Process Based Service Composition and Verification

By

Ashika Afreen

Nowshin Laila

&

Md Sathil Islam

Supervised By

Dr. Shamim H. Ripon

East West University

Department of Computer Science and Engineering

Spring 2017

Process Based Service Composition and Verification

Submitted By

Ashika Afreen

2012-3-60-036

Nowshin Laila

2013-1-60-050

&

Md Sathil Islam

2013-1-60-012

A project submitted in partial fulfillment for the degree of Bachelor of Science in Computer Science and Engineering

In the

Faculty of Science and Engineering

Department of Computer Science and Engineering

East West University

Spring 2017

Declaration

We, hereby certify that our thesis work solely to be our own scholarly work. To the best of our knowledge, it has not been collected from any source without the due acknowledgement and permission. It is being submitted in fulfilling the requirements for the degree of Bachelor of Science in Computer Science and Engineering. It has not been submitted, either in whole or in part for a degree or examination at this or any other university.

Ashika Afreen

(2012-3-60-036)

Nowshin Laila

(2013-1-60-050)

Md Sathil Islam

(2013-1-60-012)

Letter of Acceptance

The Project entitled **"Process Based Service Composition and Verification"** submitted by Ashika Afreen (2012-3-60-036), Nowshin Laila (2013-1-60-050) & Md Sathil Islam (2013-1-60-012) to the Department of Computer Science and Engineering, East West University, Dhaka, Bangladesh in the semester of Fall 2015 is approved satisfactory in partial fulfillment of requirements for award of the degree of Bachelor of Science in Computer Science and Engineering.

Dr. Shamim H. Ripon

SUPERVISOR

Department of Computer Science and Engineering

East West University

Dhaka-1212, Bangladesh

Abstract

A platform-independent software component available in the distributed environment of the Internet is titled as Web service. Many Business organizations are publishing their applications functionalities on the web. A web service has a limited functionality alone. So to support business to business interactions it is a crying need to aggregate web services and assembled them in a goal oriented interface. To provide atomicity to a transaction where multiple partners are involved handling faults are both difficult and critical. A possible solution of the problem would be that the system designer can provide a mechanism to compensate the actions that cannot be undone automatically. In this project we have composed a car broker system and implemented a compensation mechanism that will compensate all services from their point of cancelation. We have modeled the service choreography in FSP and used LTSA tool to animate the transitions. Every model should be verified before implementation, we tried to verify the system composition using property processes available in FSP.

Acknowledgement

It is with enormous appreciation that we acknowledge the contribution of our supervisor, **Dr. Shamim H. Ripon**, Associate Professor of Department of Computer Science and Engineering, East West University. Without his proper guidance, encouragement and support this thesis would have remained a dream. We consider it as an honor to work with him. We are also indebted to our parents, other professors of the department and friends for their support and encouragements. Finally, thanks to the Almighty, who gave us the patience to complete the task successfully.

Table of Contents

List of Figures

CHAPTER 1

Introduction

1.1 Introduction and Motivation

Business transactions need multiple partner involvement, coordination and interaction with each other. Many business companies or enterprises publish their applications functionalities on the web using a web service format. Web services are defined as self-contained, modular units of application logic, which provide business functionality to other applications through an Internet connection. Each service provider is a self-contained software system having its own threads of control.

In this technological era business applications like web services allows greater efficiency and availability for business. A web service alone has a limited functionality which may not be sufficient to respond to the user's request. Whereas a composition of several web services can achieve a specific goal. From a user perspective, the composition might be considered as a simple web service, even though it is composed of several web services. In an essence, the aggregation is a collaboration of many Web service providers.

Models are simplified representations of real-world entities. We model something to better understand it. We can use models to focus on interesting aspects, visualize potential outcomes and create mechanisms to test and verify an approach. We need model checking to verify correctness properties such as the absence of deadlocks and similar critical states that can cause the system to crash. Every model should be verified before implementation. There are various languages to model a system and verify it properly. BPEL, cCSP and FSP are most handful language to model a system with their notations. Among them FSP has the most expressive and powerful approach to visualize the system. To provide atomicity to a transaction handling faults where multiple partners are involved are both difficult and critical. A possible solution of the problem would be that the system designer can provide a mechanism to compensate the actions that cannot be undone automatically. In BPEL compensation is expressed in a XML notation, in cCSP it is expressed in a compensation pair but it can be expressed in FSP as a separate process and represented elaborately.

1.2 Objectives

The objectives of our project are as follows:

- Analyzing the web service composition in respect to the composition mechanism orchestration and choreography.
- Modeling a composition using Finite State Process (FSP) notations and Labeled Transition System Analyzer (LTSA) tool.
- Introducing a compensation mechanism as a fault handler that could handle all the failed transactions and could manage all the compensation processes of every component processes.
- Verifying the designed model as it is specified in the model specification. Ensuring that in a concurrent execution all synchronizing points executes properly and no deadlock and such critical states occur that violate the correctness properties.

1.3 Contribution

Our contributions in the project are as follows:

We have used Finite State Process (FSP) notations to describe the model and LTSA tool to generate the corresponding Labeled transition Diagrams. We select Online Marketplace Web Service as our model. We analyzed the model and identified various components of the web service as well as the composition among the services. Then implement the system according to their interactions.

We have implemented a compensation mechanism which will describe the compensation actions from the point of cancelation. Each process has its own compensation process and a main compensation process to control the whole mechanism. We added some safety properties to verify synchronizations among processes in a concurrent execution and checked the correctness properties such as the absence of deadlocks and similar critical states that can

1.4 Outline

Chapter 1: Firstly we represent about our motivation to work, Specify our objectives and then the contribution that we have made.

Chapter 2: Web service composition and two ways to compose the web services (Choreography and orchestration). Then a brief description is given about Finite State Process (FSP) which is used to specify our model and about LTSA tool to compile FSP notations. After that we discussed about Compensating CSP (cCSP).

Chapter 3: This chapter describes about car broker service composition including the contribution of each web services in the system and how the compensation process works.

Chapter 4: The coding representation of our service in FSP.

Chapter 5: Define some Safety properties in order to verify our web service composition and compensation by composing them with required safety properties.

Chapter 6: At last, in this chapter we summarized our work and give a definition about our future plan.

CHAPTER 2

Background

2.1 Web Service and Composition

Web services are distributed, independent processes which communicate with each other through the exchange of messages. The coordination between business processes is particularly crucial as it includes the logic that makes a set of different software components become a whole system. Web services provided by various organizations can be interconnected to implement business collaborations, leading to composite web services. Business collaborations require interactions driven by explicit process models. Web services are driven by the paradigm of the so called service oriented architecture (SOA), which describes the relationships, that exists among service providers, service consumers, and service brokers and there by provides an abstract execution environment for web services. We refer to a service implemented by combining the functionality provided by other web services as a *composite service*, and the process of developing a composite web service as *service composition* [1, 2].

There are two key aspects in web service composition those are choriography and orchestration.

2.1.1 Orchestration

In Orchestration several web services are involved in an operation. In the operation one central process, can be a web service, leads the other web services and coordinates the execution of different parts of the operation on different web services. As all the data are exchanged via the central coordinator of the orchestration so it needs to understand the specific composition logic and other web services need not to know that they are being incorporated in a composition process and taking part in a larger business process. Every component service considers the central coordinator just as one consumer of its service. Orchestration describes how web services interact with each other through messages, including the business logic and execution order [3, 4].

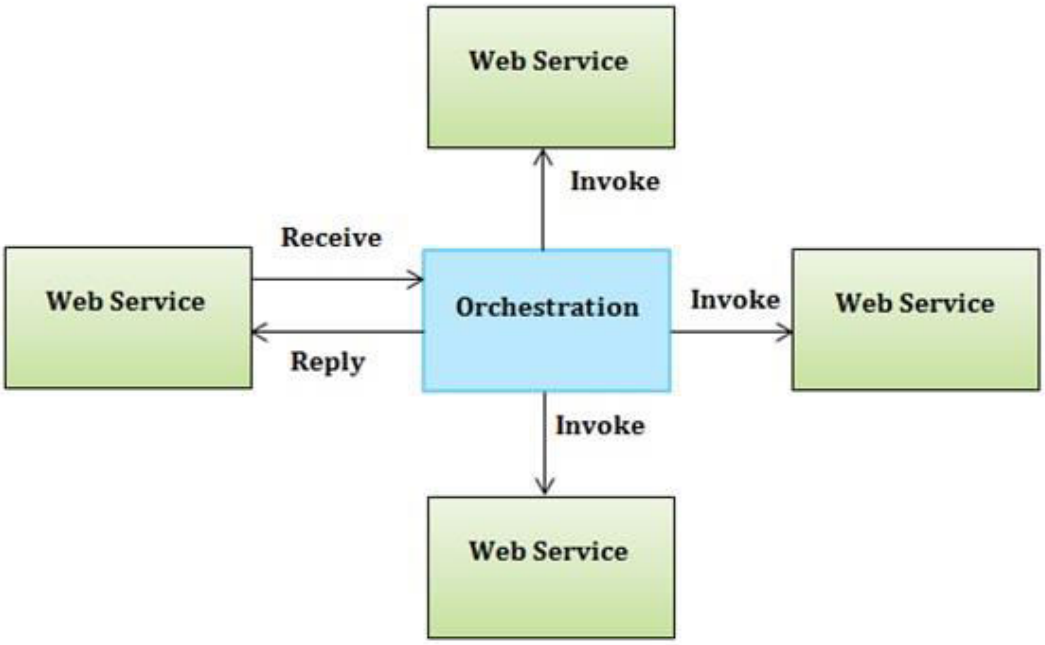

Figure 2.1: Composition of Web Services with Orchestration [3]

2.1.2 Choreography

Choreography is based on collaboration; it does not rely on a central coordinator. In choreography each web service needs to be aware of the business process. All participants need to know when to execute its operations, what messages to exchange, when to exchange the messages and with whom it to interact [3, 4].

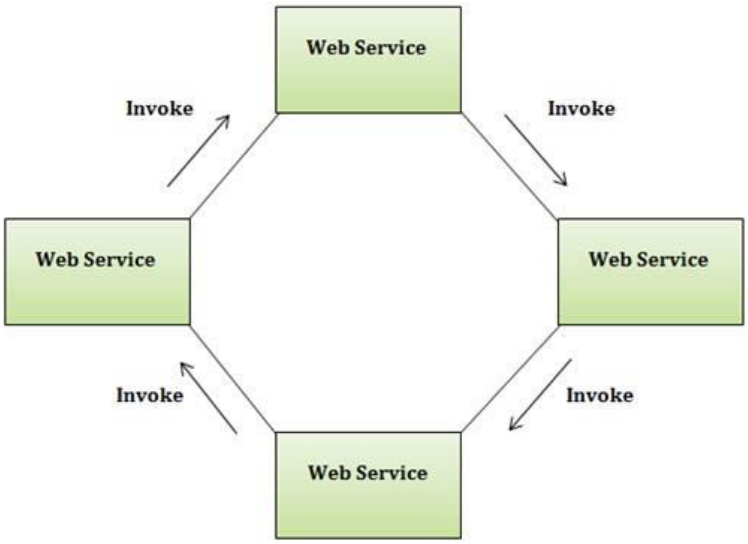

Figure 2.2: Composition of Web Services with Choreography [3]

2.2 FSP

FSP stands for Finite State Processes. Finite State Processes is an algebraic notation to describe process models. The constructed FSP can be used to model the exact transition of workflow processes through a modeling tool such as the Labeled Transition System Analyzer (LTSA), which provides compilation of an FSP into a Labeled Transition System. Models are described using state machines, known as Labeled Transition Systems LTS. These are described textually as finite state processes (FSP) and displayed and analyzed by the LTSA analysis tool. This tool gives an opportunity to test the model workflows before implementing the model. LTS is the graphical form and FSP is the algebraic form [5].

FSP consists of Action Prefix, Process Definition, Choice, Indexed Processes and Actions, Guarded Actions, properties, Constant and Range Declarations, Variable Declaration, Process Alphabets and so on.

2.2.1 Modeling Processes in FSP

A service can be a process or a composition of several processes. A process is the execution of a sequential program. It is modeled as a finite state machine which transits from state to state by executing a sequence of atomic actions. In practical terms, an action might be a communication, a signals, or perhaps, traditional execution of a task [6].

In FSP processes are two types such as Primitive Processes and Composite Processes.

Primitive Processes

Primitive processes are defined using action prefix, choice and recursion. Both action labels and local process names can be indexed or non-indexed.

Action Prefix "->"

Action prefix defines a transition between states. 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 described by P. The action prefix operator "->" always has an action on its left and a process on its right. In FSP, identifiers beginning with a lowercase letter denote actions and identifiers beginning with an uppercase letter denote processes. A primitive process definition is terminated by a full stop [6].

The following definition describes the process CLOCK which repeatedly engages in the action tick.

```
CLOCK = (tick -> CLOCK).
```

The LTS corresponding to the definition above is:

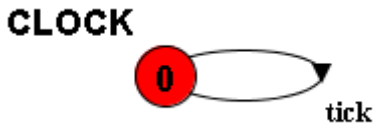

Figure 2.3: LTSA representation of CLOCK process.

Choice "|"

Choice is represented as a state with more than one outgoing transition. Choice operator "|" can express a choice of more than two actions. Choices are of two types, Deterministic and Non-Deterministic. The FSP language provides mechanisms for deterministic and non-deterministic choice. Their definitions are as follows:

Deterministic Choice: If x and y are actions then (x->P | y->Q) describes a process which initially engages in either of the actions x or y. The execution of action x will have subsequent behavior described by P. Similarly, the execution of y will have subsequent behavior described by Q.

The example describes the behavior of a dispensing machine which dispenses coffee if the red button is pressed and tea if the blue button is pressed.

```
DRINKS = (red->coffee->DRINKS | blue->tea->DRINKS).
```

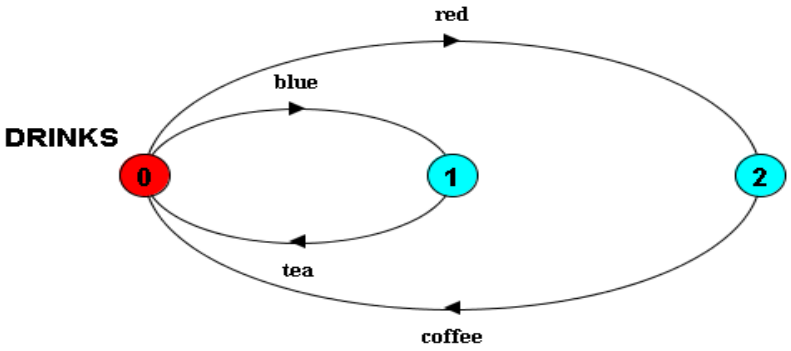

Figure 2.4: LTSA representation of Deterministic process DRINKS.

Non-deterministic Choice: The process (x->P | x->Q) is said to be *non-deterministic* since after the action x, it may behave as either P or Q. The COIN process defined below and drowns as a state machine in Figure is an example of a non-deterministic process [6, 7].

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

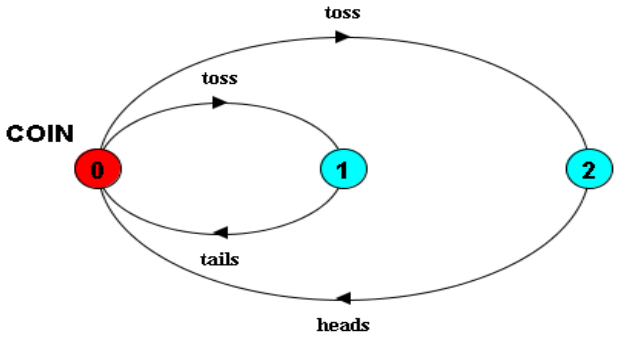

Figure 2.5: LTSA representation of Non-deterministic process COIN.

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..2] -> if x>=1 then (high -> LEVEL)  else (low ->
LEVEL)).
```

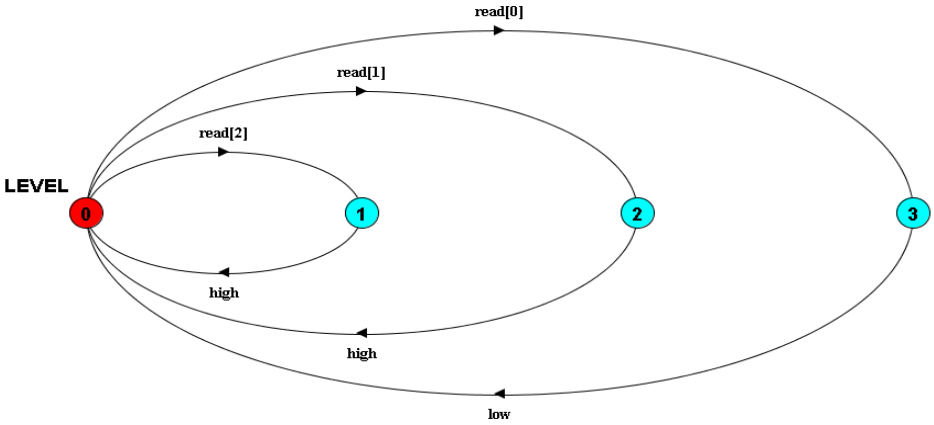

Figure 2.6: LTSA representation of LEVEL process.

Guarded Actions

It is often useful to define particular actions as conditional, depending on the current state of the machine. We use Boolean guards to indicate that a particular action can only be selected if its guard is satisfied. The choice (When B x->P | y->Q) means that when the guard B is true then the actions x and y are both eligible to be chosen, otherwise if B is false then the action x cannot be chosen. The example below is a process that encapsulates a count variable. The count can be increased by inc operations and decreased by dec operations. The count is not allowed to exceed N or be less than zero [6].

```
COUNT (N=3)    = COUNT[0],
COUNT[i:0..N] = (when(i<N) inc->COUNT[i+1]
                |when(i>0) dec->COUNT[i-1]).
```

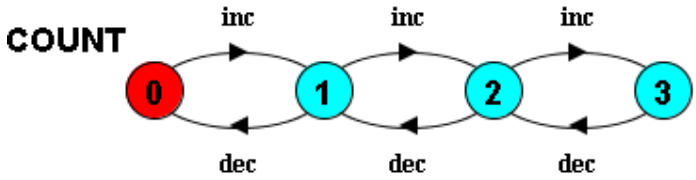

Figure 2.7: LTSA representation of COUNT process.

FSP supports only integer expressions; consequently, the value zero is used to represent false and any non-zero value represents true.

Sequential Composition in FSP

If P is a sequential process and Q is a local process, then P;Q represents the sequential composition such that when P terminates, P;Q becomes the process Q.

Composite Processes

Composite processes are defined using parallel composition, relabeling and hiding.

Parallel Composition in FSP

If P and Q are two processes then (P || Q) represents the concurrent execution of P and Q. The operator || is the parallel composition operator. Parallel composition yields a process, which is represented as a state machine in the same way as any other process. The state machine representing the composition generates all possible interleaving of the traces of its component

processes. Composite process definitions are always preceded by "||" to distinguish them from primitive process definitions. For example, the process:

```
ITCH = (scratch->STOP).
```

has a single trace consisting of the action scratch. The process:

```
CONVERSE = (think->talk->STOP).
```

has the single trace think->talk . The composite process:

```
||CONVERSE_ITCH = (ITCH || CONVERSE).
```

has the following traces

```
think->talk->scratch
think->scratch->talk
scratch->think->talk
```

The state machine representing the composition is formed by the Cartesian product of its constituents [6].

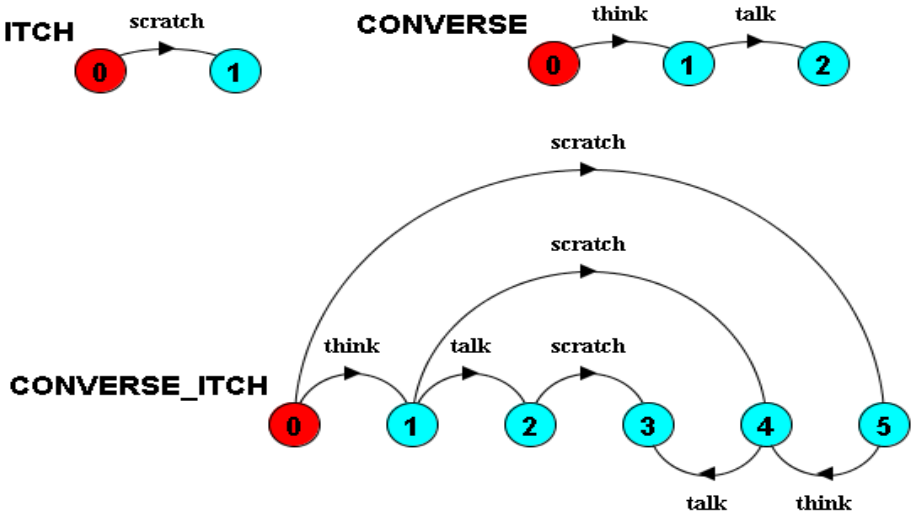

Figure 2.8: LTSA representation of Composition CONVERSE_ITCH.

Modeling interaction - Shared Actions

If processes in a composition have actions in common, these actions are said to be shared. Concurrent processes that share actions interact with each other for synchronization. A shared action must be executed at the same time by all the processes that participate in that shared action while unshared actions may be arbitrarily interleaved. For an example, a process that manufactures an item and then signals that the item is ready for use by a shared ready action. A user can only use the item after ready action occurs. Two items can be made before the first is used which is an undesirable behavior and we do not wish the MAKER process to get ahead in this way. The solution is to ensure that the user indicates that the item is used. The

used action is shared with the MAKER who now cannot proceed to manufacture another item until the first is used. The interaction between MAKER and USER in such a way is an example of a handshake. A handshake is an action acknowledged by another action. Handshake protocols are widely used to structure interactions between processes [6].

```
MAKER = (make->ready->used->MAKER).

USER = (ready->use->used->USER).

||MAKER_USER = (MAKER || USER).
```

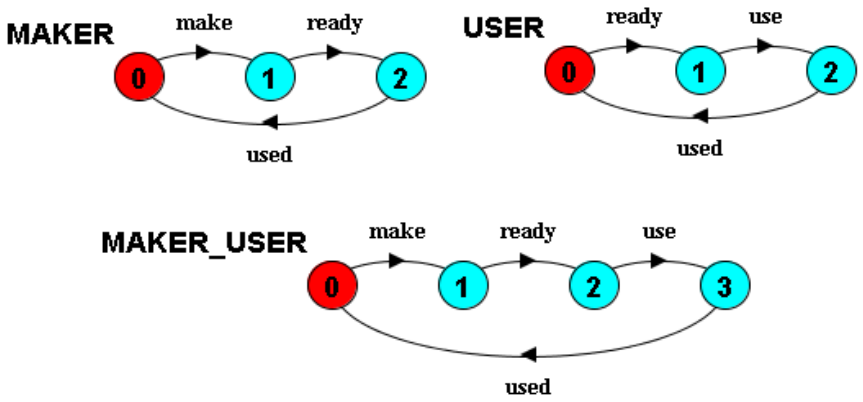

Figure 2.9: LTSA representation of Composition MAKER_USER.

Relabeling Actions in FSP

Relabeling functions are applied to processes and change the names of action labels. This is usually done to ensure that composite processes synchronize on the desired actions. `/{newlabel_1/oldlabel_1,…newlabel_n/oldlabel_n}` is the general form of the relabeling function. For an example, a server process that provides some service and a client process that invokes the service are described below.

```
CLIENT = (call->wait->continue->CLIENT).

SERVER = (request->service->reply->SERVER).
```

Using relabeling we can associate call action of the CLIENT with the request action of the SERVER and similarly the reply and the wait actions.

```
||CLIENT_SERVER = (CLIENT || SERVER)
                  / {call/request reply/wait}.
```

The effect of applying the relabeling function can be seen in the state machine as the label call replaces request in SERVER and reply replaces wait in CLIENT [6].

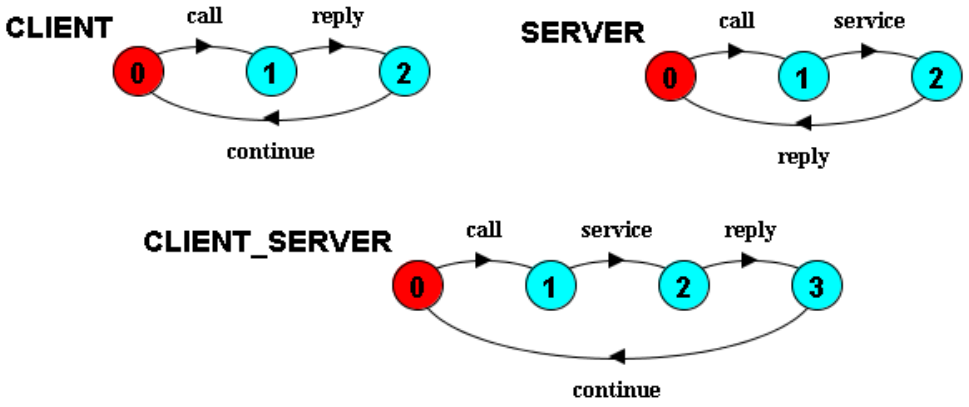

Figure 2.10: LTSA representation of Relabeling in CLIENT_SERVER.

Hiding "\" and "@"

Hiding removes action names from the alphabet of a process and thus makes these concealed actions "silent". By convention, these silent actions are labeled "tau". The general form of a hiding expression is \ {set of labels to be hidden}. Sometimes it is more convenient to state the set of action labels, which are visible and hide all other labels. This is expressed by @ {set of visible labels} [6].

2.2.2 Property Processes to verify the System

Safety Properties

Safety properties are specified to LTSA as deterministic primitive processes which contain no silent (tau) transitions (no hiding). Safety property processes are denoted by the keyword property. They are composed with a target system to ensure that the specified property holds for that system. Composing a property process with a set of processes does not affect their normal operation. If behavior can occur that violates the safety property, then a transition to the ERROR state results. For example, the following property specifies that only behavior in which knock occurs before enter is acceptable [6].

```
property POLITE = (knock->enter->POLITE).
```

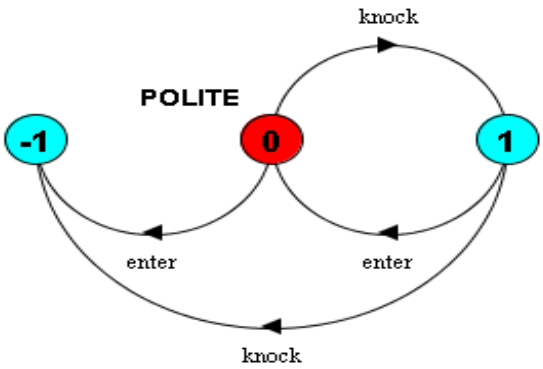

Figure 2.11: LTSA representation of Property POLITE.

CHAPTER 3

‘ONLINE MARKETPLACE’ Service Composition

3.1 ‘ONLINE MARKETPLACE’ Web Service

“Online Marketplace” is an online service which designed for maintain communication between Buyer and Seller for online purchase. Here, partner web service Marketplace uses two separate partner web service – A Buyer who requests product to the Marketplace for purchase; and A Seller who receives product request from Buyer via Marketplace and arrange products for sell. There are supporting web services – SUPPLIER which is related to SELLER and TRANSACTION which has connection with BUYER and TRANSACTION.

In this example model, a Buyer requests product to Marketplace, Marketplace receives the request and forwards into the relevant seller for the price. Seller also forwards the request to Supplier for the availability of products and wants quotation about price. If we take that all actions are reacting positively then the scenario will be – Supplier have available products and send product list with price to the seller, seller forward it to the Marketplace and Marketplace forward it to the Buyer. Price argument is a possible scenario here where Buyer requests for price changing to the seller through the MP. Again, we are taking positive action that SELLER accepts the price and ready to sell. Buyer receives this acknowledgement through MP and forward payment to transaction. MP will receive payment from Transaction and forward it to the seller. After receiving payment seller will send the product to Buyer. There can be negative possibilities also and they are considering as Compensation. Here, supplier can be running out of stocks, Seller cannot be agreed to the demanding price of Buyer or Buyer can cancel order anytime. For each negative action all the steps will be canceled which has generated before. In the Transaction state, transfer of payment from Buyer can be failed, forwarding payment to MP or Buyer can also fail. In each case Transaction will start from the beginning.

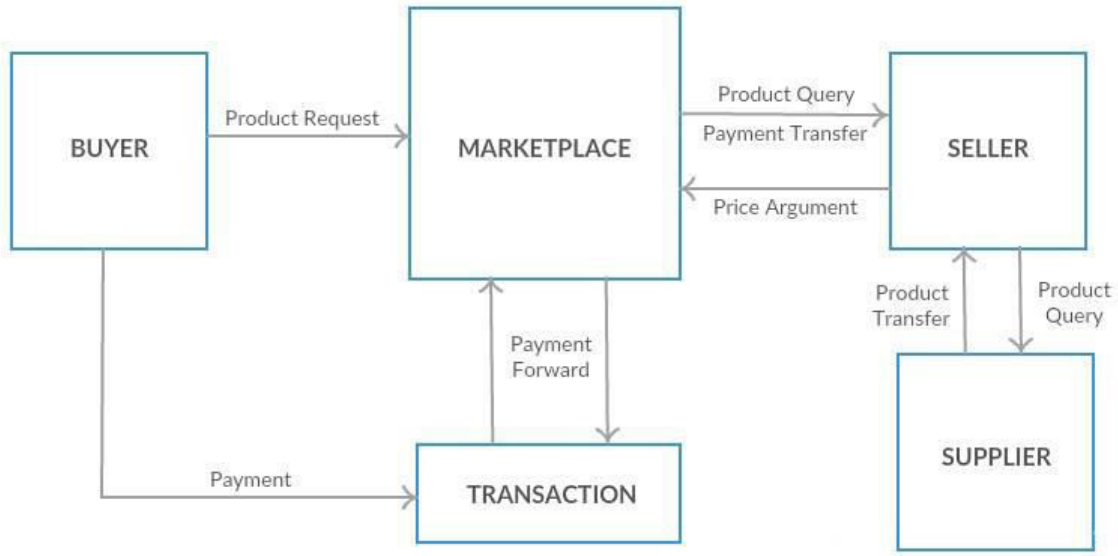

Fig. 3.1: System Composition

3.1.1 BUYER

In the system, at first buyer request a product to the Marketplace to give an order and Purchase it. According to order Buyer receives a price list of available product from marketplace. Buyer can either accept the price or may bargain. For bargaining, Buyer requests price to marketplace and receives reply from marketplace which could be positive or negative. Buyer confirms the product by sending ACK or rejects it by sending NAK. The rejection is a compensation state. Buyer transfer payment to the Transaction process and receive confirmation of the success by ACK/NAK. After successful transaction Buyer receives delivery.

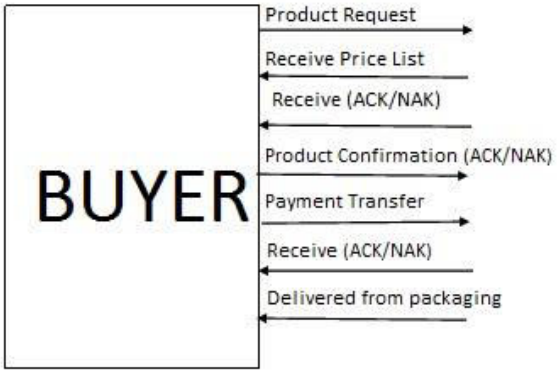

Figure 3.2: BUYER

3.1.2 Marketplace

After receiving product requests from Buyer, the marketplace sends product query to seller and receive product list with price from seller. Then the list is forwarded to Buyer and receive price request from Buyer for beginning. Marketplace forwards the requested price to seller and receives reply of agreement or rejection. Then marketplace receives both order confirmation and product confirmation from Buyer and seller. After all positive actions, marketplace receives payment from transaction and forwards it to the buyer. For all the rejection or failed transaction, Compensation process will run where all the running states will be terminate by throwing a packaging.

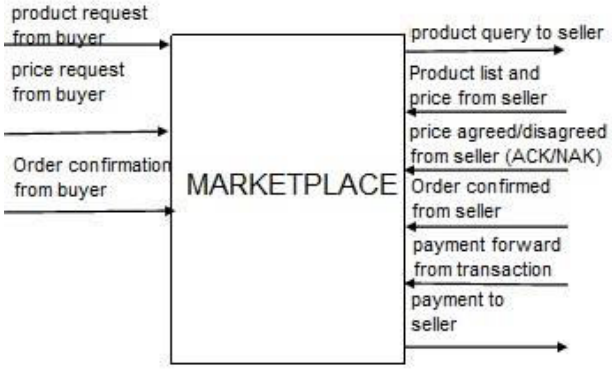

Figure 3.3: MARKETPLACE

3.1.3 SELLER

SELLER service is the connection between marketplace and supplier. It receives product quotation from marketplace and forwards it to SUPPLIER. SUPPLIER replied the product quotation with price list and SELLER forwards it to Marketplace. SELLER receive price request from marketplace for bargain and can accept it or reject it. The rejection is the compensation state and will undo all the previous states. For the acceptance of price, a confirmation message will send to both marketplace and SUPPLIER. SUPPLIER receives products from marketplace and then sends the product for delivery. Any rejection or failure of transaction will throw a cancelation message.

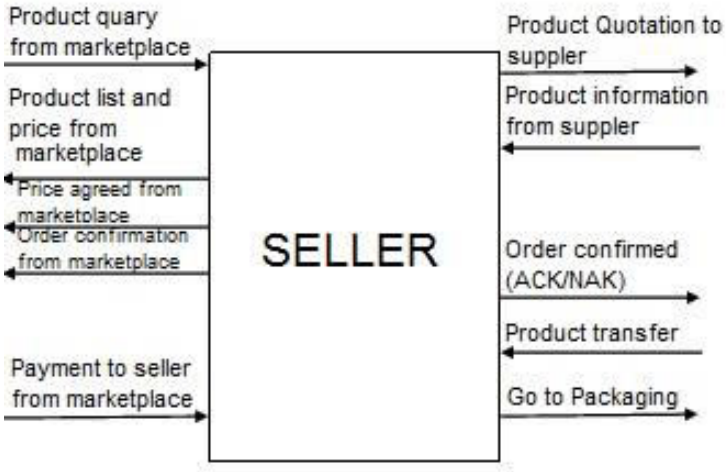

Figure 3.4: SELLER

3.1.4 SUPPLIER

The SUPPLIER web service starts by receiving product quotation from SELLER. Then SUPPLIER sends product information to SELLER. After receiving order confirmation SUPPLLIER sends a reply message with positive or negative reply. The negative reply is Compensation which will cancel all the running actions. Supplier will finally transfer product to the Seller.

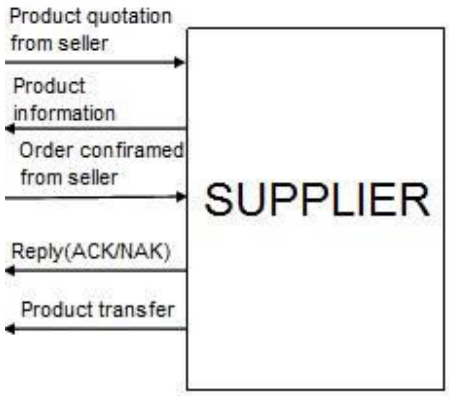

Figure 3.5: SUPPLIER

3.1.5 Transactions

The Transaction web service is responsible for all the payment transfer. At first it will receive payment from the Buyer and reply confirmation by sending messages to Buyer. Then it forwards payment to Marketplace and wait for reply message for confirmation. All successful transfer of payment will close the service but a single failure will turn of all the running states.

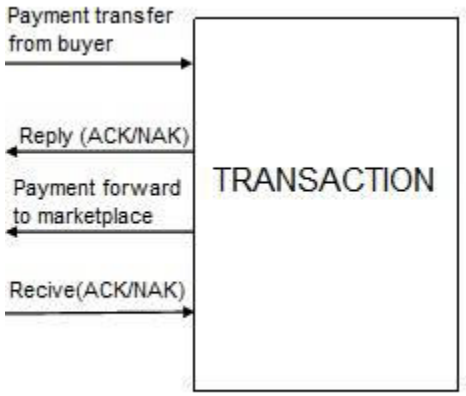

Figure 3.6: TRANSACTION

A message sequence chart is here to describe relations between the web services:-

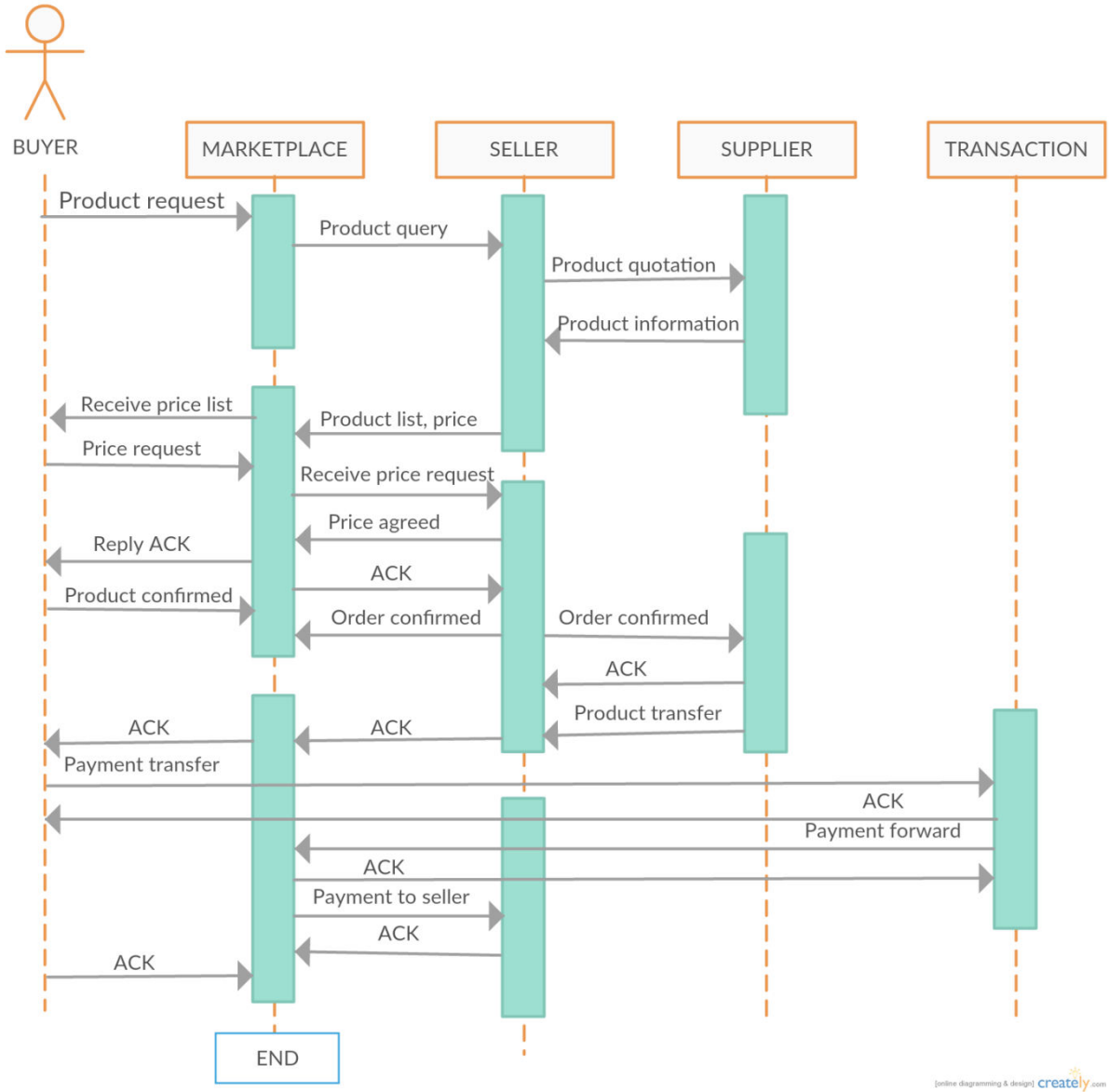

Figure 3.7: A Message Sequence Chart in 'Process Based Service Composition & Verification'

3.2 Compensation in Online Marketplace Web Service

3.2.1 Compensation

To give a proper service it needs interaction between services. One service can call another service and need to deals with error occurs during interaction. If any negative acknowledgement is thrown by any service it is considered as a fault or error of the system for which service cannot be continued anymore. That's why we have to handle errors by compensating the services. A mechanism is used to handle the errors that can arise in any stage of communication between services is called compensation. Using the compensation mechanism all services can reach in their initial state from where they have been interrupted.

3.2.2 Compensation Mechanism of Online Marketplace Web Service

In our model, we have designed each of the web services independently to handle compensation. While a negative acknowledgement is thrown by any service then the service itself will run the compensation process and also throws indication to the other services to heading towards the compensation state. The reverse actions are performed to compensate the other services from where the interruption occurs.

In the case of rejection, Buyer will send negative acknowledgement (NAK) which will compensate Buyer's activities and will throw a cancel message to compensate all the other web services.

Marketplace will compensate when BUYER or SELLER sends NAK to marketplace. After receiving NAK Marketplace itself will cancel all the actions and reverse the states and will also send cancelation messages to other process to terminate all the actions.

SELLER can be disagreed to the requested price of BUYER and throw an interruption to its related processes to compensate.

The failure message of the Supplier will be received by a compensation process which will compensate Supplier's activities and will throw an interrupt to the SELLER process to terminate and forward messages to compensate others.

Transaction state will compensate for each failure of payment transfer. But if it receives NAK then it will throw a wait messages to the related state and after the timeout it will run the compensation to terminate and throw interruption to others for compensate also.

A message sequence chart is here to describe the web services with Compensation:-

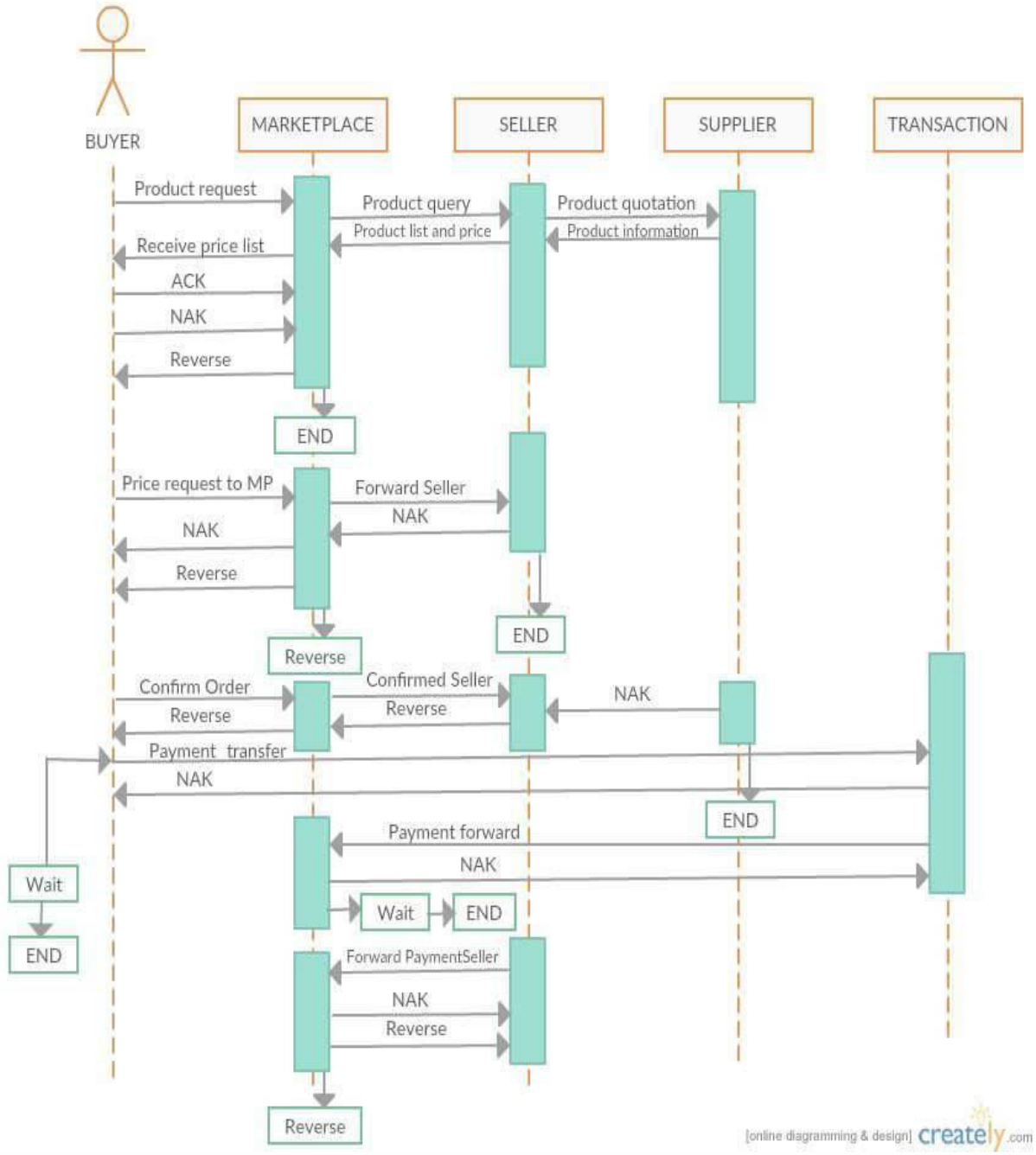

Figure 3.8: A Message Sequence Chart in 'Process Based Service Composition & Verification' with Compensation

CHAPTER 4

Service Composition in FSP

4.1 Coding Representation

In our system we have five major processes which have their own compensation process and safety property to ensure a good composition. In FSP we modeled the system like that, a Buyer requests product to Marketplace, Marketplace receives the request and forwards into the relevant seller for the price. Seller also forwards the request to Supplier for the availability of products and wants quotation about price. If we take that all actions are reacting positively then the scenario will be – Supplier have available products and send product list with price to the seller, seller forward it to the Marketplace and Marketplace forward it to the Buyer. Price argument is a possible scenario here where Buyer requests for price changing to the seller through the MP. Again, we are taking positive action that SELLER accepts the price and ready to sell. Buyer receives this acknowledgement through MP and forward payment to transaction. MP will receive payment from Transaction and forward it to the seller.

4.2 Modeling the 'ONLINE MARKETPLACE' Service in FSP

ONLINE MARKETPLACE is divided into five major services. Each service contains its own general process and its compensation process. Main Compensation Process handles any kind of anomaly that occurs in the System.

4.2.1 Declaring Original Processes

BUYER

Buyer process consists of a sequence of actions. The process starts the service by requesting a product by sending pro.req.to.mp for a product to the MARKETPLACE. When Buyer receives a list from MARKETPLACE named as rcv.prc.list. Then for bargaining BUYER send a request price.req.to.mp and if got confirmation as ack then send order confirmation by sending conf.ordr . If MARKETPLACE confirms product as pro.conf.mp then BUYER sends the payment to TRANSACTION as pay.trans. After successful transaction notified by ack.tr BUYER terminate through END.

```
BUYER= (pro.req.to.mp->rcv.prc.list->price.req.to.mp->ack
      ->conf.ordr->pro.conf.mp->pay.trans->ack.tr->END).
```

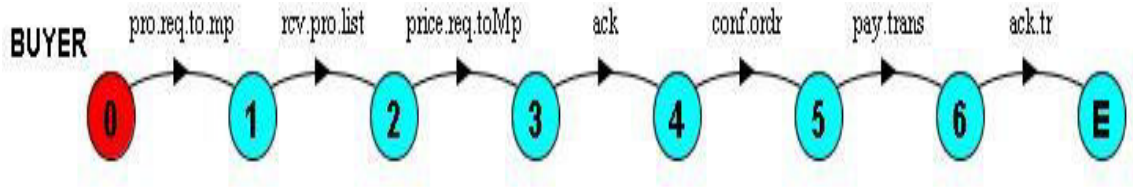

Figure 4.1: LTSA representation of Buyer Process

MARKETPLACE

MARKETPLACE is a process interacts with other three partner service processes. This process starts with receiving a request from BUYER labeled as rcvpro_req.It requested product query from seller and receive quotation as pque_slr and rcv_prlist respectively. It forwards BUYER's bargaining price to SELLER and receive reply also. Positive reply is received as and it is forwarded to BUYER. It receives both order and product confirmation from both BUYER and SELLER respectively labeled as rcv_ordr_conf_buyer and rcv_pro_conf. MARKETPLACE also receives payment by rcv_payment from TRANSACTION and forward to SELLER as fwd_payment_slr.

```
MP= (rcvpro_req->pque_slr->rcv_prlist->fwd_prlist_buyer
      ->rcv_prcreq_buyer->fwd_prcreq_slr->rcv.ack.frm.slr
      ->send_ack_buyer->rcv_ordr_conf_buyer->send_ack_slr
      ->rcv_pro_conf->send.pro.conf.bur->rcv_payment
      ->fwd_payment_slr->END).
```

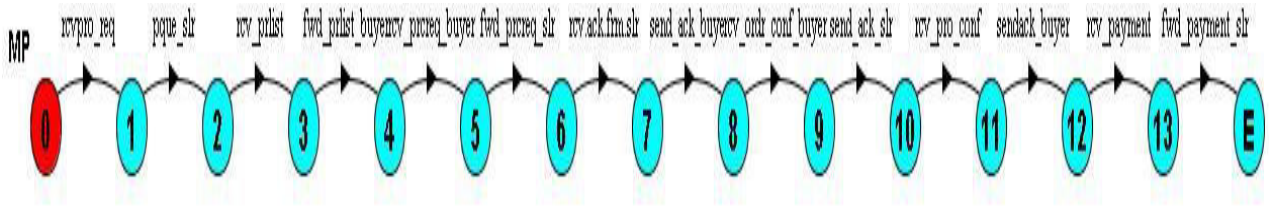

Figure 4.2: LTSA representation of MARKETPLACE Process

SELLER

SELLER process receives product quotation as rcv_pro_query and forwards it to SUPPLIER by req_pro_qt . SUPPLIER replied the product quotation and SELLER forwards it to Marketplace respectively by rcv.info and send_pro_list. SELLER receives price request from BUYER through MARKETPLACE as rcv_prc_req and if agreed then it will send prc.agreed.

It sends confirmation messages to both MARKETPLACE as send_pro_conf and SUPPLIER as send_or_conf_sup. SELLER receives products availability from supplier and also receives payment from MARKETPLACE by rcvpayment.frm.mp.

```
SELLER=(rcv_pro_query->req_pro_qt->rcv.info->send_pro_list
      ->rcv_prc_req->prc.agreed->rcv_order_conf->send_or_conf_sup
      ->rcv.pro.avail->send_pro_conf->rcv_pro->rcvpayment.frm.mp
      ->send_for_pckg->END).
```

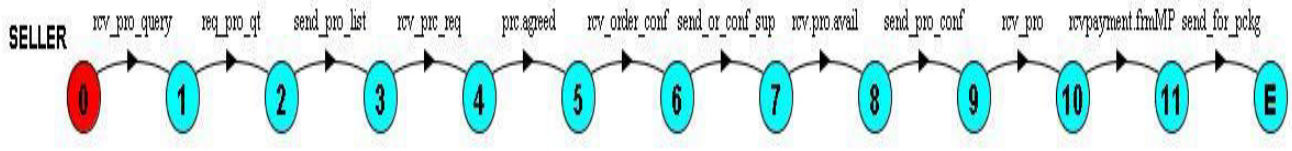

Figure 4.3: LTSA representation of SELLER Process

SUPPLIER

Supplier receives a request for quotes from the SELLER by rcv.pro.qt. According to the request, Supplier sends accumulated quotes to the SELLER by send.info.slr. After receiving order confirmation from MARKETPLACE, SELLER sends confirmation to SUPPLIER by the action labeled rcv.pro.conf . If the Supplier able to deliver the order it confirms to SELLER by a action pro.avail and then transfer product by trans.pro.

```
SUPPLIER= (rcv.pro.qt->send.info.slr->rcv.pro.conf->pro.avail
          ->trans.pro->rcv.ack.slr->END).
```

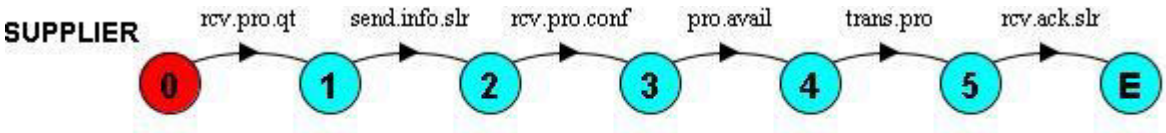

Figure 4.4: LTSA representation of Supplier Process

TRANSACTION

Transaction receives payment from BUYER process by rcv.payment.buyer. And if receive payment then sends positive acknowledgement send.ack.bur and then forward payment to

MARKETPLACE as paymentfwd.to.mp. It receives ack after successful transfer as a positive reply.

```
TRANSACTION= (rcv.payment.buyer->send.ack.bur->paymentfwd.to.mp->ack
            ->END).
```

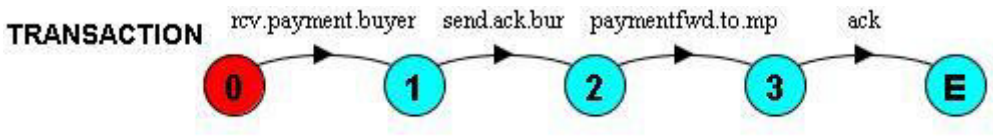

Figure 4.5: LTSA representation of Transaction Process

4.2.2 Declaring Compensation Processes
Compensation Process for BUYER

Buyer's compensation process is completed by BUYER itself. Every time BUYER recives any negative reply as mp.cancel.ordr, rcv.nak.tr, nak will compensate itself by undoBYR.

```
BUYER= (pro.req.to.mp->rcv.prc.list->prc.req.to.mp->(ack
      ->(cancel.ordr->END|ordr.conf->(mp.cancel.ordr->undoBYR
      ->BUYER|mp.conf->pay.trans->(rcv.ack.tr->END|rcv.nak.tr
      ->pay.trans->END))|nak->BUYER)).
```

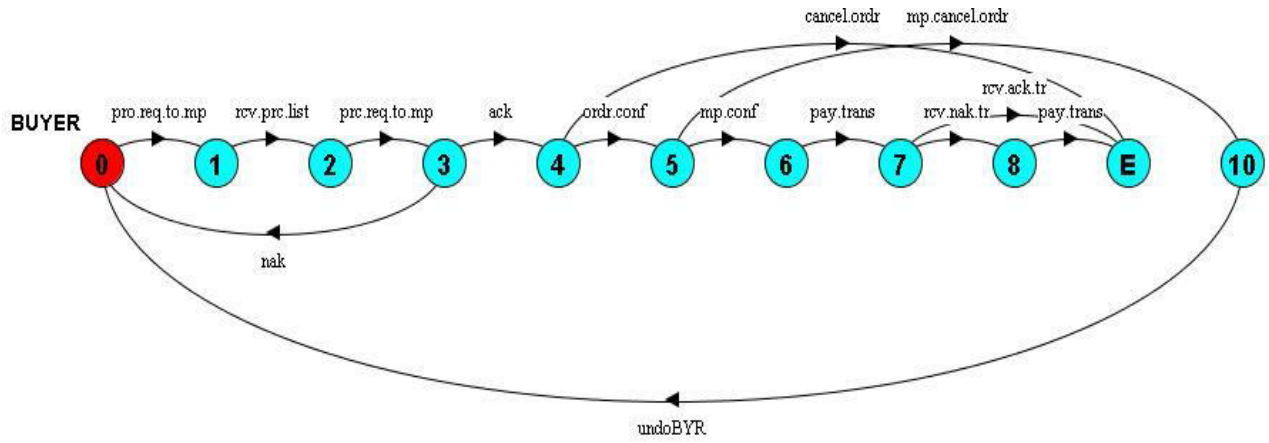

Figure 4.6: LTSA representation of BUYER Process with compensation

Compensation Process for MARKETPLACE

Marketplace can receive negative reply from any process. BUYER can cancel order by sending byr.cancel.ordr. SELLER can be disagreed to the price request by price.nak . Seller can cancel order or transfer of payment may failed. For each scenario MARKETPLACE will send a negative reply like cancel.ordr.to.slr, send_nak_buyer, price.nak.bur to the related processes and reverse itself to terminate the process.

```
MP= (rcvpro_req->pque_slr->rcv_prlist->fwd_prlist_buyer
     ->rcv_prcreq_buyer->fwd_prcreq_slr->(rcv.ack.frm.slr
     ->send_ack_buyer->(byr.cancel.ordr->cancel.ordr.to.slr
     ->undomp->MP|rcv_ordr_conf_buyer->send_ack_slr->rcv_pro_conf
     ->sendack_buyer->(rcv_payment->fwd_payment_slr
     ->END|rcv.nak.frm.slr->send_nak_buyer->thrws
     ->END)|slr.cancel.ord->send.cencel.to.bur->undomp
     ->MP)|price.nak->price.nak.bur->MP)).
```

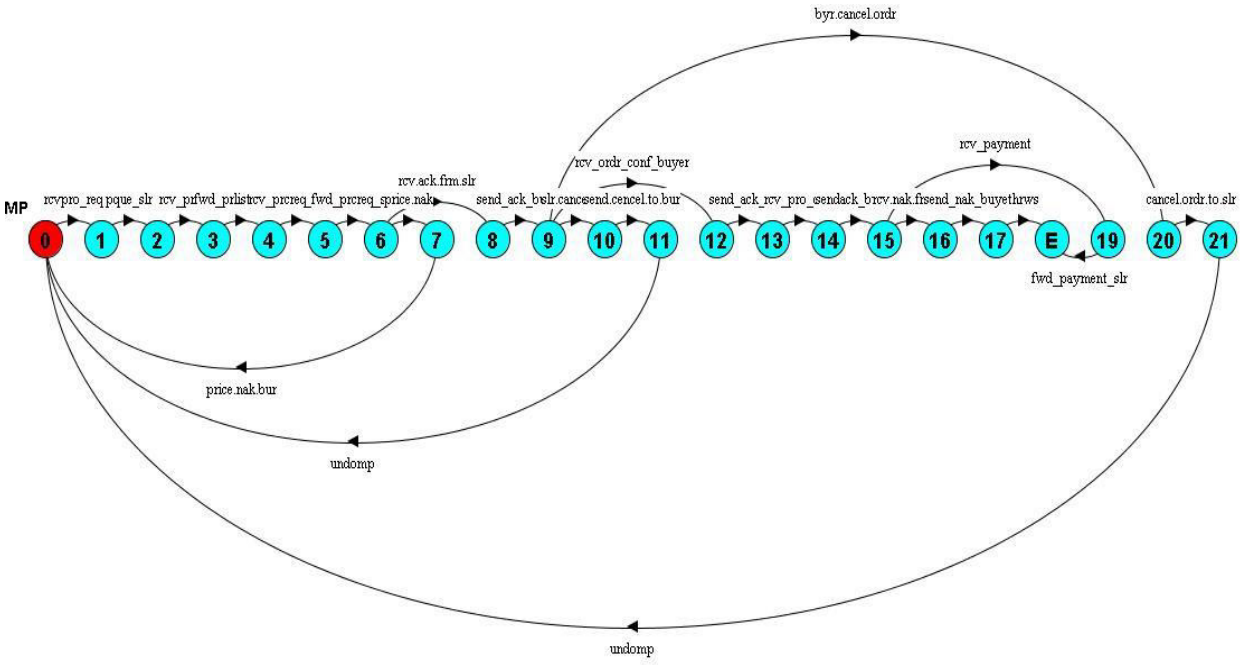

Figure 4.7: LTSA representation of MARKETPLACE Process with compensation

Compensation Process for SELLER

SELLER process maintains relation between MARKETPLACE and SUPPLIER. Each time it receives any negative action from them, it will throw a termination/reverse message to others for running compensation. SELLER receives negative indications as mp.cancel.ordr, rcv.cant.supp.SELLER can disagreed to price and then send messages to others for compensation like send.nak.

```
SELLER= (rcv_pro_query->req_pro_qt->send_pro_list->rcv_prc_req
      ->(prc.agreed->rcv_order_conf->send_or_conf_sup
      ->(mp.cancel.ordr->send.canle.sup->undoslr
      ->SELLER|rcv.pro.avail->send_pro_conf->rcv_pro->(ack.to.sup
      ->rcvpayment.frmMP->send_for_pckg->END|rcv.cant.supp
      ->send.cancel.to.mp->cancel->END))|send.nak->END)).
```

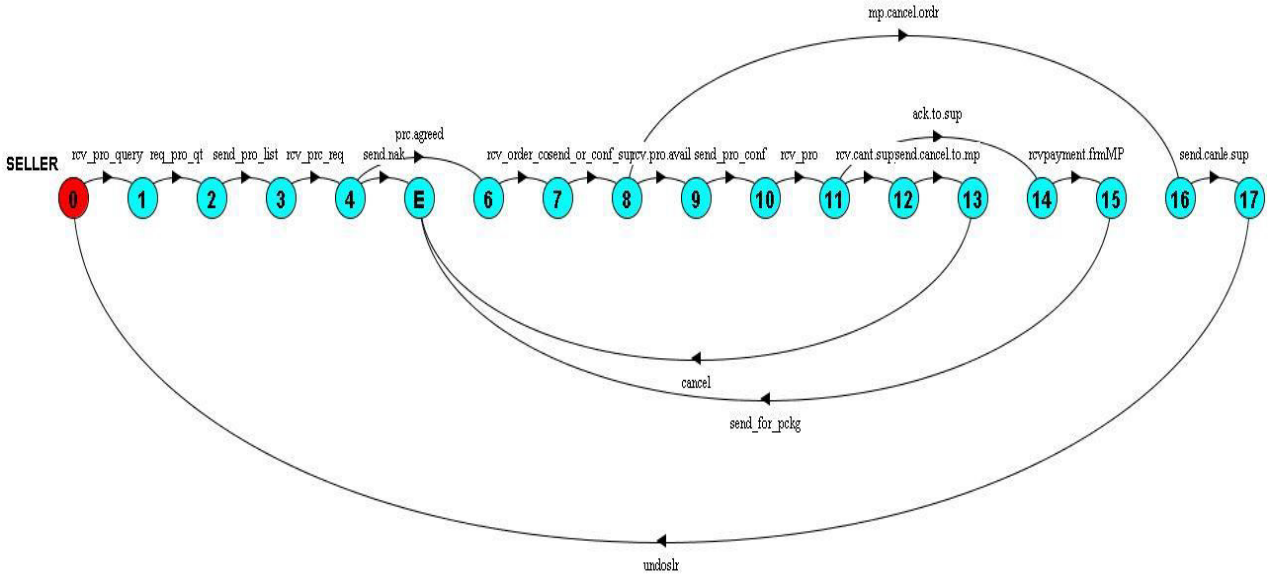

Figure 4.8: LTSA representation of SELLER Process with compensation

Compensation Process for Supplier

SUPPLIER process will compensate and reverse itself when it receives slr.cancel.ordr or rcv.nak.slr from SELLER. When SUPPLIER is unable to send product then it will run compensation by reversing itself and will send messages to other processes to compensate like cant.supply.to.slr.

```
SUPPLIER= (rcv.pro.qt->send.info.slr->rcv.pro.conf->(slr.cancel.ordr
      ->undosup->SUPPLIER|pro.avail->trans.pro->(rcv.ack.slr
```

```
->END|rcv.nak.slr->trans.pro->END)|cant.supply.to.slr
->SUPPLIER)).
```

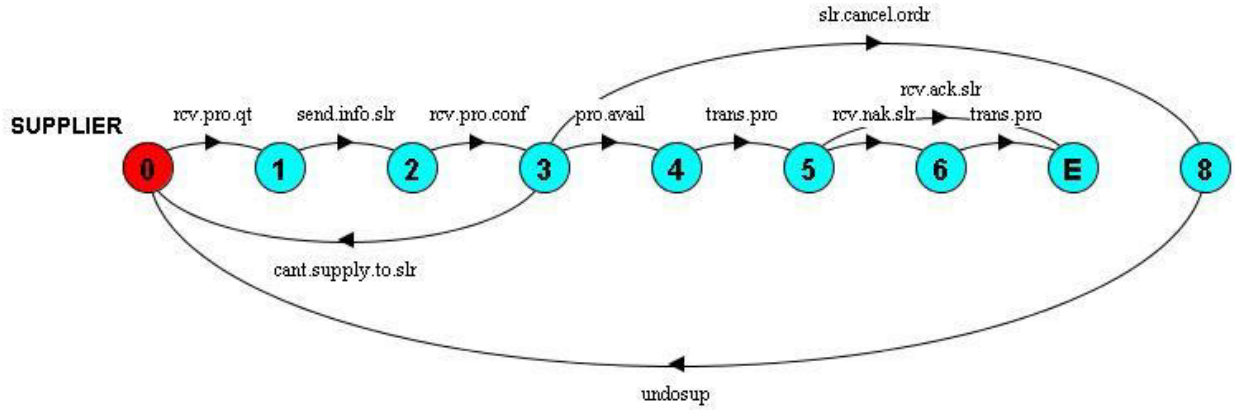

Figure 4.9: LTSA representation SUPPLIER Process with compensation

Compensation Process for TRANSACTION

If TRANSACTION process does not receive payment to BUYER then it will reverse itself and send negative acknowledgement as to BUYER and it will terminate BUYER also. Negative reply from MARKETPLACE is an indication towards compensation of TRANSACTION.

```
TRANSACTION=(rcv.payment.buyer->(send.nak.bu->failed-
>TRANSACTION|send.ack->paymentfwd->(ack->END|nak->reverse->failed-
>TRANSACTION))).
```

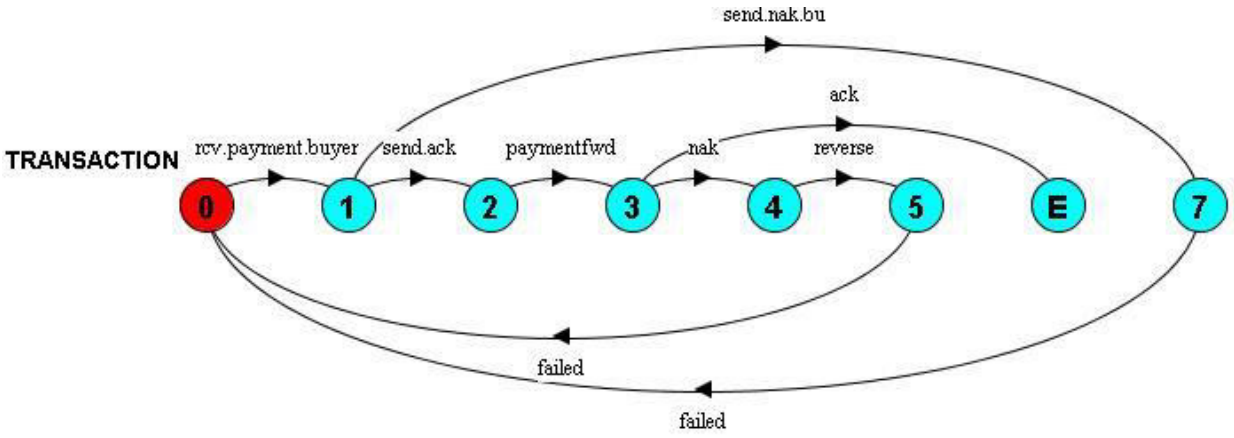

Figure 4.10: LTSA representation of TRANSACTION Process with compensation

4.2.3 BUYER & MARKETPLACE Parallel Process

It is the parallel process of BUYER and MARKETPLACE. After process relabeling, there are some common actions and messages which is throwing by a process and is receiving by another one.

```
BUYER=(pro.req.to.mp->rcv.prc.list->prc.req.to.mp->(ack
      ->(cancel.ordr->END|ordr.conf->(mp.conf->pay.trans
      ->(rcv.ack.tr->END|rcv.nak.tr->undotr->END))|mp.canc.ordr
      ->END)|nak->END)).
MP=(rcvpro_req->pque_slr->rcv_prlist->fwd_prlist_buyer
      ->rcv_prcreq_buyer->(rcv.ack.frm.slr->send_ack_buyer
      ->(byr.cancel.ordr->cancel.ordr.to.slr->END|
      rcv_ordr_conf_buyer->send_ack_slr->rcv_pro_conf
      ->sendack_buyer->(rcv.payment->fwd_payment_slr
      ->END|rcv.nak.frm.slr->send_nak_buyer->END)|slr.cancel.ord
      ->send.cencel.to.bur->undomp->MP)|prc.nak->fwd.prc.nak->END)).
SELLER=(rcv_pro_query->req_pro_qt->send_pro_list->(prc.agreed
      ->rcv_order_conf->send_or_conf_sup->(mp.cancel.ordr
      ->send.canle.sup->END|rcv.pro.avail->send_pro_conf->rcv_pro
      ->(ack.to.sup->rcvpayment.frm.mp->send_for_pckg
      ->END|rcv.cant.supp->send.cancel.to.mp->cancel
      ->END))|disagreed->END)).
SUPPLIER=(rcv.pro.qt->send.info.slr->rcv.pro.conf->(slr.cancel.ordr
      ->END|pro.avail->trans.pro->(rcv.ack.slr->END|rcv.nak.slr
      ->trans.pro->END)|cant.supply.to.slr->END)).
TRANSACTION=(rcv.payment.buyer->(send.nak.bu->failed
      ->TRANSACTION|send.ack->paymentfwd.to.mp->(ack.mp->END|nak.mp
      ->reverse->failed->END))).
||B= (BUYER||MP)/
{pro.req.to.mp/rcvpro_req,rcv.prc.list/fwd_prlist_buyer,prc.req.to.m
p/rcv_prcreq_buyer,pque_slr/rcv_pro_query,prc.agreed/rcv.ack.frm.slr
,send.nak/price.nak,send.cencel.to.bur/mp.canc.ordr,nak/fwd.prc.nak,
prc.nak/disagreed,prc.req.to.mp/rcv_prcreq_buyer,rcv.prc.req/fwd_prc
req_slr,prc.agreed/rcv.ack.frm.slr,send_ack_buyer/ack,
```

cancel.ordr/byr.cancel.ordr,cancel.ordr.to.slr/mp.cancel.ordr,cant.supply.to.slr/mp.send.nak,mp.send.nak/send_nak_buyer,pro.avail/mp.conf,pay.trans/rcv.payment.buyer,rcv.ack.tr/send.ack,rcv.nak.tr/send.nak.bu,paymentfwd.to.mp/rcvpayment.frm.mp,

ordr.conf/rcv_ordr_conf_buyer,send_ack_slr/rcv_order_conf,pro.avail/rcv_pro_conf,pro.avail/rcv.pro.avail,rcv_prc_req/fwd_prcreq_slr,paymentfwd.to.mp/rcv.payment,fwd_prlist_buyer/rcv_pro_query,rcv.pro.qt/req_pro_qt,rcv_prlist/send_pro_list,prc.req.to.mp/rcv_prcreq_buyer,rcvpayment.frm.mp/fwd_payment_slr,ack.to.sup/rcv.ack.slr,

rcv_pro/trans.pro,slr.cancel.ordr/send.canle.sup,slr.cancel.ordr/mp.cancel.ordr,rcv_prlist/send_pro_list,cant.supply.to.slr/rcv.cant.supp}.

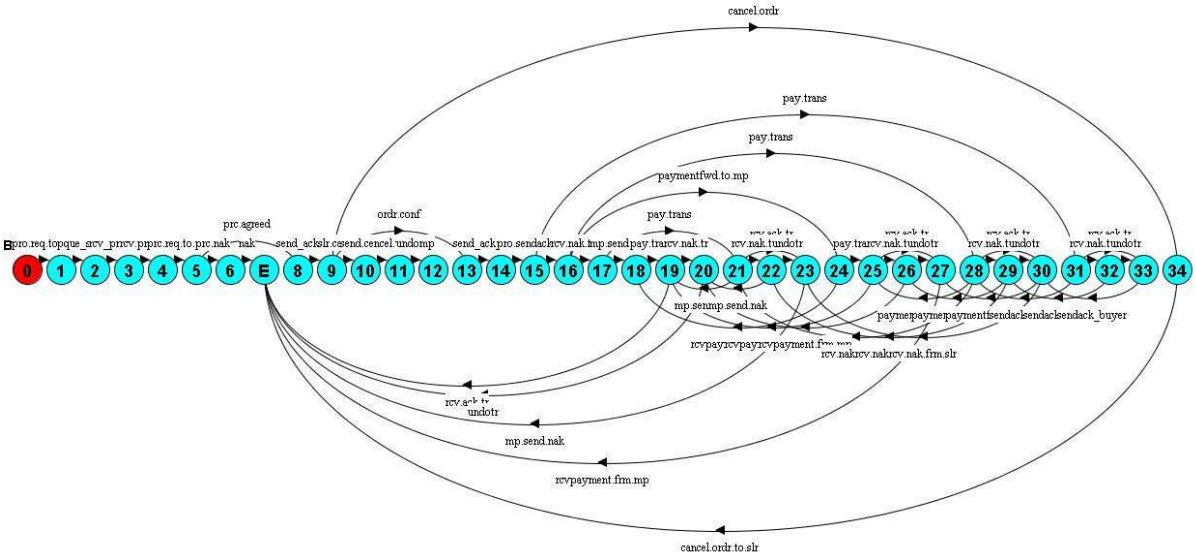

Figure 4.11: LTSA representation of BUYER & MARKETPLACE Parallel Process

4.2.5 Main Compensation Process

```
BUYER= (pro.req.to.mp->rcv.prc.list->prc.req.to.mp->(ack
      ->(cancel.ordr->END|ordr.conf->(mp.conf->pay.trans->(rcv.ack.tr
      ->END|rcv.nak.tr->undotr->END))|mp.canc.ordr->END)|nak->END)).
MP= (rcvpro_req->pque_slr->rcv_prlist->fwd_prlist_buyer
      ->rcv_prcreq_buyer->rcv.ack.frm.slr->M|prc.nak->fwd.prc.nak
      ->END).
M= (send_ack_buyer->(byr.cancel.ordr->cancel.ordr.to.slr
      ->END|rcv_ordr_conf_buyer->send_ack_slr->rcv_pro_conf
      ->sendack_buyer->(rcv.payment->fwd_payment_slr
```

```
      ->END|rcv.nak.frm.slr->send_nak_buyer->END)|slr.cancel.ord

      ->send.cencel.to.bur->undomp->MP)).

SELLER= (rcv_pro_query->req_pro_qt->send_pro_list->prc.agreed

      ->S|disagreed->END).

S= (rcv_order_conf->send_or_conf_sup->(mp.cancel.ordr->send.canle.sup

      ->END|rcv.pro.avail->send_pro_conf->rcv_pro->(ack.to.sup

      ->rcvpayment.frm.mp->send_for_pckg->END|rcv.cant.supp

      ->send.cancel.to.mp->cancel->END))).

SUPPLIER= (rcv.pro.qt->send.info.slr->rcv.pro.conf->(slr.cancel.ordr

      ->END|pro.avail->trans.pro->(rcv.ack.slr->END|rcv.nak.slr

      ->trans.pro->END)|cant.supply.to.slr->END)).

TRANSACTION= (rcv.payment.buyer->(send.nak.bu->failed

      ->TRANSACTION|send.ack->paymentfwd.to.mp->(ack.mp->END|nak.mp

      ->reverse->END))).


||B= (BUYER||MP||TRANSACTION||SUPPLIER||SELLER)/

{pro.req.to.mp/rcvpro_req,rcv.prc.list/fwd_prlist_buyer,prc.req.to.mp/
rcv_prcreq_buyer,pque_slr/rcv_pro_query,prc.agreed/rcv.ack.frm.slr,sen
d.nak/price.nak,send.cencel.to.bur/mp.canc.ordr,nak/fwd.prc.nak,prc.na
k/disagreed,
prc.req.to.mp/rcv_prcreq_buyer,rcv.prc.req/fwd_prcreq_slr,prc.agreed/r
cv.ack.frm.slr,send_ack_buyer/ack,cancel.ordr/byr.cancel.ordr,cancel.o
rdr.to.slr/mp.cancel.ordr,cant.supply.to.slr/mp.send.nak,
mp.send.nak/send_nak_buyer,pro.avail/mp.conf,pay.trans/rcv.payment.buy
er,rcv.ack.tr/send.ack,rcv.nak.tr/send.nak.bu,paymentfwd.to.mp/rcvpaym
ent.frm.mp,
ordr.conf/rcv_ordr_conf_buyer,send_ack_slr/rcv_order_conf,pro.avail/rc
v_pro_conf,pro.avail/rcv.pro.avail,rcv_prc_req/fwd_prcreq_slr,paymentf
wd.to.mp/rcv.payment,
fwd_prlist_buyer/rcv_pro_query,rcv.pro.qt/req_pro_qt,rcv_prlist/send_p
ro_list,prc.req.to.mp/rcv_prcreq_buyer,rcvpayment.frm.mp/fwd_payment_s
lr,
ack.to.sup/rcv.ack.slr,rcv_pro/trans.pro,slr.cancel.ordr/send.canle.su
p,slr.cancel.ordr/mp.cancel.ordr,rcv_prlist/send_pro_list,
cant.supply.to.slr/rcv.cant.supp,send.ack/rcv.ack.tr,rcv.payment/ack.m
p
}.
```

We could not generate the LTSA representation of the main compensation process with our tool LTSA because we have 101 states but LTSA tool only support up to 72 states.

4.2.5 Final Compositions

```
BUYER= (pro.req.to.mp->rcv.prc.list->price.req.to.mp->ack->conf.ordr
      ->pro.conf.mp->pay.trans->ack.tr->END).
MP= (rcvpro_req->pque_slr->rcv_prlist->fwd_prlist_buyer
      ->rcv_prcreq_buyer->fwd_prcreq_slr->END).
SEND= (send_ack_buyer->rcv_ordr_conf_buyer->send_ack_slr
      ->rcv_pro_conf->send.pro.conf.bur->rcv_payment->fwd_payment_slr
      ->END).
SELLER= (rcv_pro_query->req_pro_qt->rcv.info->send_pro_list
      ->rcv_prc_req->rcv_order_conf->send_or_conf_sup->rcv.pro.avail
      ->send_pro_conf->rcv_pro->rcvpayment.frm.mp->send_for_pckg
      ->END).
SUPPLIER= (rcv.pro.qt->send.info.slr->rcv.pro.conf->pro.avail
      ->trans.pro->rcv.ack.slr->END).
TRANSACTION= (rcv.payment.buyer->paymentfwd.to.mp->ack->END).

||N=(BUYER||MP||SELLER||SUPPLIER||TRANSACTION)/

{pro.req.to.mp/rcvpro_req,pque_slr/rcv_pro_query,rcv.prc.list/fwd_prli
st_buyer,
price.req.to.mp/rcv_prcreq_buyer,fwd_prcreq_slr/rcv_prc_req,ack/send_a
ck_buyer,rcv.payment.buyer/pay.trans,
send.pro.conf.bur/pro.conf.mp,paymentfwd.to.mp/rcv_payment,rcvpayment.
frm.mp/fwd_payment_slr,req_pro_qt/rcv.pro.qt,
send.info.slr/rcv.info,send_or_conf_sup/rcv.pro.conf,rcv_order_conf/se
nd_ack_slr,pro.avail/rcv.pro.avail,
send_pro_list/rcv_prlist
}.
```

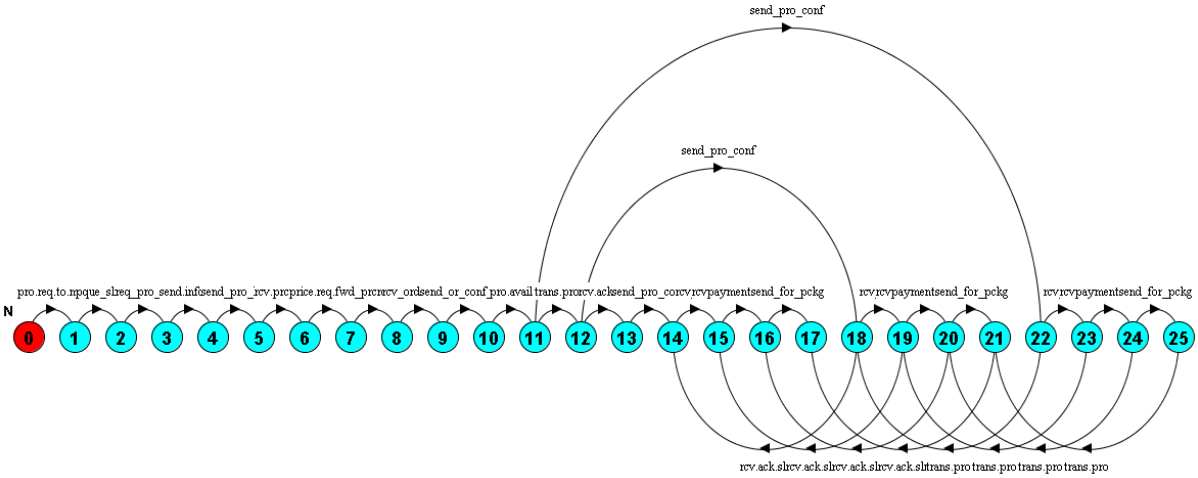

Figure 4.12: LTSA representation of Main Process

MAIN Process is the parallel composition of all engaged processes to the system with all safety properties (that will be discussed in Chap. 5). All major services composed in MAIN. All the services have been synchronized with each other through the relabeling. All the compensation states are combined in MAIN COMPENSATION (B).

CHAPTER 5

Composition Verification

5.1 Property Processes for Verification

When a safety property is executed parallel y with a process and no trace violation is generated after the execution, we can tell that the safety property verifies the process. If any trace violation is generated we will understand that the safety property could not verify the process.

5.2 Property Processes to Verify Compensation

5.2.1 Verifying Buyer Compensation

```
property SAFE_BYR = (nak->cancel_order->SAFE_BYR).
```

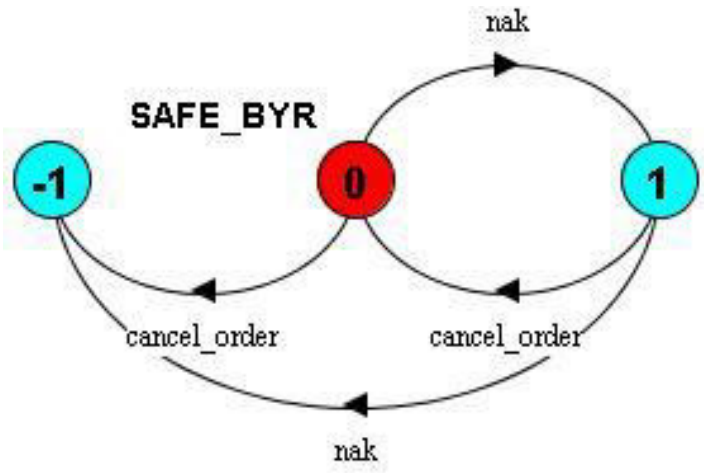

Figure 5.1: LTSA representation of safety property SAFE_BYR

The property SAFE_BYR consists of two actions nak and cancel_order. This property ensures that when BUYER throws a negative acknowledgement, it synchronizes with the compensation process of BUYER. From the two actions nak and cancel_order, nak is the negative acknowledgement and cancel_order is the action which indicates the cancellation of the order after throwing the nak.

When this property process is executed in parallel with Buyer process in BYRSAFE, the two actions of the property process should be found in sequential manner, and should not show any trace violations in the resulting LTS. If so then we can say that they are synchronized with each other successfully and satisfied the condition of our property process, otherwise not.

```
||BYRSAFE= (BUYER||SAFE_BYR).
```

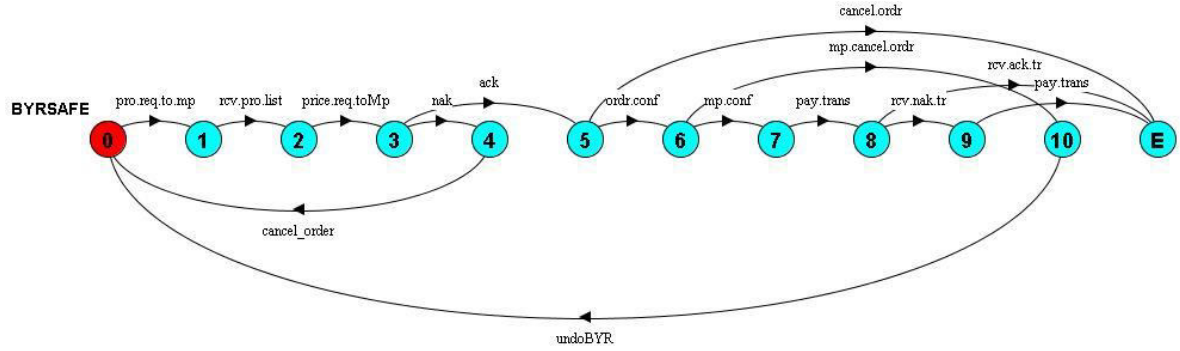

Figure 5.2: LTSA representation of BYRSAFE process

If property process doesn't synchronize with the BUYER process successfully, the LTSA representation of BYRSAFE process would be following –

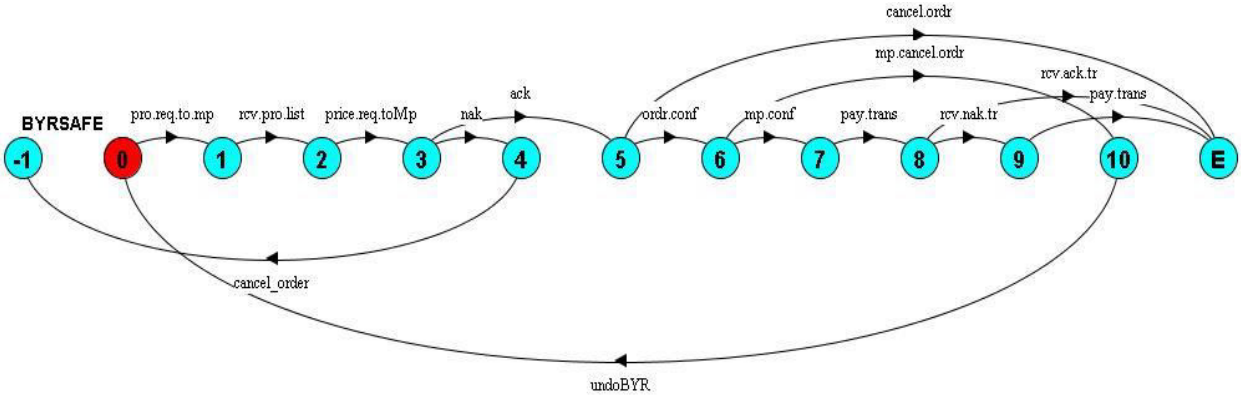

Figure 5.3: LTSA representation of BYRSAFE process with Invalid State

5.2.2 Verifying Seller Compensation

```
property SAFE_SLR=(send.nak->cancel->SAFE_SLR).
```

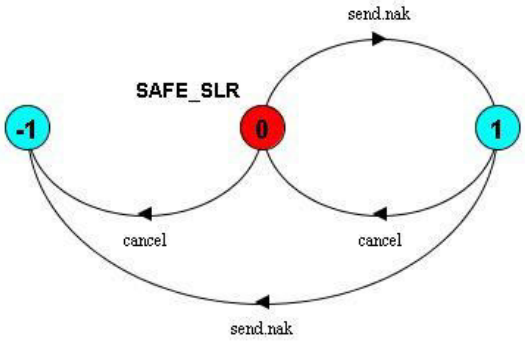

Figure 5.4: LTSA representation of safety property SAFE_SLR

Unlike the property process SAFE_BYR, SAFE_SLR is described with two actions, send.nak and cancel. send.nak is the negative acknowledgement and cancel is the action which indicates that the order is cancelled after throwing the send.nak.

When this property process is executed in parallel with SELLER process in SLRSAFE, the two actions of the property process should be found in sequential manner, and should not show any trace violations in the resulting LTS. If so then we can say that they are synchronized with each other successfully and satisfied the condition of our property process, otherwise not.

```
||SLRSAFE= (SELLER||SAFE_SLR).
```

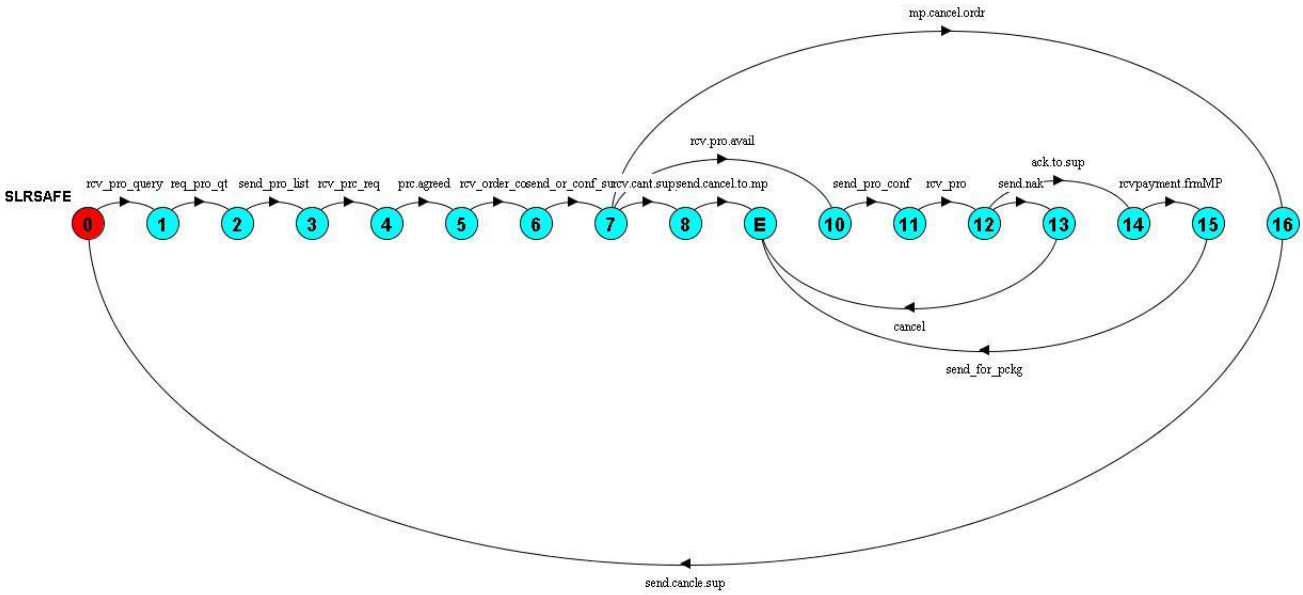

Figure 5.5: LTSA representation of SLRSAFE process

If property process doesn't synchronize with the SELLER process successfully, the LTSA representation of SLRSAFE process would be following –

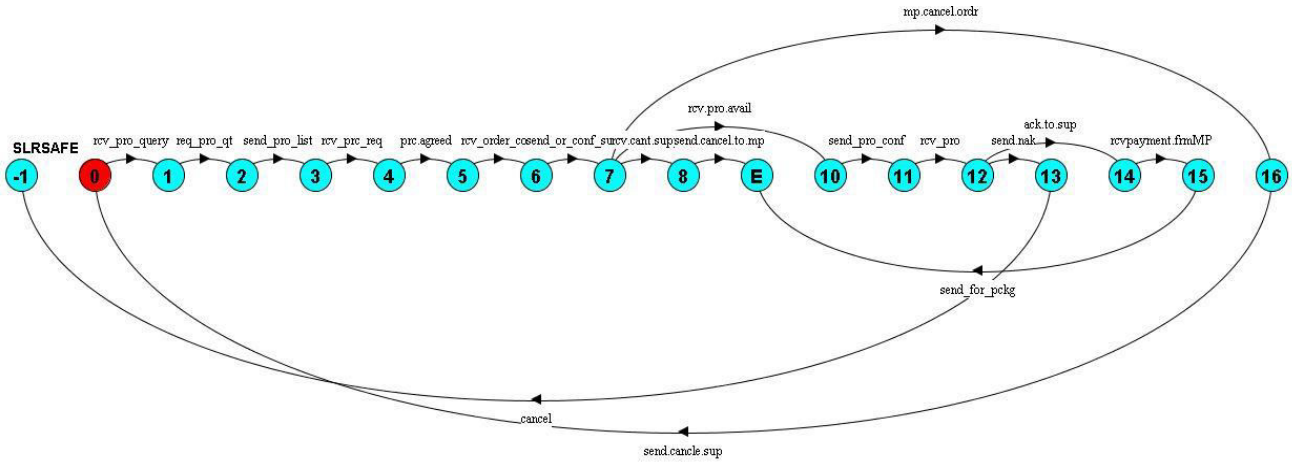

Figure 5.6: LTSA representation of SLRSAFE process with Invalid State

5.2.3 Verifying Marketplace Compensation

```
Property SAFE_MP=(rcv.nak.frm.slr->send_nak_buyer->thrws->SAFE_MP).
```

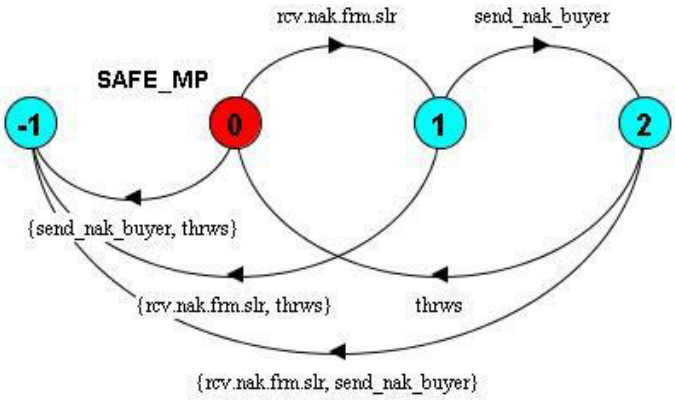

Figure 5.7: LTSA representation of safety property SAFE_MP

The SAFE_MP property is used to ensure that when the negative acknowledgement is received from the process SELLER using rcv.nak.frm.slr, it is transferred to the BUYER by using send_nak_buyer.

By composing the property process with MP and observing the resulting LTS, if there is no trace violation, we can conclude that they have been synchronized with each other successfully and satisfied the condition of our property process, otherwise not.

```
||MPSAFE= (MP||SAFE_MP).
```

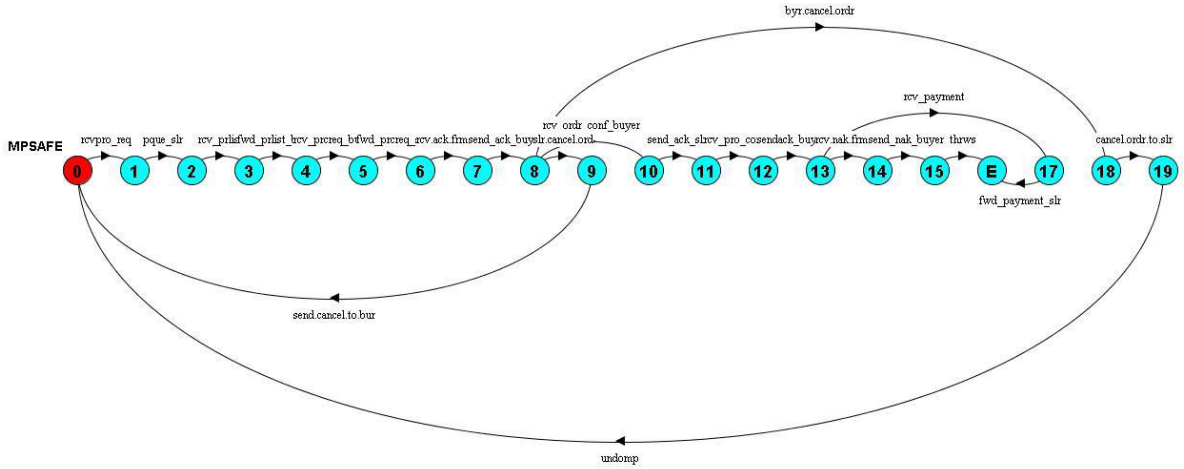

Figure 5.8: LTSA representation of MPSAFE process

If property process doesn't synchronize with the MP process successfully, the LTSA representation of MPSAFE process would be following –

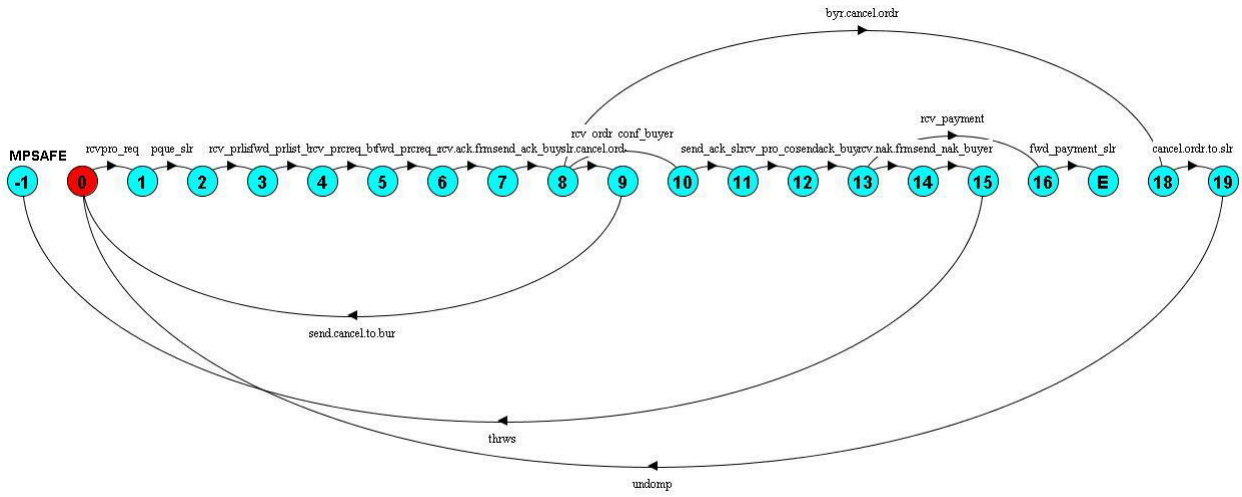

Figure 5.9: LTSA representation of MPSAFE process with Invalid State

5.2.4 Verifying Supplier Compensation

```
property SAFE_SUP=(cant.supply.to.slr->cancel->SAFE_SUP).
```

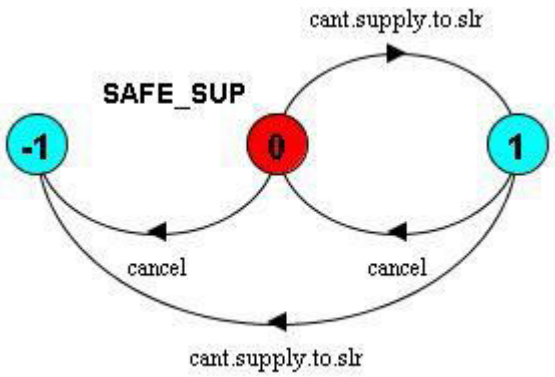

Figure 5.10: LTSA representation of safety property SAFE_SUP

The property SAFE_SUP is used for the negative acknowledgement which is sent to the SELLER if the requested product is not available using cant.supply.to.slr.

After composing the property process with SUPPLIER, if there is no trace violation, we can say that they have been synchronized with each other and satisfied property processes' condition, otherwise not.

```
||SUPSAFE= (SUPPLIER||SAFE_SUP).
```

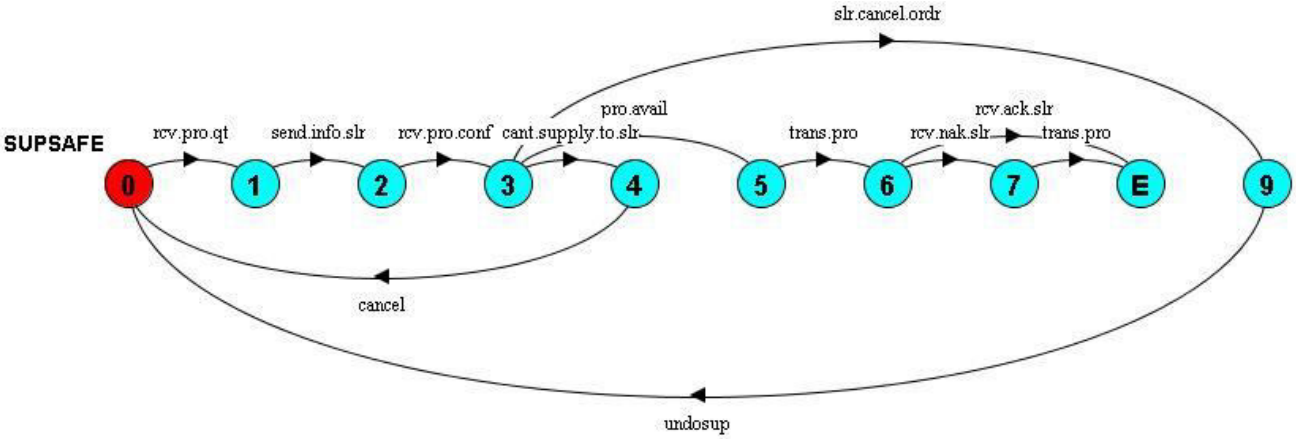

Figure 5.11: LTSA representation of safety property SUPSAFE process

If property process doesn't synchronize with the SUPPLIER process successfully, the LTSA representation of SUPSAFE process would be following –

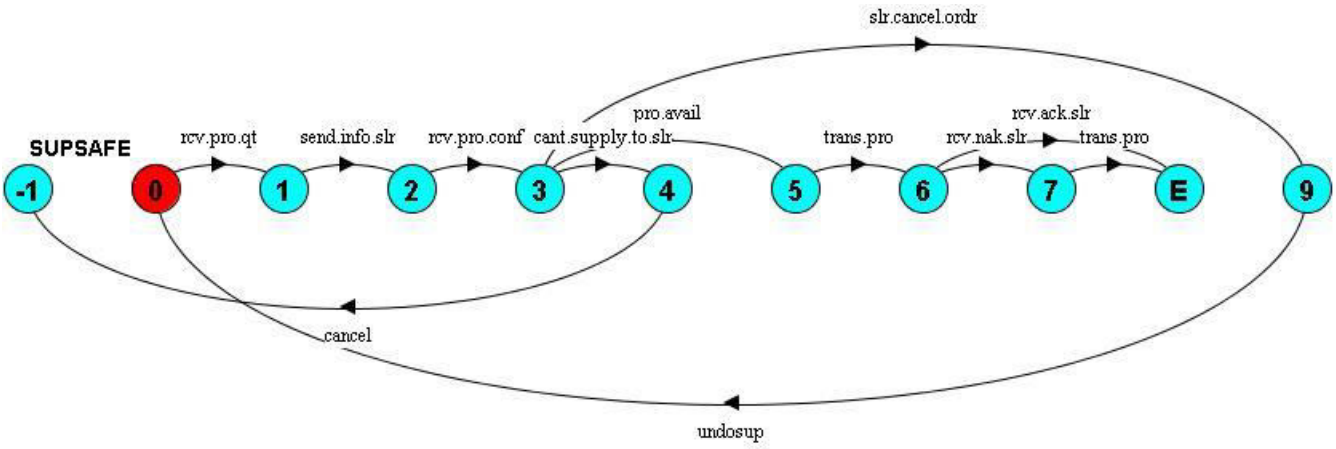

Figure 5.12: LTSA representation of SUPSAFE process with Invalid State

5.2.5 Verifying Transaction Compensation

```
property SAFE_TRANS=(nak->reverse->SAFE_TRANS).
```

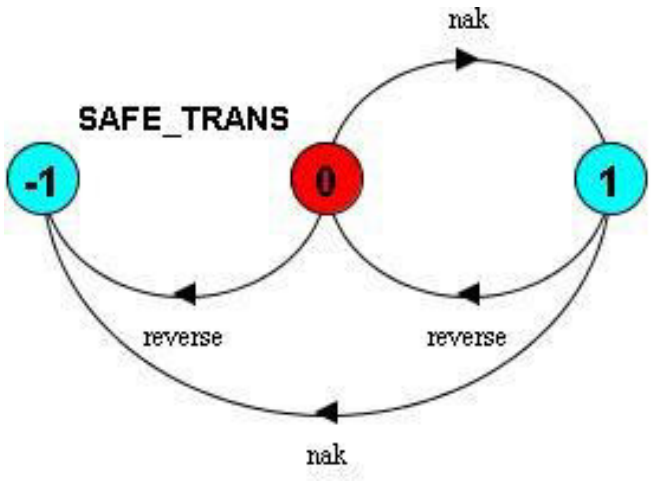

Figure 5.13: LTSA representation of safety property SAFE_TRANS

The property SAFE_TRANS consists of two actions, nak and reverse. These properties ensure that when TRANSACTION throws a negative acknowledgement, it synchronizes with the compensation process of TRANSACTION. When this property process is executed in parallel with TRANSACTION process in TRANSSAFE, the two actions of the property process should be found in sequential manner, and should not show any trace violence in the

resulting LTS. . If so then we can say that they are synchronized with each other successfully and satisfied the condition of our property process, otherwise not.

```
||TRANSSAFE= (TRANSACTION||SAFE_TRANS).
```

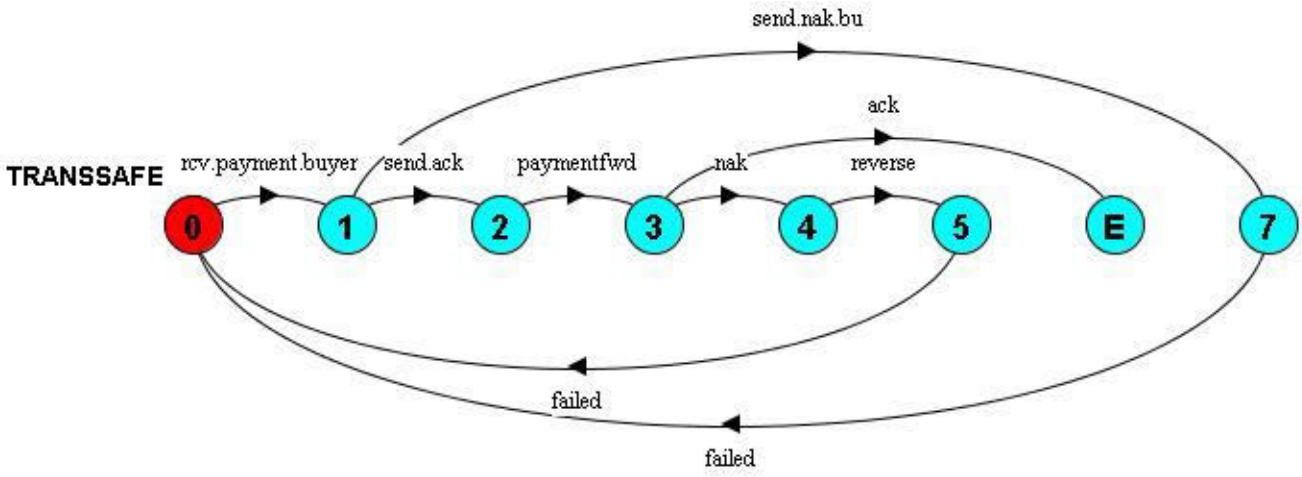

Figure 5.14: LTSA representation of safety property TRANSSAFE process

If property process doesn't synchronize with the SUPPLIER process successfully, the LTSA representation of SUPSAFE process would be following –

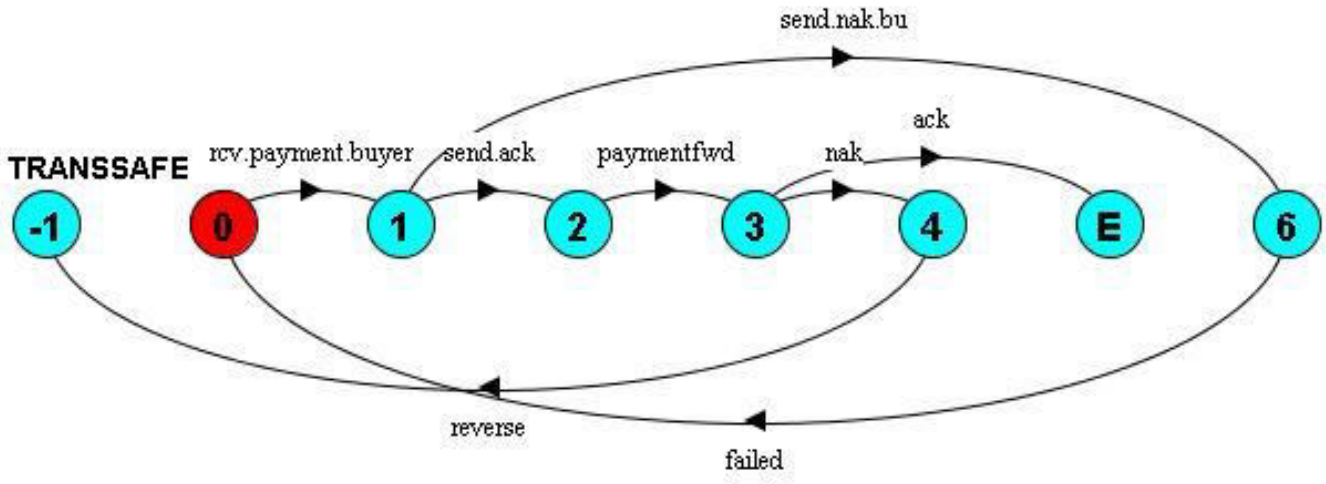

Figure 5.15: LTSA representation of TRANSSAFE process with Invalid State

CHAPTER 6

Conclusion

6.1 Summary

We have analyzed about web services and its composition. We have modeled the ONLINE MARKETPLACE Web Service by composing several web services to create a composite web service in a choreographic manner. In order to deal with transaction errors we included compensation mechanism in our system. We have used FSP notations to model and represented it with LTSA. We have also verified our desired system by using property process.

6.2 Future Work

Our future plan is to add some other property processes in the system and observing the impacts on our verification mechanism. We also want to model and verify service orchestration with another system including compensation in a similar manner that we have followed in this project.

Appendix A

A.1 Buyer Web Service

```
BUYER= (pro.req.to.mp->rcv.prc.list->prc.req.to.mp->(ack
      ->(cancel.ordr->END|ordr.conf->(mp.cancel.ordr->undoBYR
      ->BUYER|mp.conf->pay.trans->(rcv.ack.tr->END|rcv.nak.tr
      ->pay.trans->END)))|
      nak->BUYER)).
```

A.2 Marketplace Web Service

```
MP= (rcvpro_req->pque_slr->rcv_prlist->fwd_prlist_buyer
      ->rcv_prcreq_buyer->fwd_prcreq_slr->(rcv.ack.frm.slr
      ->send_ack_buyer->(byr.cancel.ordr->cancel.ordr.to.slr
      ->undomp->MP|
      rcv_ordr_conf_buyer->send_ack_slr->rcv_pro_conf
      ->sendack_buyer->(rcv_payment->fwd_payment_slr
      ->END|rcv.nak.frm.slr->send_nak_buyer->thrws
      ->END)|slr.cancel.ord->send.cencel.to.bur->undomp
      ->MP)|price.nak->price.nak.bur->MP)).
```

A.3 Seller Web Service

```
SELLER= (rcv_pro_query->req_pro_qt->send_pro_list->rcv_prc_req
      ->(prc.agreed->rcv_order_conf->send_or_conf_sup
```

```
      ->(mp.cancel.ordr->send.canle.sup->undoslr
      ->SELLER|rcv.pro.avail->send_pro_conf
      ->rcv_pro->(ack.to.sup->rcvpayment.frmMP->send_for_pckg
      ->END|rcv.cant.supp->send.cancel.to.mp->cancel
      ->END))|send.nak->END)).
```

A.4 Supplier Web Service

```
SUPPLIER= (rcv.pro.qt->send.info.slr->rcv.pro.conf->(slr.cancel.ordr
      ->undosup->SUPPLIER|pro.avail->trans.pro->(rcv.ack.slr
      ->END|rcv.nak.slr->trans.pro->END)|cant.supply.to.slr
      ->SUPPLIER)).
```

A.5 Transaction Web Service

```
TRANSACTION= (rcv.payment.buyer->(send.nak.bu->failed
      ->TRANSACTION|send.ack->paymentfwd->(ack->END|nak->reverse
      ->failed->TRANSACTION))).
```

A.6 Main Process

```
BUYER= (pro.req.to.mp->rcv.prc.list->prc.req.to.mp->(ack
      ->(cancel.ordr->END|ordr.conf->(mp.conf->pay.trans
      ->(rcv.ack.tr->END|rcv.nak.tr->undotr->END))|mp.canc.ordr
      ->END)|nak->END)).
```

```
MP= (rcvpro_req->pque_slr->rcv_prlist->fwd_prlist_buyer
      ->rcv_prcreq_buyer->(rcv.ack.frm.slr->send_ack_buyer
      ->(byr.cancel.ordr->cancel.ordr.to.slr->END|
      rcv_ordr_conf_buyer->send_ack_slr->rcv_pro_conf
      ->sendack_buyer->(rcv.payment->fwd_payment_slr
      ->END|rcv.nak.frm.slr->send_nak_buyer->END)|slr.cancel.ord
      ->send.cencel.to.bur->undomp->MP)|prc.nak->fwd.prc.nak->END)).

SELLER= (rcv_pro_query->req_pro_qt->send_pro_list->(prc.agreed
      ->rcv_order_conf->send_or_conf_sup->(mp.cancel.ordr
      ->send.canle.sup->END|rcv.pro.avail->send_pro_conf
      ->rcv_pro->(ack.to.sup->rcvpayment.frm.mp->send_for_pckg
      ->END|rcv.cant.supp->send.cancel.to.mp->cancel
      ->END))|disagreed->END)).

SUPPLIER= (rcv.pro.qt->send.info.slr->rcv.pro.conf
      ->(slr.cancel.ordr->END|pro.avail->trans.pro->(rcv.ack.slr
      ->END|rcv.nak.slr->trans.pro->END)|cant.supply.to.slr->END)).


TRANSACTION= (rcv.payment.buyer->(send.nak.bu->failed
      ->TRANSACTION|send.ack->paymentfwd.to.mp->(ack.mp->END|nak.mp
      ->reverse->failed->END))).

||B= (BUYER||MP)/
{pro.req.to.mp/rcvpro_req,rcv.prc.list/fwd_prlist_buyer,prc.req.to.m
p/rcv_prcreq_buyer,pque_slr/rcv_pro_query,prc.agreed/rcv.ack.frm.slr
,send.nak/price.nak,send.cencel.to.bur/mp.canc.ordr,nak/fwd.prc.nak,
prc.nak/disagreed,prc.req.to.mp/rcv_prcreq_buyer,rcv.prc.req/fwd_prc
```

```
req_slr,prc.agreed/rcv.ack.frm.slr,send_ack_buyer/ack,cancel.ordr/by
r.cancel.ordr,cancel.ordr.to.slr/mp.cancel.ordr,cant.supply.to.slr/m
p.send.nak,mp.send.nak/send_nak_buyer,pro.avail/mp.conf,pay.trans/rc
v.payment.buyer,rcv.ack.tr/send.ack,rcv.nak.tr/send.nak.bu,paymentfw
d.to.mp/rcvpayment.frm.mp,ordr.conf/rcv_ordr_conf_buyer,send_ack_slr
/rcv_order_conf,pro.avail/rcv_pro_conf,pro.avail/rcv.pro.avail,rcv_p
rc_req/fwd_prcreq_slr,paymentfwd.to.mp/rcv.payment,fwd_prlist_buyer/
rcv_pro_query,rcv.pro.qt/req_pro_qt,rcv_prlist/send_pro_list,prc.req
.to.mp/rcv_prcreq_buyer,rcvpayment.frm.mp/fwd_payment_slr,ack.to.sup
/rcv.ack.slr,rcv_pro/trans.pro,slr.cancel.ordr/send.canle.sup,slr.ca
ncel.ordr/mp.cancel.ordr,rcv_prlist/send_pro_list,cant.supply.to.slr
/rcv.cant.supp
}.
```

A.7 Property Process

Buyer Property Process

```
BUYER= (pro.req.to.mp->rcv.pro.list->price.req.toMp->(ack
      ->(cancel.ordr->END|ordr.conf->(mp.cancel.ordr->undoBYR
      ->BUYER|mp.conf->pay.trans->(rcv.ack.tr->END|rcv.nak.tr
      ->pay.trans->END)))|
      nak->cancel_order->BUYER)).

property SAFE_BYR = (nak->cancel_order->SAFE_BYR).

||BYRSAFE= (BUYER||SAFE_BYR).
```

Marketplace Property Process

```
MP= (rcvpro_req->pque_slr->rcv_prlist->fwd_prlist_buyer
      ->rcv_prcreq_buyer->fwd_prcreq_slr->(rcv.ack.frm.slr
      ->send_ack_buyer->(byr.cancel.ordr->cancel.ordr.to.slr
      ->undomp->MP|rcv_ordr_conf_buyer->send_ack_slr->rcv_pro_conf
      ->sendack_buyer->(rcv_payment->fwd_payment_slr
      ->END|rcv.nak.frm.slr->send_nak_buyer->thrws
      ->END)|slr.cancel.ord->send.cancel.to.bur->MP))).

property SAFE_MP=(rcv.nak.frm.slr->send_nak_buyer->thrws->SAFE_MP).
||MPSAFE= (MP||SAFE_MP).
```

Seller Property Process

```
SELLER= (rcv_pro_query->req_pro_qt->send_pro_list->rcv_prc_req
      ->(prc.agreed->rcv_order_conf->send_or_conf_sup
      ->(mp.cancel.ordr->send.cancle.sup->SELLER|rcv.pro.avail
      ->send_pro_conf->rcv_pro->(ack.to.sup->rcvpayment.frmMP
      ->send_for_pckg->END|send.nak->cancel->END)|
      rcv.cant.supp->send.cancel.to.mp->END))).

property SAFE_SLR=(send.nak->cancel->SAFE_SLR).
||SLRSAFE= (SELLER||SAFE_SLR).
```

Supplier Property Process

```
SUPPLIER= (rcv.pro.qt->send.info.slr->rcv.pro.conf
      ->(slr.cancel.ordr->undosup->SUPPLIER|pro.avail->trans.pro
```

```
      ->(rcv.ack.slr->END|rcv.nak.slr->trans.pro
      ->END)|cant.supply.to.slr->cancel->SUPPLIER)).

property SAFE_SUP=(cant.supply.to.slr->cancel->SAFE_SUP).
||SUPSAFE= (SUPPLIER||SAFE_SUP).
```

Transaction Property Process

```
TRANSACTION= (rcv.payment.buyer->(send.nak.bu->failed
      ->TRANSACTION|send.ack->paymentfwd->(ack->END|nak->reverse
      ->failed->TRANSACTION))).

property SAFE_TRANS=(nak->reverse->SAFE_TRANS).
||TRANSSAFE=(TRANSACTION||SAFE_TRANS).
```

References

[1] Florian Daniel, Barbara Pernici, *Politecnico deo Milano, Italy,* "Web Service Orchestration and Choreography: Enabling Business Processes on the Web", Chapter XII, January 2006.

[2] Shamim Ripon, Mohammad Salah Uddin and Aoyan Barua, "Web Service composition – BPEL vs cCSP Process Algebra", *Department of Computer Science and Engineering, East West University, Dhaka, Bangladesh.*

[3] Abdaladhem Albreshne, Patrik Fuhrer, Jacques Pasquier, "Web Services Orchestration and Composition Case Study of Web services Composition", September 2009.

[4] B. Margolis with J. Sharpe, "Based on SOA for Business Developer, Concepts, BPEL, and SCA", McPress, Lewisville, TX, 2007.

[5] Howard Foster, Sebastian Uchitel, Jeff Magee, Jeff Kramer, "Model-based Verification of Web Service Compositions", *Department of Computing, Imperial College London*.

[6] Jeff Magee, Jeff Kramer, "Concurrency: State Models and Java Programs", Text Book, 2nd Edition, John Wiley & Sons, Ltd, 2006.

[7] Mark Austin, John Johnson, "Compositional Behavior Modeling and Formal Validation of Canal System Operations with Finite State Automata", ISR Technical report 2011-04, The Institute for Systems Research.

[8] Shamim H. Ripon, *Department of Computing Science, University of Glasgow, UK*, Michael Butler, *School of Electronics and Computer Science, University of Southampton, UK*, "Formalizing cCSP Synchronous Semantics in PVS".

[9] Shamim H. Ripon, *Department of Computing Science, University of Glasgow, UK*, "Process Algebraic Support for Web Service Composition".

[10] Shamim H. Ripon, Farhana Sultana and Fahmida Rahman, "Verification Of Web Service Composition And Compensation By Using FSP", Department *of Computer Science and Engineering, East West University, Dhaka, Bangladesh.*