applications, services are basic elements. So design and implementation of services is necessary steps in developing service oriented product line. In this way service composition is done to reuse existing services instead of implementing the new service. Service composition can be defined as the process of combining and linking existing services (atomic or composite) to create new working services. It constitutes an essential part of service provisioning, since it leads to novel service offering thus adding value that was not existent in the individual services. [5]

In service composition, the result of combining services is referred to as a composite service. When you use services together to achieve new functionality in a business process, the composition processes itself that dictates that the order and interactions between the lower-level services is exposed as this composite service. [5]

Feature modeling identifies a product line's features by identifying externally visible product characteristics in a product line and organizing them into a model. Product features are identified

and classified in terms of capability, domain technology, implementation technique, and operating-environment features. Capability features are user-visible characteristics that can be identified as distinct services, operations, and nonfunctional characteristics. Domain technology features represent ways of implementing services or operations. Implementation technique features are generic functions or techniques for implementing services, operations, and domain functions. Operating-environment features represent the environments where the applications are used. [4]

Service-based systems are distributed and composed of various services that can be discovered and replaced at runtime.

QoS has traditionally been associated with telephony and computer networking. Certain applications, such as voice over IP (VoIP), might require QoS because such applications have various requirements concerning network data flow (latency, jitter, number of dropped packets, and so on). Service-based systems consider qualities as constraints on a service's functionality, so mechanisms are necessary to guarantee the expected system quality at runtime. [6]

SPLE usually addresses quality issues statically during system design and implementation. Static quality management approaches rely on predicting system properties on the basis of its constituent components' properties. If, however, we simply statically predict resource use when developing a service-based system, the product might not have the resources to function correctly at the necessary quality level at runtime. To address this issue, we first statically define QoS in terms of features, with a maximum limit of available resources for each product. We then use this information when the product starts negotiating with service providers to select available services at runtime. [4]

To automate the advertisement, discovery, and negotiation of services, participants in a service-based system must share a common set of terms for describing service qualities and constraints. [8] A standard description method facilitates processes such as service advertisement, discovery, selection, composition, substitution, negotiation, and runtime service monitoring.[6] The most

prominent standard is the Web Services Description Language, which provides a service's location and a functional description of the service's input and output messages. [7]

Most SPLE approaches focus on configuring product line variations before deployment and don't consider dynamic-service composition. One way to address this problem is to distinguish between statically configured services (*static services*) and *dynamic services* during feature analysis. The configuration of static services can be tailored to each product. However, dynamic services might rely on third-party providers. So, a product must search for such a dynamic service when needed at runtime using the service-oriented architecture. Our proposed solution specifies static services, along with the tasks that constitute them, as workflows, and thus also specifies these services' pre and post conditions, invariants, and dynamic-service interfaces. Finally, by integrating and parameterizing dynamic services at runtime, our solution lets user's access static services with dynamic ones.[4]

SPLE promotes systematic reuse within an organization and usually doesn't consider external organizations when developing reusable assets. Moreover, relying on third-party providers and promoting the use of their services was out of scope for SPLE. The closest thing to third-party involvement that SPLE considers might be the use of commercial off-the-shelf (COTS) components. In SO, however, third-party involvement is one of the main drivers that make this approach attractive, and it leads to several initiatives, including service negotiations, service monitoring, and service reputation system. [4]

In a service-oriented marketplace, transactions often occur between parties that haven't previously interacted. Reputation systems are collaborative mechanisms that address trust issues between such parties, and they help distinguish between low- and high-quality service providers.[9] Including provider reputation in the service selection criteria benefits the quality assurance process.

Traditional SPLE approaches don't consider these three key aspects of dynamic-service provision, but SOPL methods should incorporate them. Therefore, we propose a QoS aware framework that provides automated runtime support for service discovery, negotiation,

monitoring, and service provider rating. QoS awareness lets consumers handle recovery from SLA violations, service failures, and runtime environment limitations by renegotiating and substituting problematic services.

4.2 Service Orientated Product line

The term Service-Oriented Product Line is used for service oriented applications that share common parts and vary in a regular and identifiable manner. In this context, high customization and systematic planned reuse are achieved through managed variability and the use of a two life-cycle model as in SPL engineering: core assets and product development.

In each service oriented application, there are two kinds of service compositions (static and dynamic), in this model both static and dynamic composition are considered in separate steps. Static composition implies that the compositions is performed at design or compile time. Dynamic service composition, on the other hand, composes an application autonomously when a user queries for an application at runtime. Therefore, dynamic composition involves adapting running applications by changing their functionalities and/or behavior via the addition or removal of service components at run time.

If there is a list of possible service candidates, we should select them from reusable service repository. The purpose of service selection is to select optimal web service for a particular task. As the selected service has to become an integral part of the reference architecture, domain design imposes architecture constraints to be considered during this selection, such as the architectural styles and patterns that the service must conform to, compatibility constraints, and constraints caused by the process structure of the reference architecture. At this time, if there are not any matched services and we select the static composition, we should decompose the main feature into sub features, chose the best available services that are matched with these sub features, linked together these atomic services and finally compiled and deployed the new (composite) service.

Two main approaches are currently investigated for static service composition. The first approach, referred to as web service *orchestration*, combines available services by adding a central coordinator (the orchestrator) that is responsible for invoking and combining the single

sub-activities. The second approach, referred to as web service *choreography*, does not assume the exploitation of a central coordinator but rather defines complex tasks via the definition of the conversation that should be undertaken by each participant.

Static composition is purely manual i.e. firstly, the user problem must be defined and then a manual selection of services according to desired outputs is performed. There are many potential problems, exceptions, and errors that may occur during this process. The challenge lies in dealing with these unexpected issues in the limited time frame that is permitted for a particular composition. Also, it is not possible to precisely predict or test at design time what the exact environmental circumstances of operation will be at composition time and whether the process will be successful. While steps are taken to decrease the chance of a failed composition, it cannot always be avoided.

Dynamic service composition is the process of creating new services at runtime from a set of service components. This process includes activities that must take place before the actual composition such as locating and selecting service components that will take part in the composition, and activities that must take place after the composition such as registering the new service with a service registry.

A very important aspect of dynamic service composition is that the new composite service need not be envisioned at design time. This feature, known as unanticipated dynamic composition, provides considerable flexibility for modifying and extending the operation of software systems during runtime. However, it also introduces a number of complications and problems for designing and operating software systems that support dynamic service composition. In this paper, we will describe our experiences with dynamic service composition and discuss how it can be used to improve the agility, flexibility, and availability of business software systems, particularly for e- and m commerce systems.

In dynamic composition, automated tools are used to analyze a user problem, select and assemble web service interfaces so that their composition will solve the user problem. Furthermore, even if the dynamic composition process seems successful, there is the potential for

unexpected feature interactions that cannot be easily and rapidly discovered and recovered from. A feature interaction is the way a service component (i.e., a feature) modifies or affects at runtime the behavior of other service components in a particular composition. The problem is similar to a program that compiles without errors but still fails to execute properly.

Compilation is only one part of the successful execution of a program just as the composition process will not guarantee the composite service will function correctly. When unexpected feature interactions arise despite all measures taken to avoid them, it might be almost impossible for the composition infrastructure to correct the situation. Human (i.e., user) input is needed to determine if the side effects are neutral or service affecting. If the feature interactions cause the composite service to function incorrectly or behave erratically, the composite service can be terminated and never reassembled. However, in many situations it may be appropriate to simply ignore those feature interactions that do not seriously affect the operation of the composite service. There is also a lack of support for dynamic composition techniques in programming languages and other development tools. The fundamental challenge in composing services at runtime is the design and implementation of an infrastructure that will support the process. Locating components at runtime requires a component library or code repository that is integrated with the software infrastructure that is actually performing the composition. The infrastructure should also support mechanisms to recover (e.g., rollback) from an unsuccessful composition and to discover and, if possible, recover from unexpected feature interactions. All these and other issues make the dynamic composition process inherently complex. Consequently, cost-benefit analysis must be taken into consideration before applying dynamic service composition techniques to a particular circumstance.

Chapter 5

Web Service Choreography

5.1 Introduction

For explaining choreography of web service we choose car broker system. A web service Broker provides online support to customers to negotiate car purchases and arranges loans for these. A buyer provides a need for a car model. The broker first uses its business partner Supplier to find the best possible quote for the requested model and then uses another business partner Lender to arrange a loan for the buyer for the selected quote. The buyer is also notified about the quote and the necessary arrangements for the loan. Both Lender and the Buyer can cause an interrupt to be invoked. A loan can be refused due to a failure in the loan assessment and a customer can reject the loan and quoted offer. In both cases, there is a need to run the compensation, where the car might have already been ordered, or the loan has already been offered.

5.2 Buyer

A buyer, is the one who want to buy a product from online. For that buyer request for product to Broker and get list of desired products. Then choose and order products.



Figure 5.1: Architectural view of Buyer

5.3 Lender Web Service

A loan service is a frequent example for business processes. Lender is a lender that offers a loan to customer, who submits a proposal containing name, address and loan amount. If the amount is 10000 or more, Lender asks its business partner to perform a full assessment. Business partner is an approver that thoroughly evaluates a loan proposal. The loan rate it determines is returned by Lender to its customer.

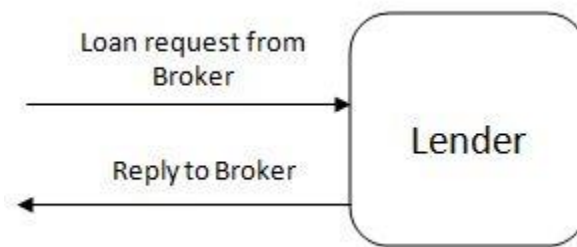


Figure 5.2: Architectural view of Lender web service

5.4 Supplier Web Service

A supplier that offers Buyer a good deal on product orders. A customer provides a need containing name, address and product details to Broker. The request for a quotation is passed through Broker to Supplier. Supplier and Buyer doesn't have any direct contact. Buyer orders for the product to Broker and Broker contact with Supplier and makes all arrangement.

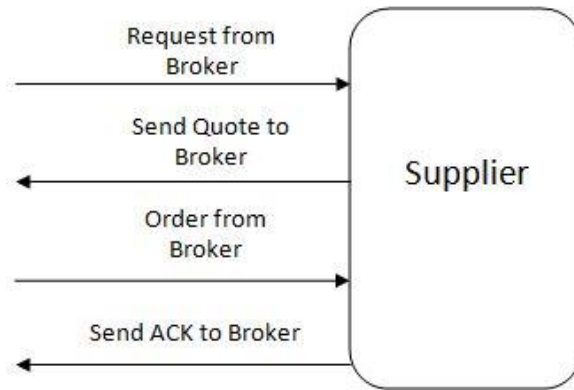


Figure 5.3: The Architectural view of supplier web service

5.5 Broker Web Service

A broker that provides an online service to negotiate product purchases and loans for these. A customer provides a need with name, address and product details. Broker first uses its business partner supplier to order the product on the best terms. If the product is unavailable broker informs its customer. Otherwise, Broker asks its business partner Lender to arrange a loan for the product price. If a loan can be provided, the customer receives a schedule containing the product name, price, and delivery period and loan rate. If a loan is refused, a loan refusal fault will occur. Since the product has already been ordered, compensation requires the order to be cancelled. The refusal is then returned to the customer.

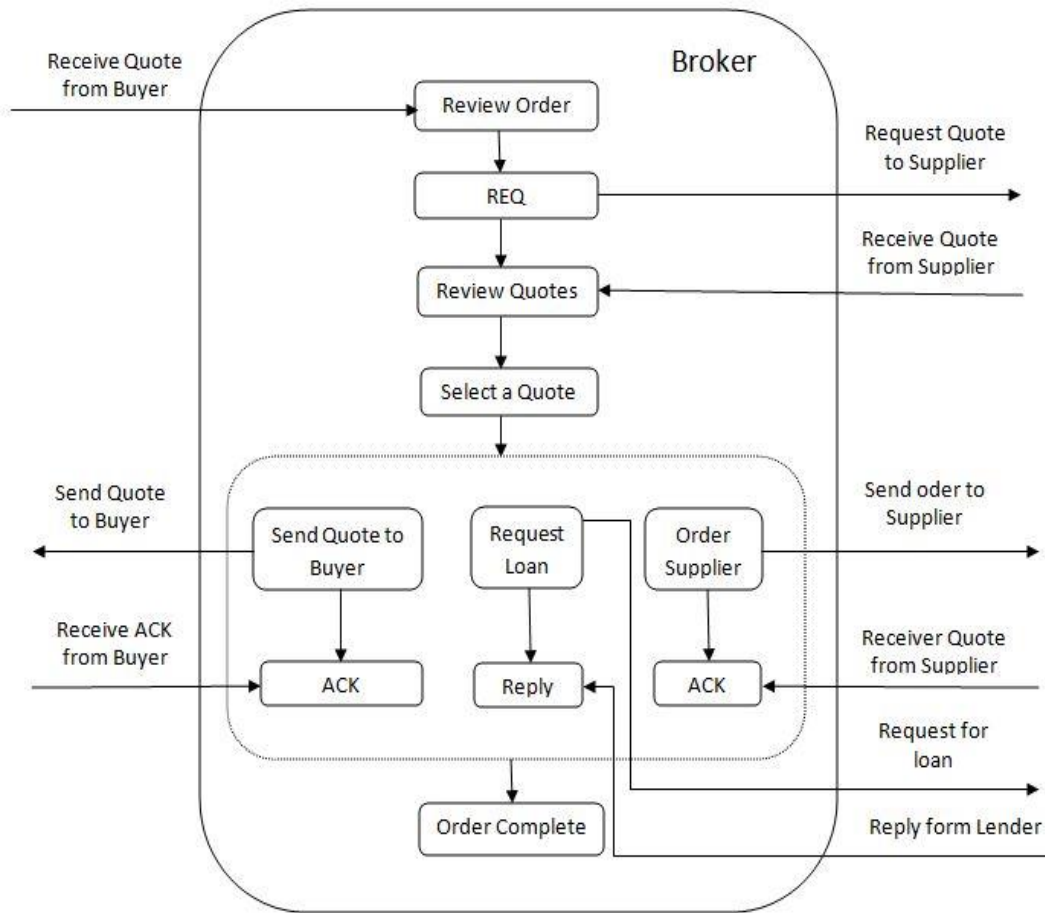


Figure 5.4: The Architectural view of Broker web service

Figure 5.5 shows the full architectural view of Car Broker System, where Buyer, Supplier, Lender and Broker Web service works together.

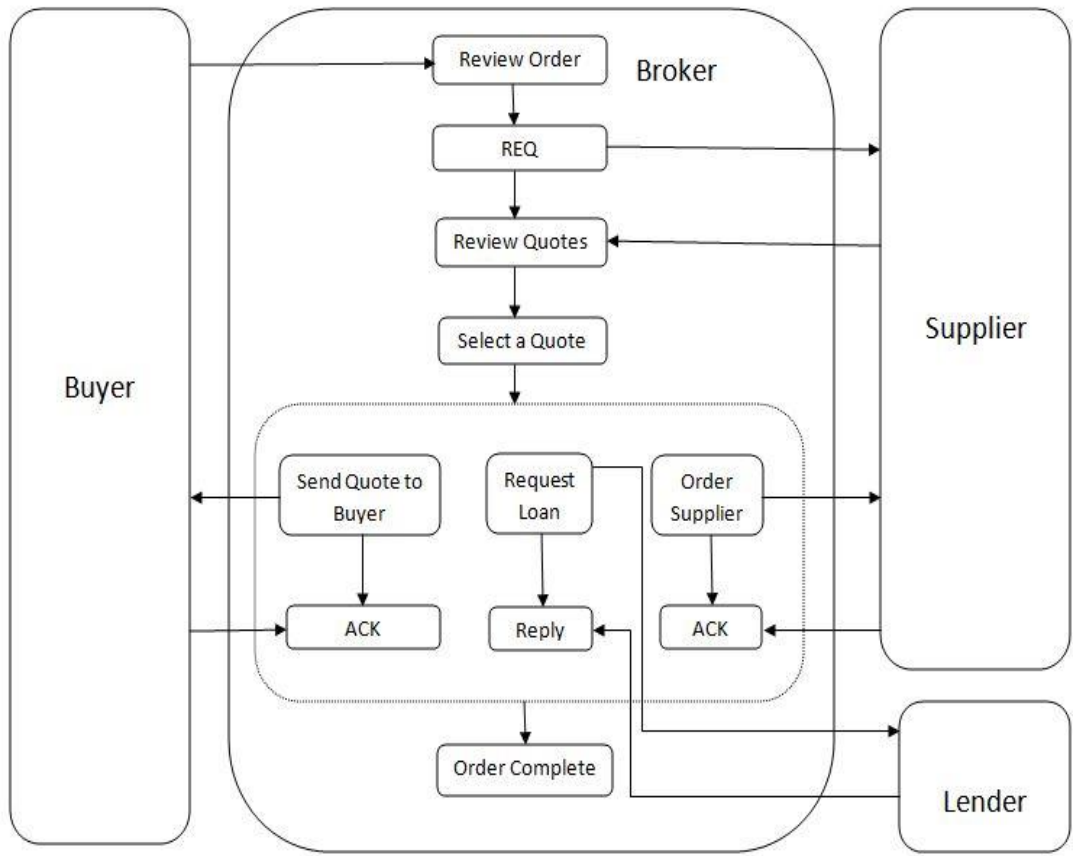


Figure 5.5: Architectural view of Broker System

Chapter 6

Choreography Analysis

6.1 Petri Net Representation

A broker system web service deals any purchases for its buyers and also arranges loans for buyer. The broker uses two separate web services: a Supplier to find a suitable quote for the requested and a Lender to arrange loans. Each web service can operate separately and can be used in other web services. In this case study, our focus is on how the processes communicate with each other. For brevity, describing the broker system example

We model a web service named Broker system. It provides online support to customers to negotiate product purchases and arranges loans for the product. At first buyer provides a need for a product model. The broker first uses its business partner Supplier to find the best possible quote for the requested model and then uses another business partner Loan Star to arrange a loan for the buyer for the selected quote. The buyer is also notified about the quote and the necessary arrangements for the loan. The process should be completed by talking the confirmation from buyer. We model this web service using Petri Nets.

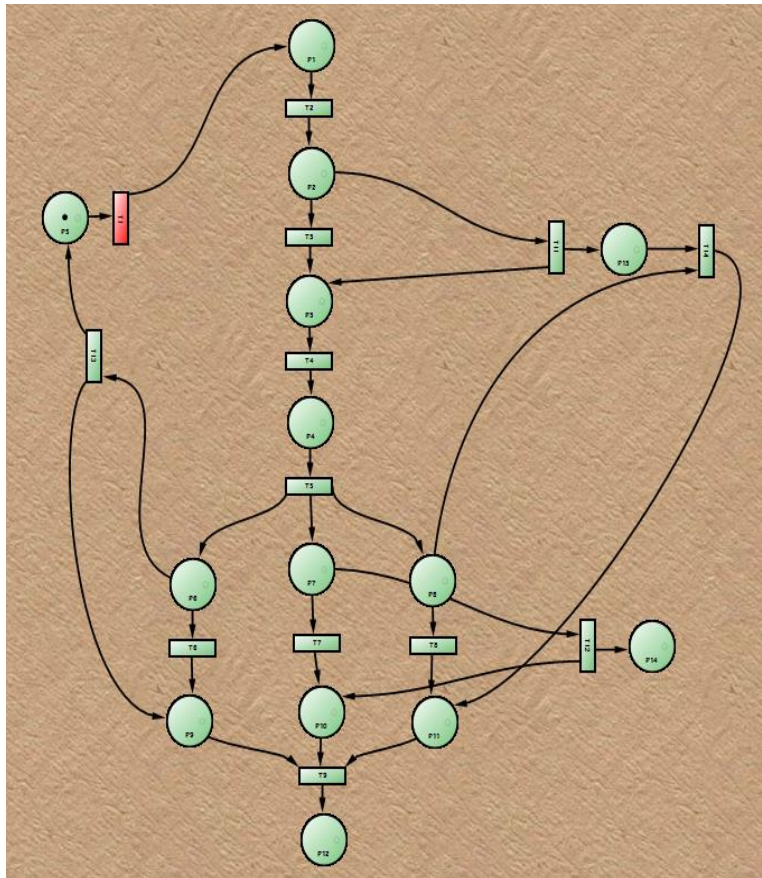


Figure 6.1: The Petri Nets Representation of Broker system web service

At first the Broker received a token from Buyer (the token contain information about product). After receiving the token Broker add more parameter with the token and send it to Supplier. Supplier also adds some parameter with the token and return back it to Broker. After that, the Broker simultaneously sends the token to Lender, Buyer and Supplier. Lender receives the token as request for Loan, Buyer receives it as a notification (acknowledgement) for confirmation and Supplier receives it as a delivery order (acknowledgement). The process is terminated when all of those tokens are return back to Broker.

6.2 FSP Representation

Through the process we have seen the Petri Net representation now let's see the same example in FSP (Finite state process). In FSP we divided our whole system into different pieces (i.e. Buyer, Broker, Supplier and Loan) and represent them.

6.2.1 Model

When we divided the whole system into pieces and represent them into FSP then we got.

Buyer:

```
BUYER = (req_to_broker->response_from_broker->ack_to_broker->BUYER) .
```

Broker:

```
BROKER_SEQ = BROKER_FROM_B,
```

```
BROKER_FROM_B = (rec_order->BROKER_REQ),
```

```
BROKER_REQ = (req_sup->rec_quote->select_quote->END) .
```

```
BROKER_TO_B = (select_quote->send_quote_B->ack_from_B->BROKER_TO_B) .
```

```
BROKER_TO_LOAN = (select_quote->req_loan->reply->BROKER_TO_LOAN) .
```

```
BROKER_SUPPLIER = (select_quote->order_sup->ack_sup->BROKER_SUPPLIER) .
```

```
|| BROKER =  
(BROKER_SEQ || BROKER_TO_B || BROKER_TO_LOAN || BROKER_SUPPLIER) .
```

Supplier:

```
SUPPLIER = (req_from_broker->response_to_broker->get_order_from_broker->ack_to_broker->SUPPLIER).
```

Loan:

```
LOAN = (req_loan->reply->LOAN).
```

Now let's see the whole system together in FSP. Here is the code.

Broker system:

```
const N = 2
range I = 1..N
BUYER = (req_order->resp_from_broker[I]->send_ack_to_broker->BUYER).
BROKER = (rec_oder->req_to_supplier->recieve_quote_from_supplier[I]->resp_to_buyer[I]->rec_ack_from_buyer->req_loan->reply->BROKER).
LOAN = (req_loan->reply->LOAN).
SUPPLIER = (req_from_broker->prep->send_quote_to_broker[I]->SUPPLIER).
||BROKER_SYS = (BUYER || BROKER || SUPPLIER || LOAN)
  /{order/req_order,order/rec_oder,
  req_to_supplier/req_from_broker,
  recieve_quote_from_supplier/send_quote_to_broker,
  ack/send_ack_to_broker,ack/rec_ack_from_buyer,
  resp/resp_from_broker,
  resp/resp_to_buyer
}.

```

6.2.2 LTS Analysis

The common approach to analysis is described and specific properties analyzed given different sets of models, for service choreography, compatibility and implementation analysis, and properties for analysis. For verification we analyze compositions of the models from the earlier section.

Buyer:

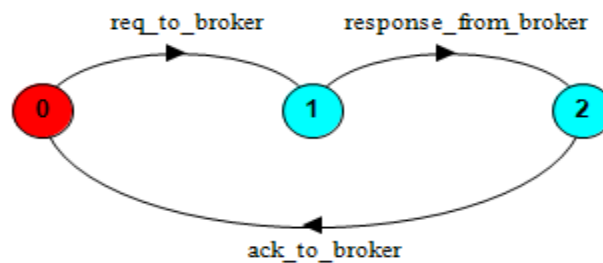


Figure 6.2: LTS of Buyer

Transitions of Buyer:

Process:

BUYER

States:

3

Transitions:

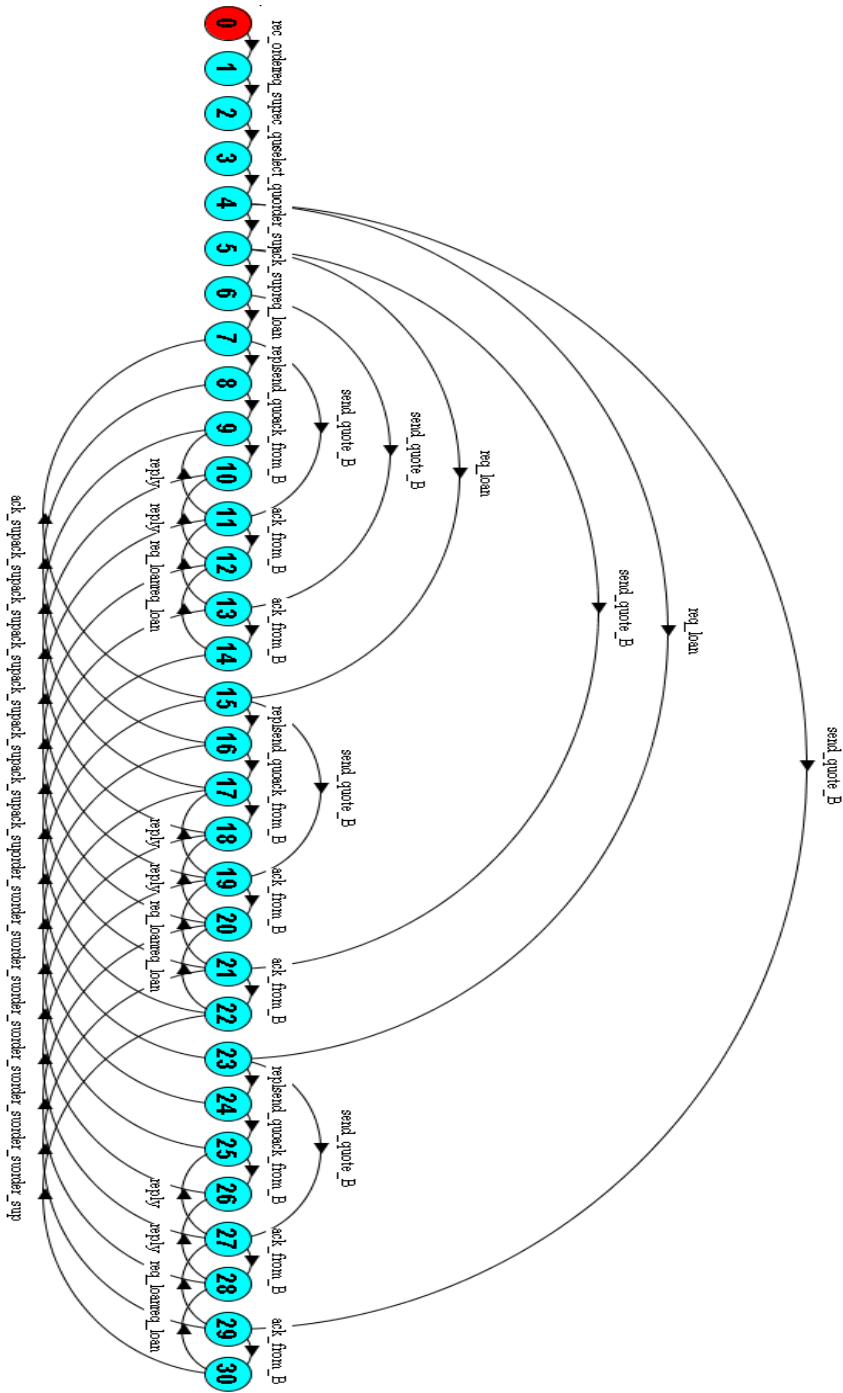
BUYER = Q0,

Q0 = (req_to_broker -> Q1),

Q1 = (response_from_broker -> Q2),

Q2 = (ack_to_broker -> Q0).

Broker:



Q13 = (req_loan -> Q11
 |ack_from_B -> Q14),
 Q14 = (req_loan -> Q12),
 Q15 = (ack_sup -> Q7
 |reply -> Q16


```

    |send_quote_B -> Q19),
Q16 = (ack_sup -> Q8
    |send_quote_B -> Q17),
Q17 = (ack_sup -> Q9
    |ack_from_B -> Q18),
Q18 = (ack_sup -> Q10),
Q19 = (ack_sup -> Q11
    |reply -> Q17
    |ack_from_B -> Q20),
Q20 = (ack_sup -> Q12
    |reply -> Q18),
Q21 = (ack_sup -> Q13
    |req_loan -> Q19
    |ack_from_B -> Q22),
Q22 = (ack_sup -> Q14
    |req_loan -> Q20),
Q23 = (order_sup -> Q15
    |reply -> Q24
    |send_quote_B -> Q27),
Q24 = (order_sup -> Q16
    |send_quote_B -> Q25),
Q25 = (order_sup -> Q17
    |ack_from_B -> Q26),
Q26 = (order_sup -> Q18),
Q27 = (order_sup -> Q19
    |reply -> Q25
    |ack_from_B -> Q28),
Q28 = (order_sup -> Q20
    |reply -> Q26),
Q29 = (order_sup -> Q21
    |req_loan -> Q27
    |ack_from_B -> Q30),
Q30 = (order_sup -> Q22
    |req_loan -> Q28).

```

Trace to DEADLOCK:

```

rec_order
req_sup
rec_quote
select_quote

```

```
send_quote_B
ack_from_B
req_loan
reply
order_sup
ack_sup
```

So in the above LTS of Broker there is a possibility of Deadlock, which we can remove by replace END with BROKER_SEQ of this portion.

```
BROKER_SEQ = BROKER_FROM_B,
BROKER_FROM_B = (rec_order->BROKER_REQ),
BROKER_REQ = (req_sup->rec_quote->select_quote->BROKER_REQ) .
```

After this there will be no deadlock but problem is LTS Draw become overflow 31 transitions become 108 transitions. Transitions are given in Appendix A.

Supplier:

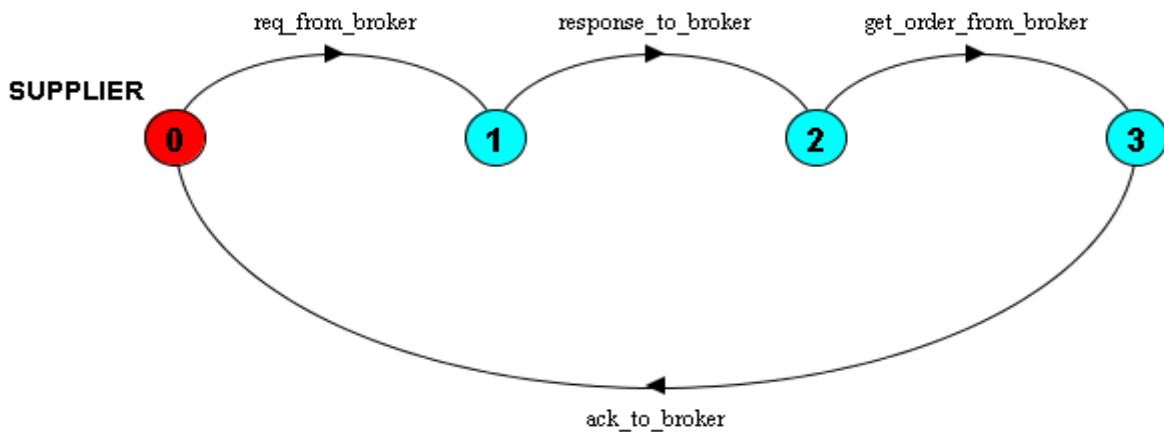


Figure 6.4: LTS of SUPPLIER

Transitions of Supplier:

Process:

SUPPLIER

States:

4

Transitions:

SUPPLIER = Q0,

Q0 = (req_from_broker -> Q1),

Q1 = (response_to_broker -> Q2),

Q2 = (get_order_from_broker -> Q3),

Q3 = (ack_to_broker -> Q0).

Loan:

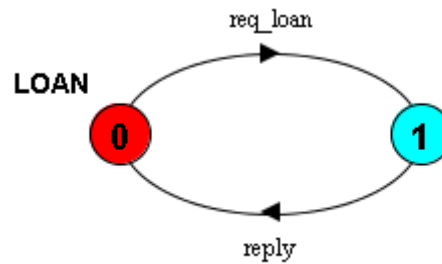


Figure 6.5: LTS of Loan

Transitions of loan:

Process:

LOAN

States:

2

Transitions:

LOAN = Q0,
Q0 = (req_loan -> Q1),
Q1 = (reply -> Q0).

Broker System:

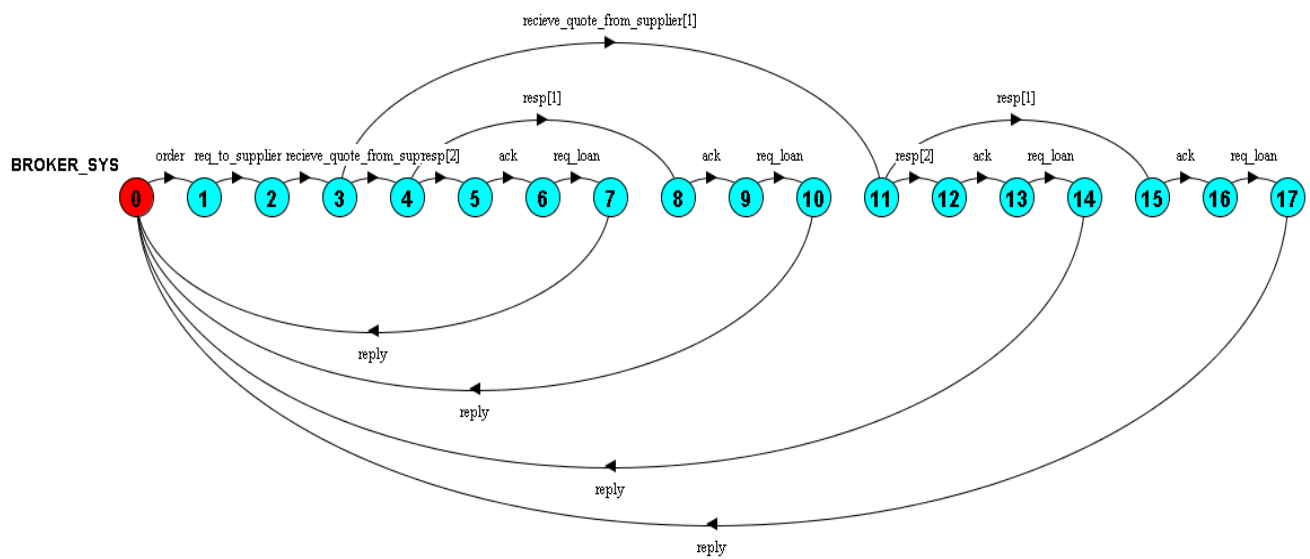


Figure 6.6: LTS of Broker System

Transitions of Broker System:

Process:

BROKER_SYS

States:

18

Transitions:

```
BROKER_SYS = Q0,  
Q0  = (order -> Q1),  
Q1  = (req_to_supplier -> Q2),  
Q2  = (prep -> Q3),  
Q3  = (recieve_quote_from_supplier[2] -> Q4  
      |recieve_quote_from_supplier[1] -> Q11),  
Q4  = (resp[2] -> Q5  
      |resp[1] -> Q8),  
Q5  = (ack -> Q6),  
Q6  = (req_loan -> Q7),  
Q7  = (reply -> Q0),  
Q8  = (ack -> Q9),  
Q9  = (req_loan -> Q10),  
Q10 = (reply -> Q0),  
Q11 = (resp[2] -> Q12  
      |resp[1] -> Q15),  
Q12 = (ack -> Q13),  
Q13 = (req_loan -> Q14),  
Q14 = (reply -> Q0),  
Q15 = (ack -> Q16),  
Q16 = (req_loan -> Q17),  
Q17 = (reply -> Q0).
```

Chapter 7

Conclusion

7.1 Summary

We have shown the composition of web service choreography in service oriented product line. We modeled the Car Broker Web Service in Petri Net to verify the composition and represents it in Finite State Process to analysis the composition.

7.2 Future work

Our future plan is to verify service orchestration by using a suitable tool by following similar fashion as in this project. We are also interested to check other properties such as livelock and other compositional errors in service composition

Appendix A

A.1 Broker System

Process:

BROKER

States:

108

Transitions:

```
BROKER = Q0,
Q0    = (rec_order -> Q1),
Q1    = (req_sup -> Q2),
Q2    = (rec_quote -> Q3),
Q3    = (select_quote -> Q4),
Q4    = (order_sup -> Q5
        |req_loan -> Q73
        |send_quote_B -> Q97
        |rec_order -> Q105),
Q5    = (ack_sup -> Q6
        |req_loan -> Q38
        |send_quote_B -> Q62
        |rec_order -> Q70),
Q6    = (req_loan -> Q7
        |send_quote_B -> Q27
        |rec_order -> Q35),
Q7    = (reply -> Q8
        |send_quote_B -> Q16
        |rec_order -> Q24),
Q8    = (send_quote_B -> Q9
        |rec_order -> Q13),
Q9    = (ack_from_B -> Q0
        |rec_order -> Q10),
Q10   = (ack_from_B -> Q1
        |req_sup -> Q11),
Q11   = (ack_from_B -> Q2
        |rec_quote -> Q12),
```

```
Q12 = (ack_from_B -> Q3),
Q13 = (send_quote_B -> Q10
      |req_sup -> Q14),
Q14 = (send_quote_B -> Q11
      |rec_quote -> Q15),
Q15 = (send_quote_B -> Q12),
Q16 = (reply -> Q9
      |ack_from_B -> Q17
      |rec_order -> Q21),
Q17 = (reply -> Q0
      |rec_order -> Q18),
Q18 = (reply -> Q1
      |req_sup -> Q19),
Q19 = (reply -> Q2
      |rec_quote -> Q20),
Q20 = (reply -> Q3),
Q21 = (reply -> Q10
      |ack_from_B -> Q18
      |req_sup -> Q22),
Q22 = (reply -> Q11
      |ack_from_B -> Q19
      |rec_quote -> Q23),
Q23 = (reply -> Q12
      |ack_from_B -> Q20),
Q24 = (reply -> Q13
      |send_quote_B -> Q21
      |req_sup -> Q25),
Q25 = (reply -> Q14
      |send_quote_B -> Q22
      |rec_quote -> Q26),
Q26 = (reply -> Q15
      |send_quote_B -> Q23),
Q27 = (req_loan -> Q16
      |ack_from_B -> Q28
      |rec_order -> Q32),
Q28 = (req_loan -> Q17
      |rec_order -> Q29),
Q29 = (req_loan -> Q18
      |req_sup -> Q30),
Q30 = (req_loan -> Q19
      |rec_quote -> Q31),
```



```

Q31 = (req_loan -> Q20),
Q32 = (req_loan -> Q21
      |ack_from_B -> Q29
      |req_sup -> Q33),
Q33 = (req_loan -> Q22
      |ack_from_B -> Q30
      |rec_quote -> Q34),
Q34 = (req_loan -> Q23
      |ack_from_B -> Q31),
Q35 = (req_loan -> Q24
      |send_quote_B -> Q32
      |req_sup -> Q36),
Q36 = (req_loan -> Q25
      |send_quote_B -> Q33
      |rec_quote -> Q37),
Q37 = (req_loan -> Q26
      |send_quote_B -> Q34),
Q38 = (ack_sup -> Q7
      |reply -> Q39
      |send_quote_B -> Q51
      |rec_order -> Q59),
Q39 = (ack_sup -> Q8
      |send_quote_B -> Q40
      |rec_order -> Q48),
Q40 = (ack_sup -> Q9
      |ack_from_B -> Q41
      |rec_order -> Q45),
Q41 = (ack_sup -> Q0
      |rec_order -> Q42),
Q42 = (ack_sup -> Q1
      |req_sup -> Q43),
Q43 = (ack_sup -> Q2
      |rec_quote -> Q44),
Q44 = (ack_sup -> Q3),
Q45 = (ack_sup -> Q10
      |ack_from_B -> Q42
      |req_sup -> Q46),
Q46 = (ack_sup -> Q11
      |ack_from_B -> Q43
      |rec_quote -> Q47),
Q47 = (ack_sup -> Q12

```

```

    |ack_from_B -> Q44),
Q48 = (ack_sup -> Q13
    |send_quote_B -> Q45
    |req_sup -> Q49),
Q49 = (ack_sup -> Q14
    |send_quote_B -> Q46
    |rec_quote -> Q50),
Q50 = (ack_sup -> Q15
    |send_quote_B -> Q47),
Q51 = (ack_sup -> Q16
    |reply -> Q40
    |ack_from_B -> Q52
    |rec_order -> Q56),
Q52 = (ack_sup -> Q17
    |reply -> Q41
    |rec_order -> Q53),
Q53 = (ack_sup -> Q18
    |reply -> Q42
    |req_sup -> Q54),
Q54 = (ack_sup -> Q19
    |reply -> Q43
    |rec_quote -> Q55),
Q55 = (ack_sup -> Q20
    |reply -> Q44),
Q56 = (ack_sup -> Q21
    |reply -> Q45
    |ack_from_B -> Q53
    |req_sup -> Q57),
Q57 = (ack_sup -> Q22
    |reply -> Q46
    |ack_from_B -> Q54
    |rec_quote -> Q58),
Q58 = (ack_sup -> Q23
    |reply -> Q47
    |ack_from_B -> Q55),
Q59 = (ack_sup -> Q24
    |reply -> Q48
    |send_quote_B -> Q56
    |req_sup -> Q60),
Q60 = (ack_sup -> Q25
    |reply -> Q49

```

```

    |send_quote_B -> Q57
    |rec_quote -> Q61),
Q61 = (ack_sup -> Q26
    |reply -> Q50
    |send_quote_B -> Q58),
Q62 = (ack_sup -> Q27
    |req_loan -> Q51
    |ack_from_B -> Q63
    |rec_order -> Q67),
Q63 = (ack_sup -> Q28
    |req_loan -> Q52
    |rec_order -> Q64),
Q64 = (ack_sup -> Q29
    |req_loan -> Q53
    |req_sup -> Q65),
Q65 = (ack_sup -> Q30
    |req_loan -> Q54
    |rec_quote -> Q66),
Q66 = (ack_sup -> Q31
    |req_loan -> Q55),
Q67 = (ack_sup -> Q32
    |req_loan -> Q56
    |ack_from_B -> Q64
    |req_sup -> Q68),
Q68 = (ack_sup -> Q33
    |req_loan -> Q57
    |ack_from_B -> Q65
    |rec_quote -> Q69),
Q69 = (ack_sup -> Q34
    |req_loan -> Q58
    |ack_from_B -> Q66),
Q70 = (ack_sup -> Q35
    |req_loan -> Q59
    |send_quote_B -> Q67
    |req_sup -> Q71),
Q71 = (ack_sup -> Q36
    |req_loan -> Q60
    |send_quote_B -> Q68
    |rec_quote -> Q72),
Q72 = (ack_sup -> Q37
    |req_loan -> Q61

```

```

    |send_quote_B -> Q69),
Q73 = (order_sup -> Q38
    |reply -> Q74
    |send_quote_B -> Q86
    |rec_order -> Q94),
Q74 = (order_sup -> Q39
    |send_quote_B -> Q75
    |rec_order -> Q83),
Q75 = (order_sup -> Q40
    |ack_from_B -> Q76
    |rec_order -> Q80),
Q76 = (order_sup -> Q41
    |rec_order -> Q77),
Q77 = (order_sup -> Q42
    |req_sup -> Q78),
Q78 = (order_sup -> Q43
    |rec_quote -> Q79),
Q79 = (order_sup -> Q44),
Q80 = (order_sup -> Q45
    |ack_from_B -> Q77
    |req_sup -> Q81),
Q81 = (order_sup -> Q46
    |ack_from_B -> Q78
    |rec_quote -> Q82),
Q82 = (order_sup -> Q47
    |ack_from_B -> Q79),
Q83 = (order_sup -> Q48
    |send_quote_B -> Q80
    |req_sup -> Q84),
Q84 = (order_sup -> Q49
    |send_quote_B -> Q81
    |rec_quote -> Q85),
Q85 = (order_sup -> Q50
    |send_quote_B -> Q82),
Q86 = (order_sup -> Q51
    |reply -> Q75
    |ack_from_B -> Q87
    |rec_order -> Q91),
Q87 = (order_sup -> Q52
    |reply -> Q76
    |rec_order -> Q88),

```

```

Q88 = (order_sup -> Q53
      |reply -> Q77
      |req_sup -> Q89),
Q89 = (order_sup -> Q54
      |reply -> Q78
      |rec_quote -> Q90),
Q90 = (order_sup -> Q55
      |reply -> Q79),
Q91 = (order_sup -> Q56
      |reply -> Q80
      |ack_from_B -> Q88
      |req_sup -> Q92),
Q92 = (order_sup -> Q57
      |reply -> Q81
      |ack_from_B -> Q89
      |rec_quote -> Q93),
Q93 = (order_sup -> Q58
      |reply -> Q82
      |ack_from_B -> Q90),
Q94 = (order_sup -> Q59
      |reply -> Q83
      |send_quote_B -> Q91
      |req_sup -> Q95),
Q95 = (order_sup -> Q60
      |reply -> Q84
      |send_quote_B -> Q92
      |rec_quote -> Q96),
Q96 = (order_sup -> Q61
      |reply -> Q85
      |send_quote_B -> Q93),
Q97 = (order_sup -> Q62
      |req_loan -> Q86
      |ack_from_B -> Q98
      |rec_order -> Q102),
Q98 = (order_sup -> Q63
      |req_loan -> Q87
      |rec_order -> Q99),
Q99 = (order_sup -> Q64
      |req_loan -> Q88
      |req_sup -> Q100),
Q100 = (order_sup -> Q65

```

```
    |req_loan -> Q89
    |rec_quote -> Q101),
Q101 = (order_sup -> Q66
    |req_loan -> Q90),
Q102 = (order_sup -> Q67
    |req_loan -> Q91
    |ack_from_B -> Q99
    |req_sup -> Q103),
Q103 = (order_sup -> Q68
    |req_loan -> Q92
    |ack_from_B -> Q100
    |rec_quote -> Q104),
Q104 = (order_sup -> Q69
    |req_loan -> Q93
    |ack_from_B -> Q101),
Q105 = (order_sup -> Q70
    |req_loan -> Q94
    |send_quote_B -> Q102
    |req_sup -> Q106),
Q106 = (order_sup -> Q71
    |req_loan -> Q95
    |send_quote_B -> Q103
    |rec_quote -> Q107),
Q107 = (order_sup -> Q72
    |req_loan -> Q96
    |send_quote_B -> Q104).
```

References

- [1] Mohammad Salah Uddin, “Web Service Composition: A Comparison of BPEL with A Process Algebra”, Lambert Academic Publishing, ISBN: 978-3-659-46748-6.
- [2] Howard Foster “A Rigorous Approach to Engineering Web Service Compositions”, January 2006.
- [3] <http://www.doc.ic.ac.uk/~jnm/LTSdocumentation/FSP-notation.html>
- [4] Jaejoon Lee, Kotonya, G “Combining Service Orientation with Product Line Engineering.” In *IEEE Software*, 27 (3): 35-41, 2010. DOI BibTeX
- [5] Fatemeh Mafi, Shariar Mafi, Mehran Mohsenzadeh, Service Composition in Service Oriented Product Line. (*IJCSE*) *International Journal on Computer Science and Engineering* Vol. 02, No. 09, 2010, 2859-2864
- [6] G. Kotonya, J. Lee, and D. Robinson, “A Consumer- Centered Approach for Service-Oriented Product Line Development,” *Proc. Working IEEE/IFIP Conf. Software Architecture (WICSA 09)*, IEEE Press, 2009, pp. 211–220.
- [7] E. Christensen et al., *Web Services Description Language (WSDL) 1.1*, World Wide Web Consortium (W3C) note, Mar. 2001, www.w3.org/TR/wsdl.
- [8] G. Dobson, R. Lock, and I. Sommerville, “QoS Ont: A QoS Ontology for Service-Centric Systems,” *Proc. 31st Euromicro Conf. Software Eng. and Advanced Applications*, IEEE CS Press, 2005, pp. 80–87
- [9] A. Josang, R. Ismail, and C. Boyd, “A Survey of Trust and Reputation Systems for Online Service Provision,” *Decision Support Systems*, vol. 43, no. 2, 2007, pp. 618–644.
- [10] Artemios Kontogogos and Paris Avgeriou, “An Overview of Software Engineering Approaches to Service Oriented Architectures in Various Fields”, 2009 18th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises.