AN AUTOMATED METHODOLOGY FOR NEGOTIATION BEHAVIORS IN MULTI-AGENT ENGINEERING APPLICATIONS

by

Moath Jarrah

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2008

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Moath Jarrah
entitled An Automated Model for Negotiation Behaviors in Multi-Agent Engineering Applications
and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy in Electrical and Computer Engineering.

_____Date: 06/26/08
Bernard P. Zeigler

_____Date: 06/26/08
Roman Lysecky

_____Date: 06/26/08
Jonathan Sprinkle

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.
I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

_____Date: 06/26/08
Dissertation Director: Bernard P. Zeigler

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

SIGNED: Moath Jarrah

ACKNOWLEDGEMENTS

I would like to express my greatest appreciation to my advisor Dr. Bernard P. Zeigler, who guided me through this research work and introduced me to many exciting areas in discrete event simulation and its application. His help and support are endless. He made me gain new knowledge and insights in my career, without him I would never reach to this point. I am very grateful for his dedication and advising.

I would like to thank Dr. Roman Lysecky and Dr. Jonathan Sprinkle for serving in my defense committee.

I would like to thank all the members of ACIMS Lab, especially Chungman Seo, Taekyu Kim and Ho Jun Lee for the useful discussion and advice.

Special thanks go to my mother and father who never let me down.

Finally, I would like to express my appreciation to my wife, Pi-Tsung and my son, Malik who never stopped supporting me and encouraging me in all aspects of my life. They never made me give up on anything. This achievement is dedicated for them.

TABLE OF CONTENTS

TABLE OF CONTENTS - *Continued*

TABLE OF CONTENTS - *Continued*

TABLE OF CONTENTS - *Continued*

LIST OF FIGURES

LIST OF FIGURES - *Continued*

LIST OF FIGURES - *Continued*

LIST OF TABLES

ABSTRACT

The ability to manage and exploit geographically distributed systems of service providers is rather limited in today engineering solutions. Existing techniques suffer from three main problems: first, current techniques cannot provide brokering in managing loosely coupled service providers. Second, the engineering design of existing management tools does not provide enough expressive capabilities for varying user behaviors or when different domains are encountered. Third, lack of interaction between different requestors and providers yields inefficient and very costly agreements. In this dissertation, we will present an automated Domain-Independent Marketplace architecture that allows user agents to interact with provider agents using two simple and yet powerful negotiation protocols which define the rules of interactions in multi-agent environments. Having a trusted third party marketplace supports privacy and transparency among collaborative agents and service providers. Service providers have different capabilities depending on the domain of interest. Such providers can be radar sensors as in oceanography surveillance systems, print servers in distributed printing jobs community, or they can be online stores providing products on the web in the E-commerce domain. In order to provide negotiation in different domains, a dynamic message structuring capability is needed. A key role to support such an expressive power is to design an ontology that contains specialization relations between the different domains of interest. The automation of integrating the Domain-Dependent message structure Ontology with the Domain-Independent marketplace architecture gives the designer a powerful tool in which systems can be tailored based on the operational purposes and objectives.

The System Entity Structure (SES) methodology, which is a formalism to define hierarchical relations among entities, is used to build the required message structures Ontology automatically through the creation of SES natural language. The architecture design of the Marketplace suggests different phases and functionalities which are mapped and implemented using the Discrete Event System Specifications (DEVS). DEVS/Service Oriented Architecture (DEVS/SOA) is used to validate our system and show a proof of the concept by deploying models of printing jobs in a web-services multi-server environment for printing server domain.

CHAPTER 1. INTRODUCTION

1.1 Goals

Our goals of this dissertation consist of:

- To develop and automate a modeling methodology that supports negotiation capabilities and services to capture different user interaction behaviors in different application domains. The emphasis is on automating domain-specific tailoring of messages so as to provide a framework for detailed specification of negotiation protocols.
- To provide proof of concept implementation of this methodology in a web services environment.
- Note: the goals do not include providing analytical proofs of behavioral properties of the negotiation process. In particular, although termination is a critical issue, the framework developed must relegate its resolution to the designer who supplies the necessary behavioral specifications.

1.2 Motivation

The ability to reserve and utilize software and/or hardware services in current complex geographically distributed system has become increasingly difficult. The

complexity results from the fact that there are many aspects and factors that represent the characteristics of these systems, such as a node bandwidth, job processing deadline, the execution time, etc. The user's decision of whether to use a computing service or not is based on these factors. Many researchers and other parties have tried to provide solutions to exploit these resources efficiently [76] [77]. However, until now the development of methods to exploit geographically distributed information storages and computing resources has been very limited. Existing techniques [21] suffer from three main issues: first, current techniques cannot provide brokering in managing loosely coupled service providers. Second, the engineering design of existing management tools does not provide enough expressive capabilities for varying user behaviors or when different domains are encountered. Third, lack of interaction between different requestors and providers yields inefficient and very costly agreements. Also one main issue in collaborative distributed multi-agent environments is providing privacy and transparency to their agents. More on multi-agent design issues and challenges can be found in [47].

Distributed environments are seldom static. Everyday more and more service providers are added to the system in order to provide more capabilities as the users grow in numbers and needs. This leads to the diversity in resources and data availability which adds new challenges to the management techniques that systems use. Hence, a manual management is not feasible in such a community because of the number of service providers and the heterogeneity in their information management. All of the above issues make discovering the "Best Match" for satisfying user requirements a tedious task.

Web Services developments are growing dramatically nowadays and millions of resources are being added every day to the World Wide Web. The success in e-commerce, e-learning, online auctions, online marketplaces, information discovery and retrieval has encouraged more and more companies to provide Web Services either to satisfy customer requirements or to manage their distributed computing resources. In order to reach to a successful framework design, the following issues must be supported:

- The system should provide brokering and negotiation services to its users.
- The system should provide transparency to its users.
- New service providers should be able to join the community in a simple and efficient way.
- The system should provide decision making capabilities on behalf of the agents whenever the user agents need it.
- The system should provide varying negotiation capabilities under different domains.
- The system must provide rich expressive negotiation primitives to its users to provide them with the capabilities to express their requirements and to able to use the system under different domains.
- The design of the system must be simple and automated to shorten the development time on the system designer under a specific domain of interest.

1.3 Our Approach

In this dissertation, we will develop an automated negotiation model that can be utilized by different engineering domains. The model defines different concepts and principles in the negotiation process. Our method consists of an automated Domain-Independent Marketplace architecture that allows user agents to interact with provider agents using two simple and yet powerful negotiation protocols which define the rules of interactions in multi-agent environments. Having a trusted third party marketplace supports privacy and transparency among collaborative agents and service providers. Service providers have different capabilities depending on the domain of interest. Such providers can be Radar sensors as in oceanography surveillance systems, print servers in distributed printing jobs community, or they can be online stores providing products on the Web in the E-commerce domain. In order to provide negotiation in different domains, a dynamic message structuring capability is needed. A key role to support such an expressive power is to design an Ontology that contains specialization relations between the different domains of interest. The automation of integrating the Domain-Dependent message structure Ontology with the Domain-Independent marketplace architecture gives the designer a powerful tool where systems can be tailored based on the operational purposes and objectives.

The System Entity Structure (SES) methodology, which is a formalism to define hierarchical relations between entities, is used to build the required message structures Ontology automatically through creating SES natural language. The architecture design

of the Marketplace suggests different phases and functionalities which are mapped and implemented using the Discrete Event System Specifications (DEVS). DEVS/Service Oriented Architecture (DEVS/SOA) is used to validate our system and show a proof of the concept by deploying models of printing jobs in a web-services multi-server environment for printing server domain.

1.4 Organization of the Dissertation

Chapter 2 gives a background on current negotiation systems and research and the development of autonomous agents for decision making process. Also it provides a discussion on discrete event modeling and simulation inside DEVS formalism. System Entity Structure is introduced as an ontological framework for data engineering purposes. Chapter 3 discusses the ontology design motivation in the Semantic Web and the different capabilities of ontology languages in the W3C recommendations. Chapter 4 details the negotiation protocols and the language of encounter of our system with a description of the Marketplace characteristics and the automation of how to select a primitive structure based on the domain of interest.

The implementation of the negotiation model in DEVS environments and ontology design in SES formalism will be given in chapter 5. In chapter 6 we automate the process of model generation to produce a tailored marketplace model for a given domain, which results in a code generation tool that shorten the development time on behalf of systems designers. Then we apply our system to different engineering

applications and show two experiments along with distributed web services deployment of the model to provide a proof of the concept in chapter 7 and chapter 8 respectively. Finally, we conclude the dissertation and mention future work that might improve the systems characteristics to support more choices and capabilities.

CHAPTER 2. BACKGROUND: NEGOTIATION PROCESS, FD-DEVS and SES

This chapter gives a review of the research areas that are relevant to our work. The first section introduces the research in negotiation systems. Section 2.2 discusses technologies in automated user agents in multi-agent environments and the decision making process. Sections 2.3 gives an overview on discrete event modeling and simulation formalism DEVS and the derived Finite Deterministic DEVS specifications. The last section ends the background discussion by introducing the System Entity Structure formalism SES.

2.1 Research in Negotiation Systems

The negotiation process is an interaction between two or more parties in an attempt to reach some agreement on a specific aspect. This aspect could be an idea as in e-learning, or a price of some goods as in e-commerce, or information availability and data provision. Hence, a multi-criterion negotiation system is needed that supports dynamic structures based on the domain of interest [64]. During the negotiation process, web-based agents exchange their capabilities, such as the services they provide, offers, counter offers, speed, bandwidth, goods, ideas, topics or computation power. The result can be an agreement or disagreement. In either case, the result depends on the interest of the agents and their achievement of profit. A negotiation agent needs to be flexible

enough to act under different kinds of situations because negotiation is a dynamic activity by nature. The process is dynamic in the sense that it involves: asking for an item or service, discovering item/service providers, negotiating with sellers/service providers, proposing counter offers, decision making upon the receiving of some offers, and then acceptance or rejection of an offer. The agent needs also to make sure that he does not go into an infinite cycle of negotiation.

Negotiation activity in multi-agent environments is an iterative behavior in which agents negotiate by exchanging Offer-CounterOffer messages. G. O'Hare and N. Jennings in their book on Distributed Artificial Intelligence [8] classified the research in negotiation into three main categories: negotiation language, negotiation decision and negotiation process. Our research interests fall into the first category. The negotiation language category consists of negotiation protocols, negotiation primitives, semantics and object structure. Protocols refer to polices or rules that agents must follow during their interactions with other agents. Primitives refer to the messages that are exchanged between the agents. Negotiation primitives (messages) can be placed into three groups: *initiators* such as "request", *reactors* such as "respond", and *completers* such as "accept". The semantics give more explanation and meaning to the *language of encounter* (primitives) that is being used in negotiation protocols. The semantics capabilities are usually achieved by building an ontology which classifies primitives based on measurements of similarity. So, for example, one can consider the two primitives "Request" and "Query" to be equivalent. Some tools are used in order to help in computing measurements of similarities such as WordNet [67], which gives synonyms

and acronyms of a given term based on the semantic meaning. This problem is well known in the natural language processing area. The most difficult challenge is the ambiguity in using sentences of a sequence of terms. In this work, the semantics are not part of our work because it is not a necessary factor for system completeness and methodology.

The object structure refers to the structure of each of the primitives during the interaction between different agents, which defines what type of information a message can carry. The negotiation decision is concerned with algorithms and mathematical models to represent how user agents evaluate their objective functions. The next section will give some insights on game theory application to this area. The last category which is the negotiation process formulates general models and global behavior of the negotiation participants.

The application of the negotiation process in current systems is very limited. One reason is that current systems lack the infrastructure that can support negotiation among parties. Also, current nodes are loosely coupled and no brokering activities are available.

Current bidding and auction systems do not provide the flexibility to negotiate on parameters chosen by the users. They consider the price as the only parameter that in which users are interested. For example, eBay [39] and Amazon Auctions [46] require from the bidders that they locate an exact item and bid on it based only on its price. The bidding is a committed action, which means that if a bidder wins, he has to buy it. This

discourages users of the system to bid on more than one item because they do not want to end up buying many items when they only need one [41].

Priceline.com [43] is an airline booking auction where a user selects his flight information (source, destination, traveling date, and returning date). And then the user bids by entering a specific price. Priceline searches its database to find a ticket price that is lower than the bidder price. If a ticket is found, then the bidder will get the ticket. This scenario of negotiation has drawbacks which are summarized as in follows:

1. If bidding is accepted, then the bidder is required to purchase the ticket.
2. The bidder cannot control other information on the flights such as waiting time in the airport, and number of stops on the way.
3. The system takes advantage of users who do not have the knowledge and experience about ticket prices. A bidder might enter a high bidding price for a cheap ticket.
4. It prevents the user from paying a little more money for a more comfortable flight.

Our objective in this research is to support negotiation capabilities over more than one dimension. We can have as many constraints as it needs. A user can choose different criteria to be considered in addition to the price; for instance, how many stops, Airline Company, period of the negotiation, and so on. The dynamic structure of the language of

encounter makes this possible and we will demonstrate later in chapter 5 and 6 how to implement that.

2.2 Negotiation in Multi-Agent Environments

In most of business and engineering distributed systems, managing the resources and services manually is impossible and autonomous agents are needed to act on behalf of the system users. Negotiation process is methodology that was applied to these systems to provide bargaining and brokering capabilities between different agents in multi-agent environments. Such agents are not just capable of making decisions in predictable situations, but also they need to be intelligent enough to act in any dynamic unpredictable interaction. The agents need to communicate with each other, share data and ontologies and negotiate with other agents to reach some agreements. Many researchers have addressed these issues and many autonomous agents were developed recently. For instance, a user can use search agents over the Web to search for a specific data or information and once the appropriate data provider is found, the data will be sent to the user.

Game theory is a branch of economics that is concerned with interactions between agents [36] [30] [44]. It imposes mathematical models (functions) that describe each agent utility function in multi-agent systems, and strategically try to maximize each individual preference. Under some domains, the mathematical function of an agent is formatted to take into account other opponents and coordinators utility functions.

However, the game theory is limited by the assumption of having the knowledge about other players (agents) preferences. Negotiation probably is one of the most frequent domains in which game theory principles have been applied. Negotiation environments use game theory in order to model the decision making process in the negotiating agents; which can give insights into the computation of the search space in order to analyze different interaction strategies.

Game theory mathematically models the interaction techniques between players along with their outcome results. It was first started with the work of von Neumann and Morgenstern [40] in 1944. Studies in game theory assume that individuals (agents) are rational and have well defined preferences over all relevant playing strategies. Hence, when an individual has to choose from alternative techniques, he will choose the most preferred strategy that maximizes his utility. This imposes difficulties in multi-agent environments where each agent tries to achieve his own interest which leads to conflicts with other individual preferences. Predictions about the resolution of conflict are derived from game-theoretic solutions that use some variants of Nash equilibrium.

Algorithms that have been developed in game theory were mostly proposed to solve or help an agent to play specific games intelligently based on the opponent choices. The agent needs to make choices that maximize or optimize his revenue in a strategic way. In other words, it must decide on his playing strategy based on the decisions that the other opponents make during playing. For example, alpha-beta pruning [68] as depicted in figure 2.2.1 uses a min-max strategy to maximize *player 1* utility against his opponent.

It is clear in this algorithm that the third issue with game theory (the knowledge of the end nodes outcomes) is hard to compute precisely ahead of time. However, it is still a helpful algorithm when approximations are assigned to the outcomes of the end nodes. Also some calculations can be made in advance.

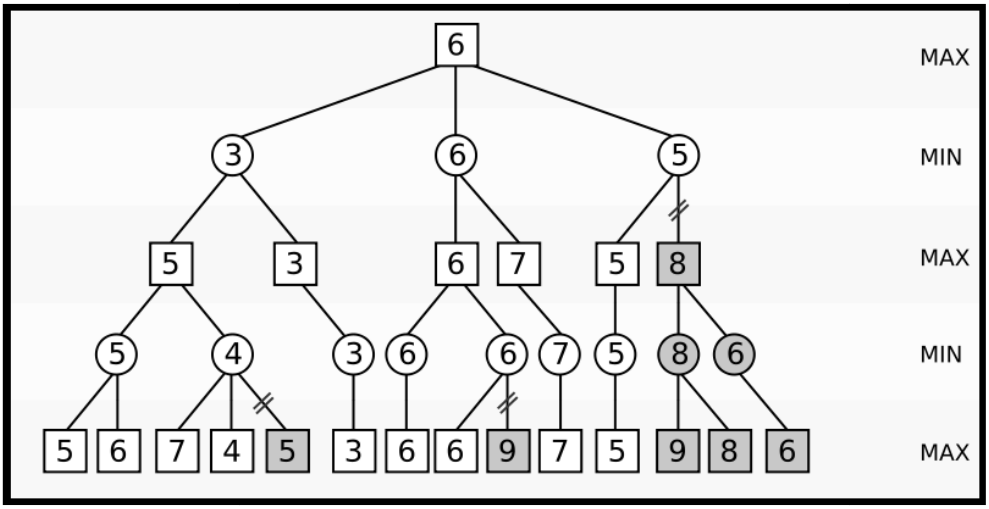


Figure 2.2.1: Alpha-Beta pruning algorithm

Martin J. Osborne and Ariel Rubinstein in [34] discussed variant sequential models in applying game theory to bargaining. The models have a sequential structure in the sense that each player makes decisions sequentially in a pre-specified order. The order reflects the rules or policies of the negotiation (negotiation protocol and rules of encounter). At all times, the negotiators care about time to protect themselves from going into an infinite cycle of offers and counter offers. So the speed at which an agreement is established is one factor that plays into defining the agent revenue or preference objective function. No agreement is forced on any agent. This means that if all agents who are involved in a specific negotiation cycle chose to accept the terms in that negotiation

cycle, then the agreement will take place, otherwise the agreement fails. Two important considerations to be taken care of in applying game theory to negotiation when it comes to computer systems are:

1. Game theory studies in multi-agent computer systems assume that agents search for the optimal solution (or strategy). This involves the computations of all search space which can grow exponentially as the number of variables increases.

2. The recent growth in the Internet and Web services raises the interest and the need for more sophisticated developments in computational negotiation techniques and autonomous web agents.

K. Binmore and N. Vulkan [37] used a simple mathematical formula to describe the decision making process. They modeled the agreement by using two real numbers a_1 and a_2. Player 1 (or buyer) and player 2 (or seller) keeps some reserved value for the item they are bargaining about [$r(1)$ and $r(2)$ respectively]. These values are kept hidden from each other (player 1 does not inform player 2 about his reserved value and visa versa). If player 1 proposes a price m (the amount of money) for the item, then $a_1 = r(1) - m$ and $a_2 = m - r(2)$. The agreement succeeds if both a_1 and a_2 are positive numbers (≥ 0). As mentioned previously in chapter 2, both agents should approve the acceptance of the agreement terms and this model guarantees that.

V. Krishna and VC. Ramesh in their work on market games and their applications used a negotiation model based on coalition partners [30] [38]. The player agent chooses a set of agents to form a coalition. Then it uses probability profiles of the chosen agents to compute the payoffs resulting from using different strategies by simulating the actual bargaining. Next, it computes the probability distribution among the whole set of agent strategies (normalized based on probability measures). Using the payoff metrics, it arranges the strategies (solutions) on a priority basis where the solution that gives the maximum payoff has the highest priority. After that, it chooses a new set of agents and repeats the same computations until the agent finds the best coalition community and chooses the best strategy for that coalition.

Negotiation also enables coordination among agents to enhance performance in multi-agent systems where all agents aim to improve the overall system performance. In this context, Mahajan, Rodrig, Wetherall and Zahorjan [35] attempted to resolve selfishness routing in multi-hop networks, where Internet Service Providers try to lower the traffic they forward (route) by either dropping packets or sending them through the closest link which results in longer paths. The system sends anonymous messages in which the sender ID is hidden. If the recipient node cheated by not forwarding the messages correctly, all the neighbors isolate the cheating node from the network. The cost here for a cheating node is that it will be punished by disconnecting it so neighbors will not forward or receive message to or from it.

In competitive negotiation, each agent tries to maximize his own utility function (maximize his satisfaction) regardless of the other agents. However, in cooperative negotiation, an agent is concerned about other agents and he needs to compromise his own preferences for the good of community satisfaction. Archibald, Hill, Johnson and Stirling [31] used a strategic-form game by evaluating the utilities of all players to reach a negotiation solution that is mutually acceptable. It does not have to be the maximum for each agent but good enough that all players are satisfied. The authors of the paper on satisfying negotiation used the Prisoner Dilemma (PD) as an illustrated example as shown in table 2.2.1. The numbers represent the payoff matrix of the players utility with 4 = best, 3 = next best, 2 = next worst and 1 = worst. In PD game, two players *P1* and *P2* either choose to compete (defect) or cooperate with the other player. If *P1* cooperates and *P2* competes, then the result is that *P1* gets 1 (worse utility) and *P2* gets 4 (best utility) and vice versa. If *P1* chooses to compete and *P2* also chooses to compete, then both get 2 (next to worst) which is the Nash equilibrium. However, if both players choose to cooperate then both get 3 (next to best) and they call this solution a “good enough” solution. Hence, in many negotiation scenarios, cooperation, compromise and even altruism brings more benefit to the players rather than using competition and defect.

PD Game	***P2***	
P1	Coordination	Competition
Coordination	(3,3)	(1,4)
Competition	(4,1)	(2,2)

Table 2.2.1: Prisoner Dilemma (PD) game

Hence, in multi-agent systems, the players (negotiators) can adopt a bottom-up approach where each player maximizes his own utility. This approach usually results in a non-optimal overall solution (group performance). Alternatively, the agents in the negotiation group can adopt a top-down approach, where the objective is to optimize the overall utility function of all players. This leads to better results in the coordination community, where agents need to compromise their utility for the group utility. In many situations, agents might refuse to announce their utility function to the public. In this regard, having a trusted third party that can manage and coordinate the cooperation between the agents is useful (marketplace). The paper [31] by Archibald to which we are referring proposes a mathematical model for the good enough solution. The approach supports cooperation, compromise and negotiation in multi-agent systems. It uses a different criterion than individual optimization and led to well-defined solutions.

2.3 Finite Deterministic-DEVS

Discrete Event systems specifications formalism (DEVS) and its concepts were introduces firstly by Professor Bernard Zeigler in 1976. Since then, the DEVS formalism has been regarded as a powerful tool in many engineering applications areas such as manufacturing [16], ecological disasters [17], computer [18], traffic [19] and command and control (SoS) [69]. *Finite Deterministic*-DEVS was first introduced as *Schedule-Controllable* DEVS in 2005 [23]. FD-DEVS motivation was to overcome the problem of ODNR (*once it dies, it never returns*) from which *Schedule-Preserved* SP-DEVS [20] was suffering. The ODNR refers to the situation when the next schedule is infinite time which prevents the simulation from returning to any of the states that have a finite time. Since FD-DEVS is based on the classical DEVS formalism concepts and relations and since we used DEVS simulation after generating the template Marketplace model using FD-DEVS tool, we will give a brief description on the modeling and simulation environment in DEVS and the hierarchical construction of atomic models.

The Discrete Event System Specifications (DEVS) formalism provides a rich environment in which any phenomena could be modeled by producing a mathematical model which in turn can be simulated under the DEVS simulation environment [4]. DEVS can model discrete event systems as well as continuous systems. Any real system (or proposed one) goes through different states or phases, receive inputs from users or from other running entities, output messages to the interconnected properties, and has

functions or algorithms that decide the transition from one state to another either concurrently with receiving inputs or after some phase period of time. Hence, a given system could be modeled as a discrete event system with some specific parameters that need to be computed by observing the system under consideration of its behaviors. Once we decided on the different parameters of the system, we can model it using DEVS formalism and then execute the simulation for performance evaluation and/or exploring possible setups of the system until we find an acceptable system behavior. Figure 2.3.1 shows a scenario of modeling and simulation of some given system A. The experimental frame refers to the conditions under which the system is being observed for its behaviors and set of outputs.

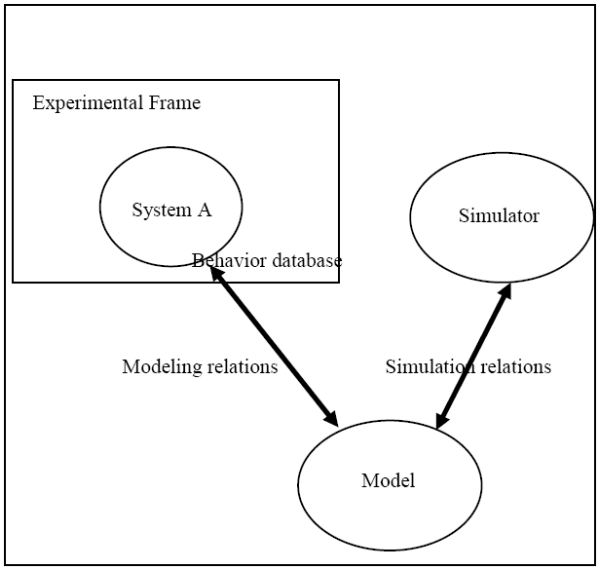


Figure 2.3.1: Different components and relations in modeling and simulation systems

The definitions of the components in the figure are as follows:

The Given System: is the system under interest which we would like to model and simulate in an attempt to monitor and alter its behavior to follow some accepted specifications.

Model**:** mathematical relations and instructions that produce similar properties as the real system under consideration. The behavior of a model is the set of all possible input/output combinations that can be generated [1].

Simulator: it executes the model in order to emulate the real system and do comparisons, evaluations and analysis.

Experimental frame**:** defines the constraints and conditions under which the system was observed to collect its output behavior. For example, a system could be running under specific temperature and pressure conditions.

These three objects in the Figure are related by two types of relations:

Modeling relations: relations to define whether the model is a valid model for the real system by comparing the behavior of the model with the behavior of the observed system. How well the model represents the system.

Simulator relations: relations to link the developed model and the simulator. The simulator carries out the instructions and specifications of the model.

The system needs to be modeled first; the model structure could be expressed in a mathematical language called *formalism*. The discrete event formalism focuses on the

changes of the variable values and generates time segments that are piecewise constant. Thus an event is a change in a variable value which occurs instantaneously. Hence, the model formalism specifies how to generate the changes in the variables values and the time at which this should occur.

Based on the research by Bernard P. Zeigler [4], a basic model from which larger ones could be built must be specified first. Basic models are connected into a hierarchal scenario. The basic model consists of the following features as illustrated in Figure 2.3.2.

- The set of input ports through which external events (messages) are received.
- The set of output ports through which external events are sent and interact with other properties.
- Two distinct parameters for each state exist which are called “phase” and “sigma”. The phase represents the current state. Sigma defines the time period during which the model stays in the corresponding phase. For example, in ON-OFF model, for the active phase, sigma = *ON T* and for inactive phase, sigma = *OFF T*.
- The time advance function which keeps the time management of the model by monitoring the clock cycles and the sigma values of all models.
- The internal transition function specifies the next state to which a model has to transit after some specified time.

- The external transition function specifies how the model should alter its behavior by changing the state given some inputs have been received that affect the current model state.
- The confluent transition function specifies the next state a model has to transit if a transition to a state occurs at the same time when an input event is received.

These three functions: internal transition function, external transition function, and the confluent transition function provides a comprehensive tools to model thoroughly all system interactions that could be possible between the components in a specific model. More about these three functions are given in the next section. The output function generates and wraps a message (packet) just before an internal transition occurs and sends it through the interconnection links between the different models.

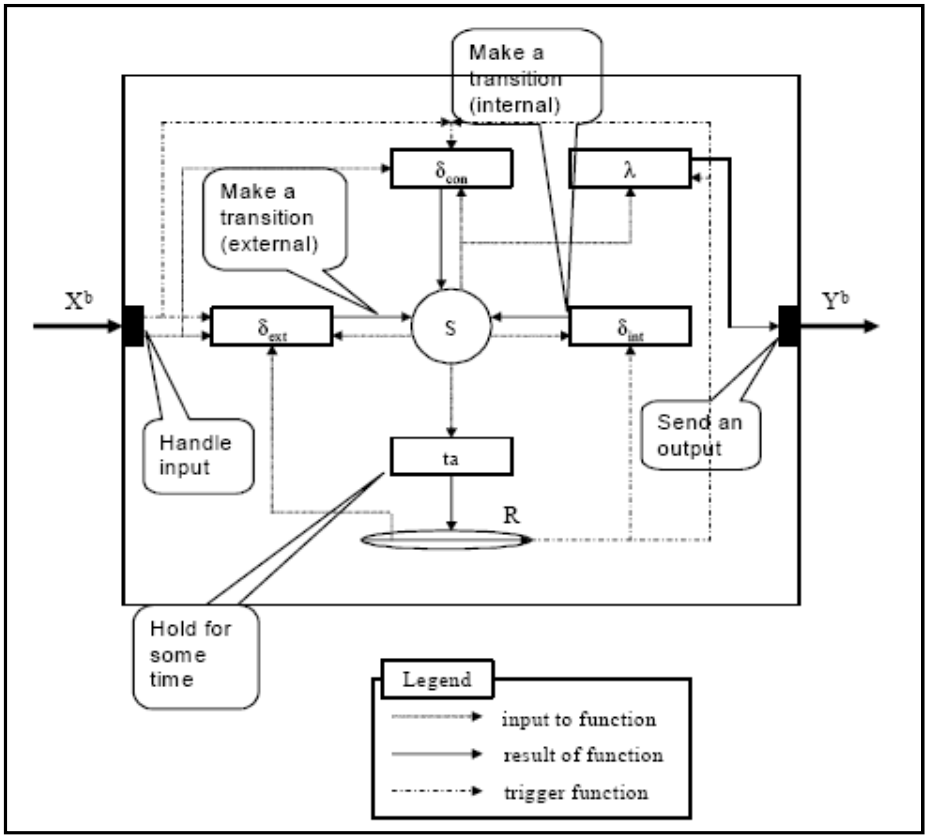


Figure 2.3.2: Internal design of the basic model

DEVS formalism also provides a mathematical model based on the set theory. The Model (*M*) structure of the above eight features is defined as follows:

$$M = < X , S , Y , \delta_{\text{int}} , \delta_{ext} , \delta_{con} , \lambda , ta >$$

Where

X is the set of input values

S is the set of states

Y is the set of output values

$\delta_{\text{int}} : S \rightarrow S$ is the internal function

$\delta_{ext} : Q \times X^b \rightarrow S$ is the external transition function, where

Q is the total state set

X^b is a collection of bags over the input set X

$\delta_{con} : Q \times X^b \rightarrow S$ is the confluent transition function

$\lambda : S \rightarrow Y^b$ is the output function

$ta : S \rightarrow \mathbf{R}$ is the time advance function.

At any time the system must be in some state, *s*. If no external event occurs, the system will stay in state *s* for time given by $ta(s)$. The advanced function $ta(s)$ can take any value between 0 and ∞. For example, if an atomic model is idle (passive state), its $ta(passive) = \infty$. When the resting time expires, elapsed time, $e = ta(s)$, the system outputs the value $\lambda(s)$ and changes to state given by $\delta_{\text{int}}(s)$. The output is only possible before an internal state transition. If an external even x in X^b happens to occur before this expiration time, the system changes to state given by $\delta_{ext}(s,e,x)$. Thus the internal transition function will be in charge of the system states transition when no external events occur. The external transition function takes over the states transitions when an event occurs on the model input ports. The confluent transition function will be triggered if an internal transition and an external event occur simultaneously.

2.3.1 Coupling in DEVS Environment

This set of mathematical variables and states defines the behavior of a specific model *M*. Such a basic model is also called an *Atomic model*. Different atomic models can be connected to produce a *Coupled model*. Coupled models can in turn be connected to other atomic models or coupled models resulting in a hierarchical structure as depicted in figure 2.3.1.1

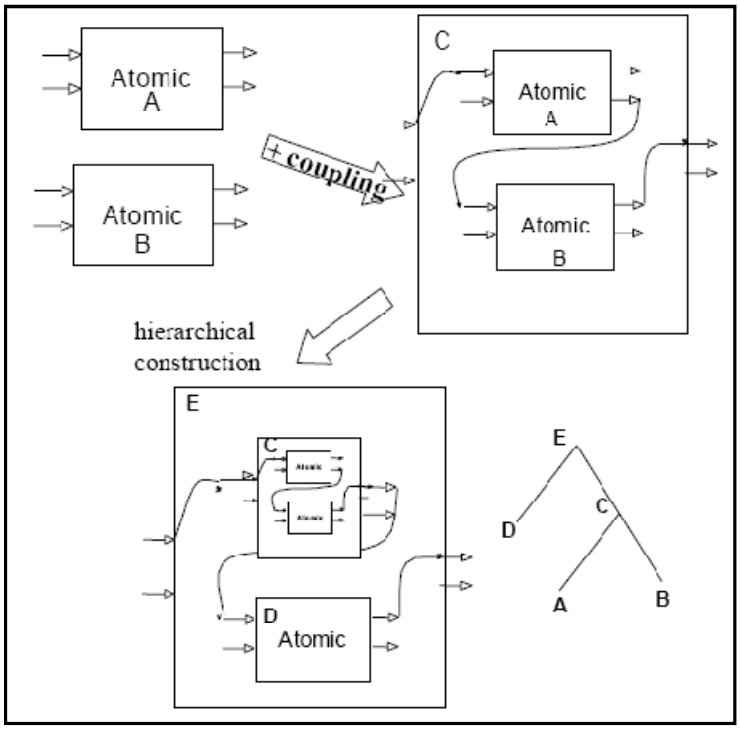


Figure 2.3.1.1: Hierarchical feature in DEVS models

A coupled model consists of the following information:

- The set of components.
- The set of input ports through which a component receives external events (messages).

- The set of output ports through which a message or event is sent.

The coupling specifications specify the routing of events between different models (or components). These specifications consist of:

- External input coupling connects input ports of a coupled model to one or more of the input ports of other components.
- External output coupling connects the output ports of a coupled model to the output ports of other components.
- Internal coupling connects output ports of components to input ports of other components. Hence, when an event is generated by a component it may be sent to the input ports of other designated components.

As illustrated in figure 2.3.1.2, when outputs (messages) are generated by component A, they are transmitted at the same time instant to the input ports of component B due to the coupling between outputs of A and inputs of B.

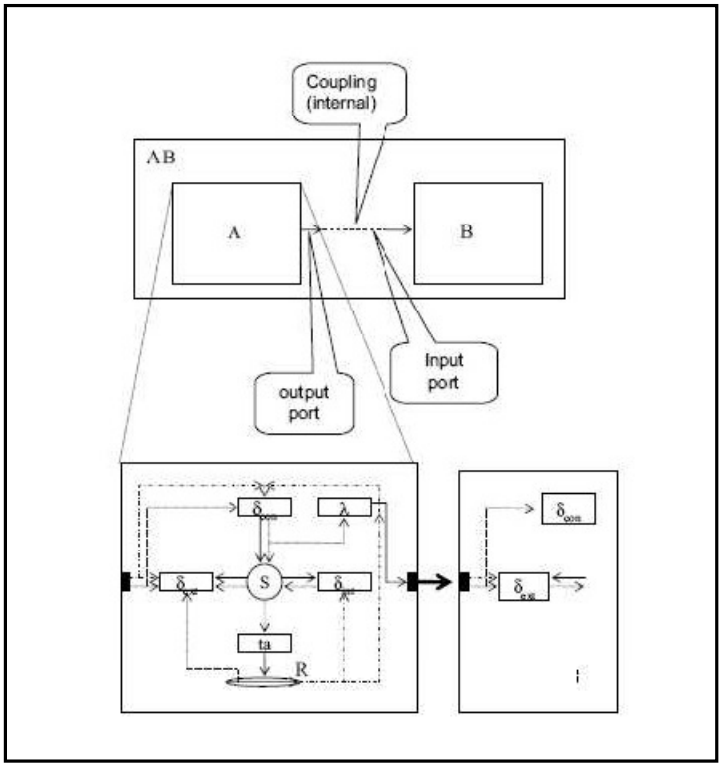


Figure 2.3.1.2: Coupled model in DEVS

2.3.2 DEVS Simulator

DEVS formalism has a well-defined concept of simulation engine to execute models and generates their behavior. The simulator has been implemented in JAVA resulting in DEVS/JAVA implementation. Afterward an implementation in C++ was needed in order to achieve an efficient runtime execution; this resulted in DEVS/C++ implementation. Figure 2.3.2.1 below shows the simulator for a coupled model which consists of:

1. Coordinator which assures and maintains the coupled model specifications.
2. Simulators associated with each one of the model components (basic models).

The coordinator performs the time management and controls the messaging exchange among the simulators consistently with the coupled specifications. Simulators respond to commands and queries from the coordinator by referencing to their assigned models specifications.

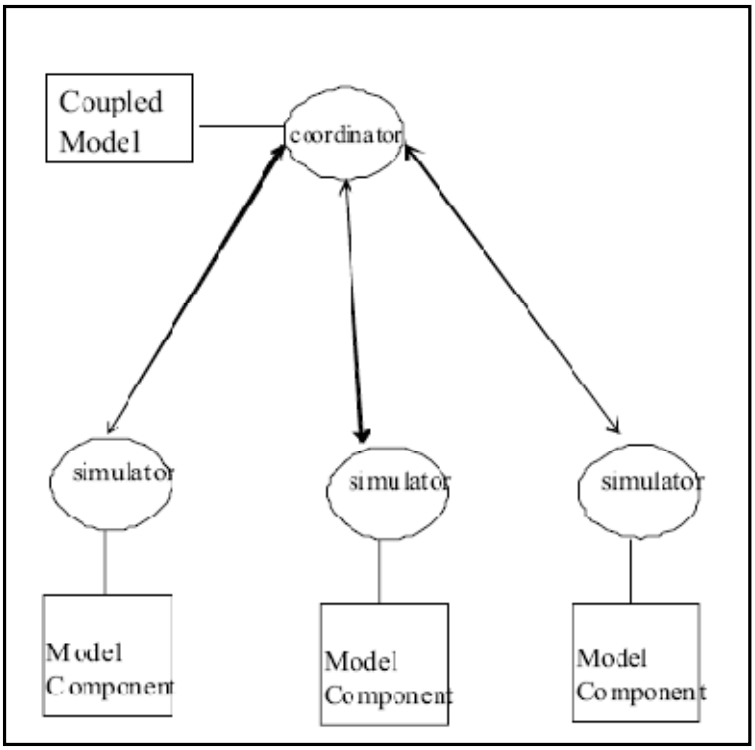


Figure 2.3.2.1: The DEVS simulation protocol

In DEVS/C++, atomic models have the following models to represent their DEVS mathematical formalism:

- Delta-internal: represents the internal transition function.
- Delta-external: represents the external transition function.
- Delta-confluent: represents the confluent transition function.
- Output: represents the output function.
- Time advance: represents the time advance function.

A model is said to be *imminent* when a certain sleep phase has been completed; such a phase is determined by the model time advance function. DEVS/C++ maintains a general-purpose template priority queue for sorting models by various keys, which could be a name, next event time or others. A tree structure is used to order the models by their next event times. The simulation cycle in DEVS/C++ simulator is as follows:

1. Advance the clock to the smallest next event time in the priority queue.
2. Put all models that have the smallest next event time in a set called **I** representing the imminent models.
3. Execute the output functions of those in the set **I** and propagate the messages to the in ports of the models that are connected directly to the output ports of the models in set **I**. These messages will be collected into an input bags for the models receiving them.
4. Put all models that have non empty input bags on their input ports into a set **M**.

5. The elements in **I** ∩ **M**, represent the models who have inputs on their input ports and have also internal transition occurring at the same time. So for those elements, execute the associated delta-confluent function.
6. The remaining elements in set **I** represent those who only have internal transition. Hence, execute the associated delta-internal function.
7. The remaining elements in **M** represent those who have external transition. Hence, execute the associated delta-external function.

After steps 5, 6 and 7, the models will be reinserted in the priority queue with their next even time updated. The simulation cycles are repeated until no models are imminent or a termination condition encounters such as exceeding simulation time. This ends our background discussion on the Discrete Event System Specifications (DEVS) formalism environment. The next section will discuss the System Entity Structure (SES) formalism.

2.4 The System Entity Structure (SES)

The System Entity Structure formalism provides a formal ontological framework for specifying real system composition with information about decomposition, specialization and taxonomy. The SES formalism has been applied to many engineering applications and proved its usefulness such as in data engineering [28] and Network systems [22]. In real systems, objects are represented by entities in system entity structure

framework. The SES represents the design space with various possible design configurations. To search for the best configuration, pruned SESs are constructed to reduce the search space into valid instances of the SES. For example, SES can have many specializations and multi-aspects relations; with pruned SES, a decision will be made on which of the entities and specialization should be chosen. The basic components of SES are:

- Entity: entities are representation of some real world objects, which in turn can be made of many other children entities.
- Aspect: represents the decomposition relation. An entity is composed of other entities. The relationship between the parent and the children is “aspect”.
- Specialization: represents alternative choices that a system entity can take. Each of the alternatives is also of type entity.
- Multi-Aspect: is a relation that expresses an all of one kind.
- Variables: are slots attached to some entities in the system. The slots can take values in a specific range. The slots define different contents of the associated entity.

To clarify the use of these components we give the “Book” example which is explained in [5]. A book is an entity; it consists of front cover pages, and back cover (Aspect relation). The front cover is an entity. Also we can say that the front cover can have the color Red, Green or Blue; this shows the specialization relation. A multi-aspect

relation exists between the entity “pages”, which is a child of the parent book, and the entity page means that the book is made of many instances of the entity “page”. Figure 2.4.1 shows the SES structure.

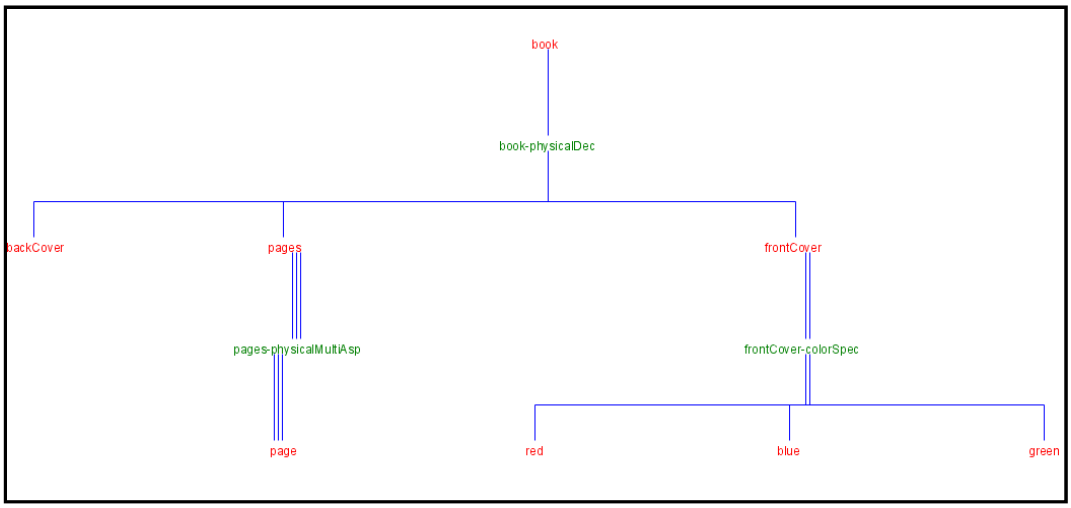


Figure 2.4.1: Example, book SES structure

The process of pruning an SES is to construct a desired structure to meet a particular domain specifications. The pruning process chooses one entity out of many in specialization relations, which results in a completely pruned entity structure PES and variables take values in their ranges [28][5]. The figure below shows the relation between the general SES and the pruned PES. At the implementation level, they are represented by XML schema or DTD and XML instances respectively. We are using SES to construct

ontology for the domain of interest. When we have more than one domain, specialization components exist.

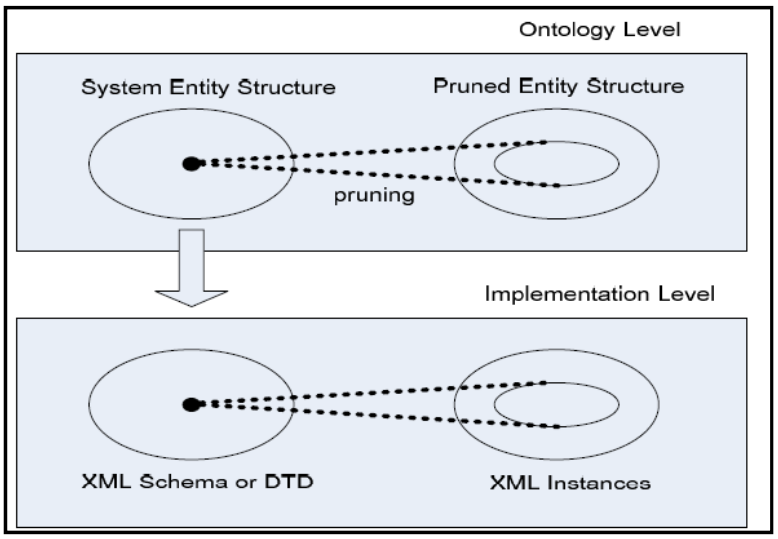


Figure 2.4.2: SES & PES relation-ontology level and the implementation level

CHAPTER 3. BACKGROUND: ONTOLOGY DESIGN LANGUAGES AND THE SEMANTIC WEB

Knowledge representation using ontology structures is a relatively new research topic that emerged with the new requirements of the web. Semantic as well as lexical data availability is intended from the next generation of the web. Which makes the information not only intended for humans, but also to be processable and understandable by machines.

Tim Berners Lee [60] who invented the World Wide Web predicted in the late 1990s that the web will be changing to support data, information and knowledge exchange. In addition to that, he reasoned that the knowledge contained in web pages will be understandable by the machines. Since then, semantic web has become a hot research area in which many parties are cooperating to develop standards and rules to govern the interaction over the net. Semantic web requires interoperability between different services on the web which in turn requires standards not only on the syntactic level, but also on the semantic level.

3.1 Ontology Design Motivation

The success of semantic web is highly depending on the success of ontology design and development. An ontology is an information model that describes concepts and relations in some specific domain. Ontologies enable the processing and sharing of knowledge among different computing sites on the web [29]. Hence, ontologies are

known to be the representation of a shared conceptualization of a specific domain. They provide a common understanding of a domain that can be communicated across people and applications. They have been also developed in Artificial Intelligence to facilitate knowledge representations and sharing. Ontologies will change the way search engines search the web. Currently, search engines use keyword-based approaches to search the web for relevant pages. By using ontologies, search engines can find pages that contain syntactically different, but semantically similar words. An ontology has a hierarchical structure of classes and concepts in the domain of interest and it describes different relations between concepts. Also, it provides a description of concepts through the use of an attribute-value mechanism. Many domains have started to develop and build their ontologies like VnHIES [72] and geographic applications [73].

A typical example of ontology structure and design is shown in figure 3.1.1. Concepts, objects or classes are defined by using *classdef* or *slot-def*, a class consists of a *type*, *subclass-off* and/or *slot-constraint*. A slot-constraint consists of a *name*, *hasvalue* and/or *value-type*. OWL standards are the most promising language for the future of ontology design for the semantic Web.

```
class-def animal                        % animals are a class
class-def plant                         % plants are a class
      subclass-of NOT animal            % that is disjoint from animals
class-def tree
      subclass-of plant                 % trees are a type of plants
class-def branch
       slot-constraint is-part-of       % branches are parts of trees
           has-value tree
class-def leaf
        slot-constraint is-part-of      % leafs are parts of branches
           has-value branch
class-def defined carnivore             % carnivores are animals
        subclass-of animal
        slot-constraint eats            % that eat only other animals
           value-type animal
class-def defined herbivore             % herbivores are animals
        subclass-of animal
        slot-constraint eats            % that eat only plants or parts of plants
           value-type plant OR (slot-constraint is-part-of has-value plant)
class-def giraffe                       % giraffes are animals
        subclass-of animal
        slot-constraint eats            % and they eat leafs
           value-type leaf
class-def lion
         subclass-of animal             % lions are also animals
         slot-constraint eats           % but they eat herbivores
            value-type herbivore
class-def tasty-plant                   % tasty plants are plants that are eaten by
         subclass-of plant              % both herbivores and carnivores
         slot-constraint eaten-by
            has-value herbivore, carnivore
```

Figure 3.1.1: Ontology structure

3.2 Ontology Design Languages and Standards

The interest in defining ontology design languages has been increasing dramatically since couple of years ago. Different research parties are trying to propose well formatted language specifications that can be meet the different requirements of the next generation of the Web [57]. In this section we will give a brief introduction to the most popular web standards that are being used to build ontology structures which are: XML and XML Schema, RDF and RDF Schema, and OWL.

3.2.1 XML and XML Schema

XML differs from HTML in that XML is intended as a markup language for any arbitrary document structure. However, HTML is a markup-language for a specific hypertext documents. This also means, that the tags in HTML are static and standard, but in XML the tags are user defined and the user has the flexibility to have even the same word to have different meanings by pointing to different namespaces. The vocabulary of the tags and their allowed combinations is not fixed. So, an XML document consists of nested set of open and closed tags, where each tag can have a number of attribute-value pairs. Despite of the language limitations, the XML documents have been used widely for different purposes such as in database and information representation and extraction. M. Kim addressed how to extract information from database that is stored as XML files [66]. The ontology structure in figure 3.1.1 can be represented by an XML document as in figure 3.2.1.1.

```
<class-def>
   <class name="plant"/>
   <subclass-of>
      <NOT><class name="animal"/></NOT>
   </subclass-of>
</class-def>
<class-def>
<class name="tree"/>
   <subclass-of>
      <class name="plant"/>
    </subclass-of>
</class-def>
<class-def>
    <class name="branch"/>
    <slot-constraint>
        <slot name="is-part-of"/>
        <has-value>
           <class name="tree"/>
        </has-value>
    </slot-constraint>
</class def>
```

Figure 3.2.1.1: XML document for the ontology structure in figure 3.1.1

It should be mentioned here is that there is no one way of representing information in XML document. Different XML documents might be resulting in the same information which depends on the developer approach. This gave XML the flexibility that attracted people on the Web. However, it is possible to enforce a grammar on tags and their allowed nesting usage. For example, in XML 1.0 a DTD (Document Type Definition) specifies the allowed combinations and nesting of tag-names, attribute-names and other components. XML schema is replacing DTD under W3C recommendations since XML schema offers many advantages and has essentially the same rule as DTD. Any XML document whose nested tags form a balanced tree is a well-formed XML document. XML provides a markup language and a uniform data exchange format for parties over the Web. However, it is important to understand that in both cases, XML enforces only a syntactical structure without any semantic meaning.

Anything that can be represented by a grammar can be encoded into XML documents because XML is for defining data grammars. Many XML parsers have been developed and they exist on the Web where applications and different parties can access them and use them. The major limitation of XML comes in the semantic interoperability, since XML aims at the structure of the documents and does not impose any common interpretation of the data contained in the pages. Hence, in XML there is no way of recognizing semantic units from a particular domain of interest.

The advantage of reusability of XML parsers is useful to the parties who have reached an agreement on their document structures. However this neglects the fact that

partners are often changing dynamically on the Web, which means the documents structures might change. New partners always have to be added to the existing relationship as new information sources become available. Since the scenario of adding new partners is a frequent operation, it is important to reduce the cost of adding the new communication partners as much as possible. Using XML and DTDs or XML schema requires much more effort because the task is not to map one grammar to another grammar, but to map objects and relations from one domain to another domain. Subsequently, we need to define the mapping between DTDs (or grammars). The following would be the steps that need to be executed:

1. Reengineering of the original domain model from the DTD or XML schema. This is a very difficult task to be performed given that the mapping is not a one-to-one relation. One DTD can result in many different domain models if agreement was not achieved in advance between parties.
2. Establishing mapping between the components of the domain model which involves concepts and relations.
3. Defining translation procedures for XML documents. This is also a hard task to be performed since it depends on the particular encoding chosen to construct the initial DTDs (or XML Schema).

From what has been mentioned previously, using a more suitable formalism for data transfer and information exchange than XML can save a lot of work. XML would be an

elegant tool to be used for data exchange between applications that both know what the data are, but not in situations where we need to add new partners frequently.

3.2.2 RDF and RDF Schema

Resource Description Framework is a recent technology recommendation by the World Wide Web Consortium (W3C). RDF aims to standardize metadata descriptions about recourses on the Web. Since RDF is capable to describe data about web resources, it is also capable of representing data. The basic component or structure in RDF is called a *triple*; triples are relations that connect objects to their values. For example, a relation *A* that exists between object *O* and the Value *V* is represented by triple *A(O,V)*. RDF triple can be represented as a graph with two nodes and an edge that connects them, referred by labeled graphs [75]. The nodes represent the object and the value, while the edge represents the type of the relations that exists between the two nodes. RDF allows objects and values to be mixed. Hence, this leads to chaining in graphs. For example, Figure 3.2.2.1 represents the following three triples:

hasName('http://www.w3.org/employee/id1321', "Jim Lerners")
authorOf('http://www.w3.org/employee/id1321','http://www.books.org/ISBN001251586')
hasPrice('http://www.books.org/ISBN006251586 1, "$62")

Reification in RDF allows an RDF statement to be the object or the value of another triple which leads to nesting or recursive definitions of semantic objects. Also an object B can

be given to designate a certain type such as *'ISBN03547X'* is of type *'book'*. RDF vocabulary has no restrictions on the property names that can be used, same as in XML. The main intended rule of RDF is to provide object-attribute-value triples data models for metadata.

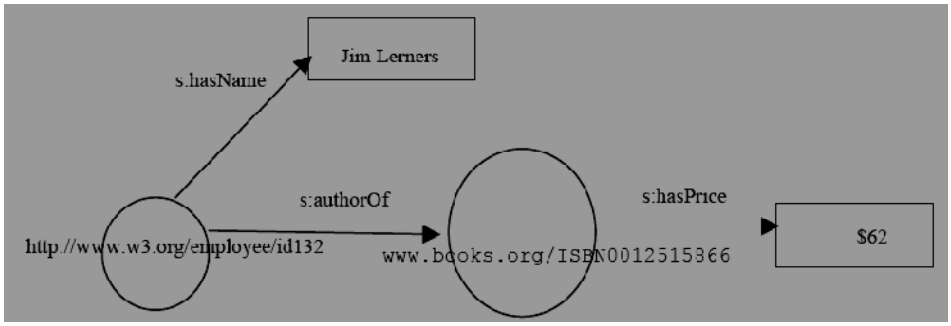


Figure 3.2.2.1: Nested RDF graph

As the XML schema provides a vocabulary definitions facility for XML, RDF schema provides a similar facility for RDF which provides a basic type system for RDF models. This basic system uses predefined terminology such as *Class* and *subClassOf*. RDF schema expressions are also valid RDF expressions, the only difference is that the RDF schema predefines a particular vocabulary that should be used for RDF attributes (e.g.*autherOf*) and specifies the types of objects that these attributes may be applied to.

RDF objects may be instances of one or more classes depending on the *type* property. Two important RDF constructions are *subClassOf* and *subPropertyOf*. The *subClassOf* property allows the specification of hierarchical organization of such classes. *subPropertyOf* does the same for properties. Constraints on properties can be specified using domain and range constructs. Using these constructions, RDF schema can extend

the vocabulary and the intended semantic interpretation of RDF expressions which puts it on top of RDF.

By using nested object-attribute-value triples, the universal expressive power holds for RDF. Also different RDF parsers have been developed and can be used by different parties on the Web, hence the reusability requirement holds for RDF.

Semantics structures and units are given by nature through the RDF triples where all objects are independent entities. This gives RDF an advantage over XML as being the suitable technique for semantic web where no need for translation steps. In describing some specific domain, representing the objects and relations in that domain are what matter, which is what RDF triples do. We can apply techniques from knowledge and representation to find the mapping between RDF descriptions. The usage of RDF for data interchange raises the level of reusability beyond the parser to the domain model itself, which cannot be achieved from using plain XML. RDF technique provides us with the capabilities for knowledge representation which can be shared over the Web. RDF Schema gives more power on top of RDF by freeing us from the limited primitives of RDF. Since our concern is to have a semantic meaning of web pages content, we should consider two main approaches in computer science which are: declarative and procedural semantics.

In the declarative semantics, the meaning of an expression E can be found from the conclusions or properties that follow from expression E; the conclusions or properties are well understood formalism where machines can process and understand. However, in

the procedural semantics, the meaning can only be found by executing some program (computational procedure) on the expression E and analyze the resulted behavior. This difference between declarative and procedural semantics is very close to the semantic interoperability difference between XML and RDF.

An expression written in XML and DTDs (or XML Schema) has no inherited semantics, and the meaning of it depends on the application that is executing it. Two different applications running the same expression will have two different meanings for it. Although for specifying structural models, XML seems better than RDF. On the other hand, an expression in RDF or RDF Schema will have the same declarative semantics. This follows naturally from it is being independently of any program or application running it. Hence, any RDF processor must conform to this intended semantics. Communities in computer science, AI and W3C all agree that the declarative semantic technique leads to a more shareable and reusable knowledge representation sources than what could be achieved from using procedural semantics [29]. Hence, we can conclude that RDF and RDF Schema is a better technique for information and data representation in the semantic web than XML and DTDs (or XML Schema).

As we mentioned earlier, ontologies are the basic units that build the sharable knowledge in the semantic web. Defining an ontology in RDF means defining RDF Schema (RDF Schema is an extension of RDF), which means defining all concepts, terms and relationships in a specific domain. This imposes some more requirements on RDF and RDF Schema which resulted in W3C recommendation of adapting OWL as the

language for developing ontologies. The next section discusses OWL standards, advantages and disadvantages.

3.2.3 Web Ontology Language (OWL)

The World Wide Web Consortium has approved two key technologies for semantic web development: the Recourse Description Framework (RDF) and the Web Ontology Language (OWL). These two semantic web standards provide the power for integration, sharing and reusing data and knowledge on the web. Users will be able to share the same information regardless of the applications. We have discussed RDF in the previous section; this section aims to introduce OWL as the required technology for designing semantic web ontologies. OWL is used to give machines the ability to process and understand information on the web. OWL can be used to explicitly represent the meaning of concepts and vocabularies and the relationships between them. OWL has more capabilities than XML, XML Schema, RDF and RDF Schema which promotes it as the semantic language for knowledge and data representation on the web.

OWL is a revision of DAML+OIL [78] web ontology language incorporating lessons that have been learned from the design and application of DAML+OIL. OWL-DL (Description Logic) is a sublanguage of OWL.

L. Rector and his co researchers in their paper [50] on using OWL to represent pizza Ontology described how to use OWL-DL to design ontology. Then they discussed the errors and pitfalls that users made in writing information representations, paraphrases and their role in clarifying meanings. Ontology to represent the pizza concept was chosen as an example because it is concrete, physical and rich enough to illustrate key issues in

the design. However, the authors showed that constructing the correct definitions of pizzas from a menu turns out to be a challenging exercise. For more details on the example, please refer to the number [50] in the references.

Since OWL is derived from Description Logic, OWL has model-theoretic semantics that provide the official meaning for OWL documents [61]. Since OWL was produced by the W3C Web Ontology Working Group, it does suffer from their vision of the future of the semantic web [53] because their vision does not allow different semantic web languages to have different syntax. The OWL provides more capabilities than RDF Schema, however there are few tools available that can process OWL documents because it is relatively a new ontology language.

Summary

In this chapter we showed that the ontology design and processing are widely used and being researched by the Semantic Web community. That is because the design of the ontology gives them the document structures capability to express the web page contents based on their semantic meanings rather than their lexical formats. Different research techniques have been addressed to automatic ontology creation, merging, integration, ontology reasoning and collaboration such as in [71].

CHAPTER 4. NEGOTIATION PROCESS AND PROTOCOLS

Negotiation process is an interaction between two or more parties in an attempt to reach some agreement on a specific aspect. The aspect for which the negotiation takes place upon takes a wide range of topics based on the application. For example, it might be a price of some goods as in e-commerce, or information availability and data provision. During the negotiation process, agents exchange their capabilities and what services they can provide. Many researchers have proposed solutions to the process of negotiation but their solutions have always been for limited cases under specific domain of applications. We aim in this work to provide a generic-automated negotiation model that can be utilized under different engineering applications. Our research interest falls into the negotiation language area which discusses the design of the negotiation protocols, the negotiation primitives, the semantics and object structure. Protocols refer to rules that agents must follow during their interactions with other agents [8]. Section 4.1 and 4.2 will explain our design of the negotiation protocols. Section 4.3 describes our approach for designing the negotiation primitives and the object structure. The semantics are not addressed in our work because they are not necessary for the completeness of our objectives and goals. Section 4.4 will show how the marketplace architecture can help the negotiation parties in reaching agreements efficiently.

4.1 One-to-One Negotiation Protocol

Many system designers have applied the negotiation process to different domains. Based on the objectives of the systems, different types of negotiation are developed such as: collaborative environments, buyer and seller negotiation, negotiations for resources and data reservations and so on. Murugesan [11] discussed different issues concerning automated negotiation for electronic commerce. Some researchers are trying to apply collaborative negotiation activity for e-commerce where different threads (parties) are independent from each other [7]. Feng and Lei used a constraint network to measure conflict costs for collaborative negotiation and a state diagram to model the negotiation protocol. In all negotiation systems, agents must follow some rules of interaction known as "Negotiation Protocols". These protocols define how parties can interact with each other which in turn affects their decision and expressiveness capabilities. In most current e-commerce solutions, the conflict is related to the price of the items between the sellers and the buyers [65].

One important property of the negotiation process is a One-to-One protocol. In this protocol, negotiating parties can communicate (negotiate) with each other via offers and counter offers cycles. The process starts when the requestor sends a *Request*, then the provider replies with either, *Accept* where an agreement is established, *Reject* where no agreement has been reached or *Offer* where requestor needs to make evaluation upon receiving it whether to accept it or reject. If the requestor response on the *Offer* was *Accept*, then an agreement has been reached; if he replies with *Reject* there is no

agreement. The third choice is to reply with a *Counter-Offer* message. The cycles can go on forever. However, in real life and software developments, a predefined time is allowed before the termination of the process as we discussed in section 1.1.

In papers [6], [14] and [70] a simple negotiation protocol is used. It occurs between two agents to support a shared semantic ontology of the terms and primitives that can be used in the negotiation process. Figure 4.1.1 shows the One-to-One protocol nature.

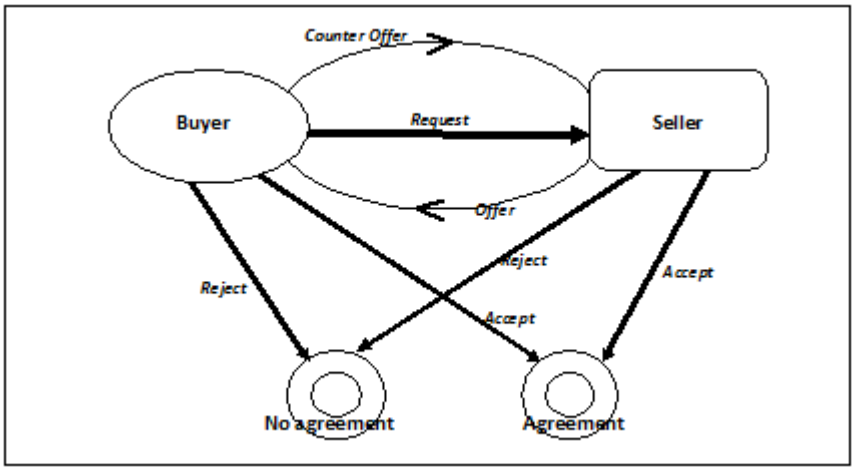


Figure 4.1.1: Simple sequence of negotiation activities

Figure 4.1.1 shows some primitives along with a sequence of negotiation protocol (rules). However, it does not reveal details on the syntax involved in using these primitives nor does it show semantic specifications. Such a simple scenario is limited to the situation where the interacted agents know each other IDs. For example, it is valid if the buyer knows the seller, and if both have sufficient ability to control their items and

services. In many cases, however, more sophisticated protocols are needed to support dynamic behaviors during the negotiation process. For instance, the buyer might not know what services or products are available for him or if the buyer wants to complain about a transaction. Hence, a more flexible and comprehensive negotiation model is required. Transparency and privacy are other requirements that negotiation models need to support.

The objective of Bailin and Truszkowski's research [2] on scientific archives was to find relevant information on a specific topic. Again the One-to-One negotiation protocol has been used as one rule between agent A and agent B (two parties trying to negotiate on scientific archives information). Masvoula, Kontolemakis, Kanellis, and Martakos [12] discuss the issue on how a negotiation model should be as close as possible to the real interactions in auctions and the bargaining behaviors in the stores. The protocol design is assumed to be One-to-One with offers, evaluation of offers and counter-offers. Research in e-learning needs different expressive requirements than other domains like the e-commerce. In a collaborative e-learning domain, the negotiators will ask questions, answer questions, confirm information, etc. [9]. The objective here is to reach an agreement and one understanding on topics or ideas. Although the negotiation primitives are different, they used a One-to-One negotiation protocol.

The work by M.Addis, P.Allen and M.Surridge on negotiation for software services used the One-to-One Negotiation protocol to support on-demand software and hardware resources sharing environments [21]. V. Krishna and VC Ramesh proposed a

model for competitive decision making agents where they used the One-to-One protocol to support the bargaining process when agent “a” calls agent “b” [10].

Because of the nature of the One-to-One negotiation model which models real life bargaining and negotiation behaviors, we are adopting this protocol for its simplicity and advantages. However, some modifications are needed concerning its definition to fit into our generic automated framework. Section 4.4 on marketplace architecture and design gives our definition of the One-to-One protocol.

4.2 Service Discovery Negotiation Protocol

Most of the current negotiation systems and distributed services management tools do not support brokering between agents. Distributed services environments do not interact with their users on different specifications. For example, if a user would like to use a service that is deployed on a distributed environment and that service is already being used, he will get usually a response that it is not available at this time. Then the user needs to request that service maybe every 1 minute. On the other hand and in some other extreme cases he might receive a decline that he cannot use this environment because of a simple error he made or because the service does not provide one of the job specifications he asked for in his request. In some case it might be that the user can ignore one of specifics because the execution of his job will satisfy his needs. In such case brokering and negotiation is very essential to reach agreements in multi-agent environments.

Yilmaz and Paspuleti [25] used a *Broker* agent to support transparency, a *Matchmaker* to bring different views using relevance metrics that are independent of keyword matching, and a *Mediator* agent to convert contents to some common reference model (constructed as ontology) that negotiators understand. The Mediator agent resolves four types of conflicts: semantic, descriptive, heterogeneous and structural. Tamma, Phelps, Dickinson and Wooldridge [24] used a shared ontology to model the protocols that could be encountered or needed in supporting agent negotiation in e-commerce environment. They call such enforcement of rules "rules of encounter". According to this paper, the agents do not go into different states or decision making phases. However agents query the shared ontology for the next step in response of an event occurrence. Such an implementation is slow and lacks a scalability requirement. Also, it is a single point of failure implementation with unmanageable size of ontology when the ontology grows up to handle more space of dynamic behaviors. Bailin and Truszkowski [2] on scientific archives, the system designers used marketplace architecture to resolve semantic mismatches in real time without human intervention. The protocol they used is a One-to-One protocol between for example agent A and agent B. However, the definition and functionality is different because they exchange different types of primitives that need different ways of handling.

In order to support flexible generic negotiation protocols that can capture different user behaviors, we determined the following requirements that we believe any negotiation system should support them:

1. The ability to ask a service provider (might be a computing node) for a service or an offer.

2. The ability to negotiate with the service provider over jobs/products specifications (e.g. execution time for a job, bandwidth of data transformation after the job is finished).

3. The customer ability to respond with a counter offer that represents its interests.

4. The ability to complain to a third party that controls web services. This is important in order to support customer satisfaction over the web to build a trustable environment between services requesters and services providers.

5. The service provider ability to advertise their capabilities to be found later when customers search for services. This provides transparency and privacy between users and providers. For example a user might request a book with a specific ISBN number; the result will be all bookstores that have that specific book available. As a result, the user will negotiate with the best book provider that meets his/her interests.

6. Supporting decision making capabilities for customers to choose the best among different service providers.

7. Monitoring agreements which were achieved between customers/users and the service providers. For example, a situation that appears in the domain

of information discovery and retrieval, a third party is needed to monitor the transfer of data from the data provider to the requester according to the agreement guidelines that both agreed upon during their negotiation.

8. The ability to reformat or add/remove some parameters to the messages being exchanged between customers and service providers to avoid misunderstanding or confusion and for future purposes.

Supporting these requirements is not an easy task because of the dynamic nature in multi-agent environments. However, the choice of using marketplace architecture (a third party such as a controller or mediator agent) is useful in this regard, in addition to the choice of a new negotiation protocol namely "*Service Discovery*". The marketplace can act as a broker, a mediator, a controller and/or a database for service providers to advertise their products/services.

The service discovery protocol is needed when a customer is searching for a specific service or product. Customers then query the marketplace to find the best providers. The marketplace responds to the customers with a group of service providers who fulfill their requests. After the customers get the results back from the marketplace, they will have a list of the available service providers and their capabilities. Then the customers can decide on whether to proceed with the negotiation process or not. If the customers choose to proceed, they send a contract query to the marketplace, and then the marketplace will forward that to the appropriate providers and wait for responses from

them. Next chapter on implementation explains the two negotiation protocols enforced by the marketplace in more details.

4.3 Domain-Dependent Language of Encounter

In this section we discuss the second part of the language which is the negotiation primitives. The primitives give us insights into the language of using them which indicates different phases and functionalities. We call such messaging language "*Language of Encounter*". So whenever we say language of encounter then we refer to the negotiation primitives or the negotiation messaging system.

4.3.1 Language of Encounter Taxonomy and Structure

In our design of the negotiation model, we specified the language of encounter that web agents or our system users need to use for their interactions. O'Hare and Jennings [8] suggest three groups for language of encounter. However, such a classification is not enough to truly enable the negotiation process over the web and in multi-agent environments. More types are needed to support customer satisfaction and the dynamic in user behaviors. We have defined two new necessary classes for the messages to increase the expressiveness power and the negotiation capabilities.

Table 4.3.1.1 shows the language of encounter (messages) to which agents refer in order to express their needs. The difference between "*Decline*" and "*Reject*" is when the marketplace is too busy and cannot handle more requests. It might not choose to start

("*Decline*") the negotiation process. Note that the negotiation did not take place in this situation, which allows the requester to try to start the same negotiation later. However, "*Reject*" means that the negotiation process already took place and the result is no agreement, so there will be no point of trying again later to establish the same negotiation process under the same parameters. On the other hand, "*NotMet*" refers to situations where the two negotiation parties have come to an agreement and they started the transaction. However, one of the two parties has violated the agreement terms that both established before. In such a situation, the marketplace needs to stop or terminate the transaction. For example: if an information agent negotiates with a service provider to transfer some audio traffic with a minimum speed of 200kB and the service provider agrees on that, and subsequently, after establishing the link between them, the service provider was transferring the traffic with speed less than 200kB, then the information agent can ask the marketplace to terminate this contract by sending "*NotMet*". "*Terminate*" message means that the negotiation process has started but is not finished (still in progress and the result is not known yet). Then the requester has the right to stop the negotiation.

Abort	Initiators	Reactors	Completers	Informative
Terminate	ContractQuery	Offer	Reject	Busy
NotMet	CapabilityQuery	CounterOffer	Accept	LinkEstablished
	ItemRequest	Decline		Item
		CapabilityStatement		ItemCheckResult
				BestProvider
				ProvidersChosen

Table 4.3.1.1: Classification of the language of encounter

The usage of these primitives will be clear when we discuss the marketplace architecture along with its phases and transitions. The last point to address in our research design of the negotiation language is the object structure, which refers to the language of encounter structures in our context. The structure of each message type depends on the domain under which it is being used. Hence, in order for our system to support negotiation services under different domains, a dynamic message structure is the key role to the success. In this implementation we used a shared Ontology that defines each message and its usage under different domains. Each domain would be a specialization in the message structure. Each Message type has a separate structural ontology defining its variables/fields. System Entity Structure (SES) formalisms is a useful tool to define the

language of encounter structure. For example, Oceanography is a sub domain of the domain surveillance and has specific structure. Online store is a sub domain of the domain e-commerce and has a specific structure. Figure 4.3.1.1 shows the purpose of designing ontology for a specific primitive, “MessageX”.

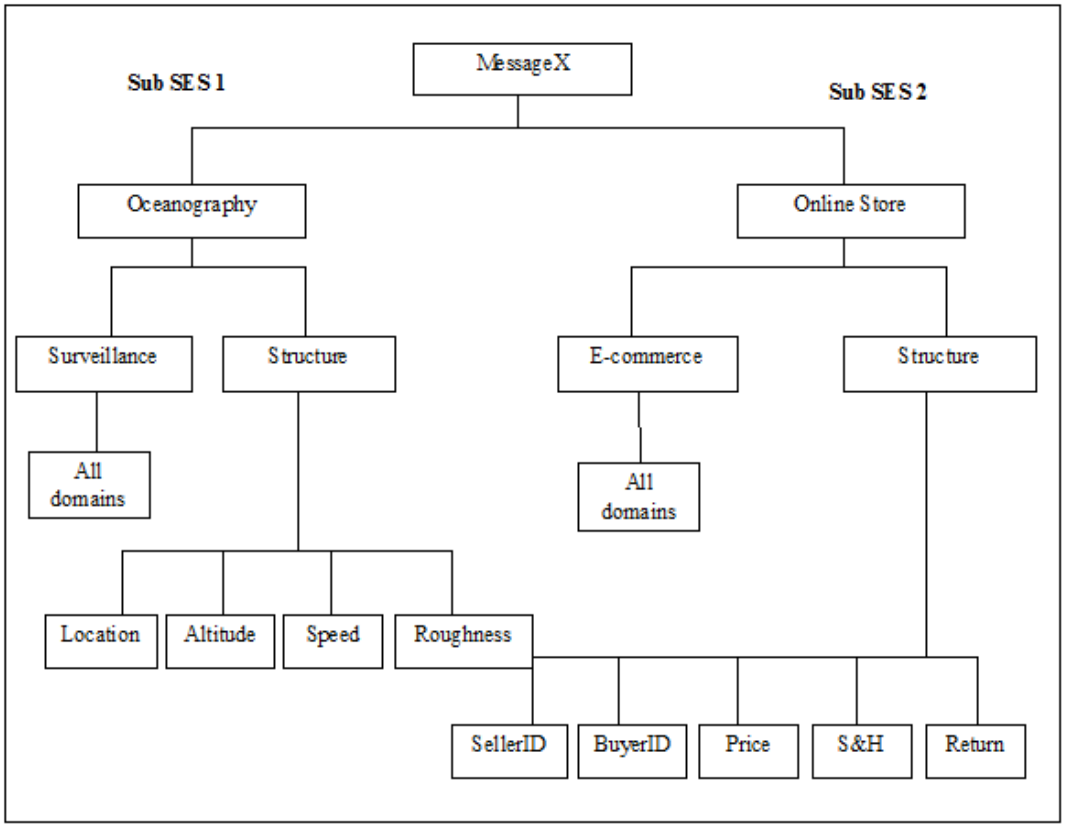


Figure 4.3.1.1: Ontology design for MessageX type

The alternatives here are that if there are two domains defined for MessageX ontology, then two cases might occur:

1. Given the input is “MessageX the sub domain Oceanography”, then the system should automatically select the message structure to be sub SES 1 represented in the variables (Location, Altitude, Speed and Roughness).
2. Given the input is “MessageX and the sub domain Online Store”, then the system should automatically select the message structure to be sub SES 2 represented in (SellerID, BuyerID, Price, S&H and Return).

4.4 Domain-Independent Marketplace Architecture

We have specified the language of encounter that user agents need to use when interacting with the marketplace and/or with each other. The Marketplace controls the behavior of the interacting agents by enforcing our model rules and policies (negotiation protocols). Here we show the marketplace architecture design which is based on finite state model. The section describes the different phases of the marketplace agent and ends with diagram showing the sequence of phases that the marketplace goes though. Table 4.4.1 shows the different states of the marketplace along with their descriptions.

Figure 4.4.1 shows the transitions between phases of the marketplace agent. This model of the marketplace can be translated easily into an FD-DEVS implementation. However, because of the specifications of FD-DEVS, some phases need to be reformatted according to the messages that cause the transition to these states. The next chapter on FD-DEVS implementations explains how to split some states according to the language of encounter.

The marketplace enforces two negotiation protocols (rules and policies) in its multi-agent negotiation environment. These are:

1. One-to-One negotiation when entities know each other's ID.
2. Service Discovery: When a customer is searching for a specific service or product, it usually looks for the best provider among the participants. The marketplace plays a role here on behalf of the requestors.

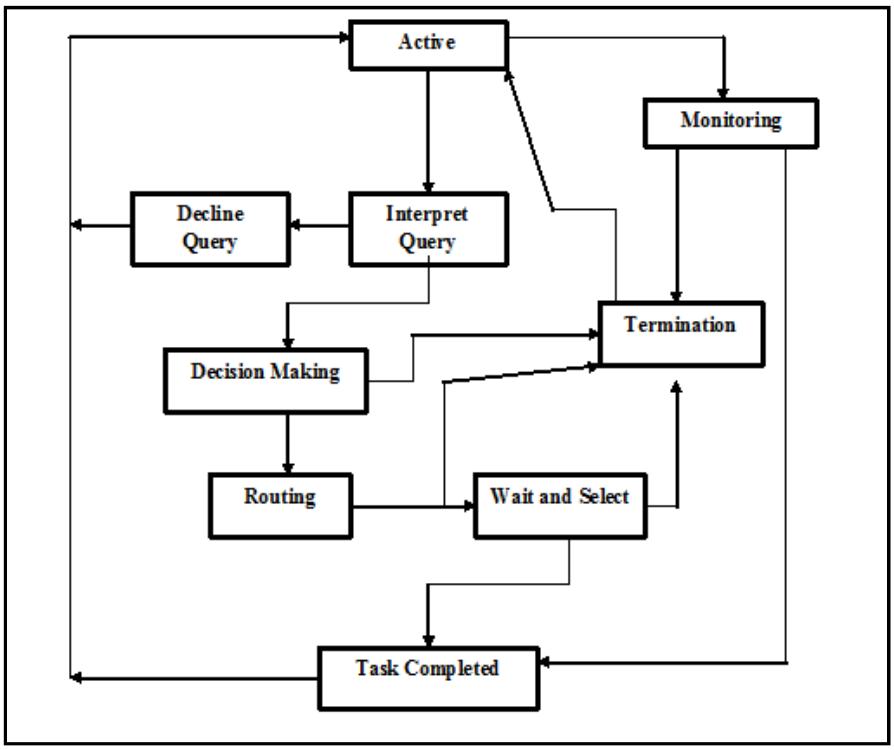


Figure 4.4.1: Marketplace state machine diagram

Phase	Description
Active	Means that the marketplace is ready to receive different types of messages (language of encounter)
Routing	The marketplace acts as a router, its task is to forward the received messages to the appropriate receivers, this scenario occurs frequently when the two parties know each other (buyer and seller know each other their Id).
InterpretQuery	Upon receiving a contract query, the marketplace goes into this state to interpret the query based on messages structures, lexical and/or semantic meaning to be understood.
DeclineQuery	If the marketplace is too busy and it cannot handle new requests, the marketplace goes into this state to send a "*Decline*" message to the customer.
DecisionMaking	After performing interpretation task on a received query and choose to serve it, the marketplace goes into this state to decide on the appropriate receivers, make some modifications on the query (such as reformatting it), accessing the database to find information about the service providers, and so on.
WaitAndSelect	After forwarding a request to service providers, the marketplace wait for responses from the specified providers to select the best that meets the requirements of the customer.
TaskCompleted	After finishing serving a query, the marketplace transit to this state to report that the request was completed successfully by reporting some information about the transaction that might be needed in the future.
Monitoring	While the marketplace in this state, it monitors the process of data transferring for the specified period of time.
TransactionReview	After a transaction is completed between a seller and a buyer and if the buyer complains about the item description, the marketplace transits to this state to resolve the issue.
Termination	This means that the transaction was terminated for some abnormal reasons.

Table 4.4.1: Marketplace states and their description

CHAPTER 5. SYSTEM IMPLEMENTATIONS

In this section we will show our implementation work of the negotiation protocols represented by the domains independent Marketplace architecture. Also we will discuss the dynamic message structure implementation. We used FD-DEVS formalisms to implement the marketplace model and we used SES formalisms to build the messages structure ontology. The following sections explain both implementations.

5.1 FD-DEVS and the Marketplace Architecture

We have implemented the above negotiation protocols in FD-DEVS. Finite & Deterministic Discrete Event System Specification) is a formalism for modeling and analyzing discrete event systems in both simulation and verification ways. FD-DEVS is based on DEVS formalism [1] [3][4] . However, to implement it in FD-DEVS, we needed to do extra work by splitting some phases into multiple copies based on the message that causes the transition. For example: when receiving message "*Request*" that needs to be forwarded to a specific seller, the marketplace transits into phase "*RoutingRequest*", where it routes the message to the seller. When receiving an "*Offer*" message from a seller that needs to be forwarded to a specific buyer, the marketplace transits into phase "*RoutingOffer*". Note here that the task to be performed by the marketplace is forwarding a message from a specific customer (e.g. buyer) to a specific service provider (e.g. seller). This scenario occurs when both negotiating entities know each other. The work to be done by the marketplace is basically the same in both phases

(forward a message), but the difference is in the message type. To implement this in FD-DEVS, we needed to differentiate between the two states to convey two different messages. Our negotiation protocol consists of two scenarios:

1- One-to-One negotiation when entities know each other ID. In this protocol, the customer agent sends messages (such as request, contract query, counter offer, accept, reject) to the marketplace including the service provider ID. The marketplace reveals the service provider ID from the received message and forwards it to the specific service provider (receiver). On the other hand, the service provider responds with replies (such as offer, accept, reject) with the customer ID included in the contents of the messages. The marketplace receives the messages, unmarshal them to find the customer ID, and then it forwards them to the specific customer. If the customer did not receive the correct item, it can choose to complain by sending "*Item*" message to the marketplace along with the transaction number. Then the marketplace searches its log files to find the transaction information in order to resolve the issue with the service provider. Figure 5.1.1 shows the protocol flow.

2- Service Discovery: When a customer is searching for a specific service or product, it usually looks for the best provider among the participants. Customers then query the marketplace to find the best providers, and since service providers advertise their services, information and products to the

marketplace, the marketplace will have an updated database of the members of the service providers and their capabilities. The marketplace responds to the customers with a group of service providers who can fulfill their requests. After the customers get the results back from the marketplace, they will have a list of the available service providers and their capabilities. Then the customers will decide on how to proceed with the negotiation process. The customers might choose to proceed with the negotiation by sending a contract query to the marketplace; then the marketplace will forward that to the selected providers that were chosen in the previous step, then it will wait in phase "Wait" to receive responses from the providers one by one. Once it finishes waiting in that phase, it will select the best offer from the list of responds. The best provider will be sent back to the customer. The customer now can choose whether to accept the offer, reject the offer or go to the one-to-one protocol and negotiate with the chosen provider. Once an agreement is reached. The customer establishes a link with the appropriate service provider to transfer data, information or products and then it informs the marketplace of the link establishment. The marketplace now enters a "*Monitoring*" phase to make sure that the agreement is fulfilled. Figure 5.1.2 shows the scenario

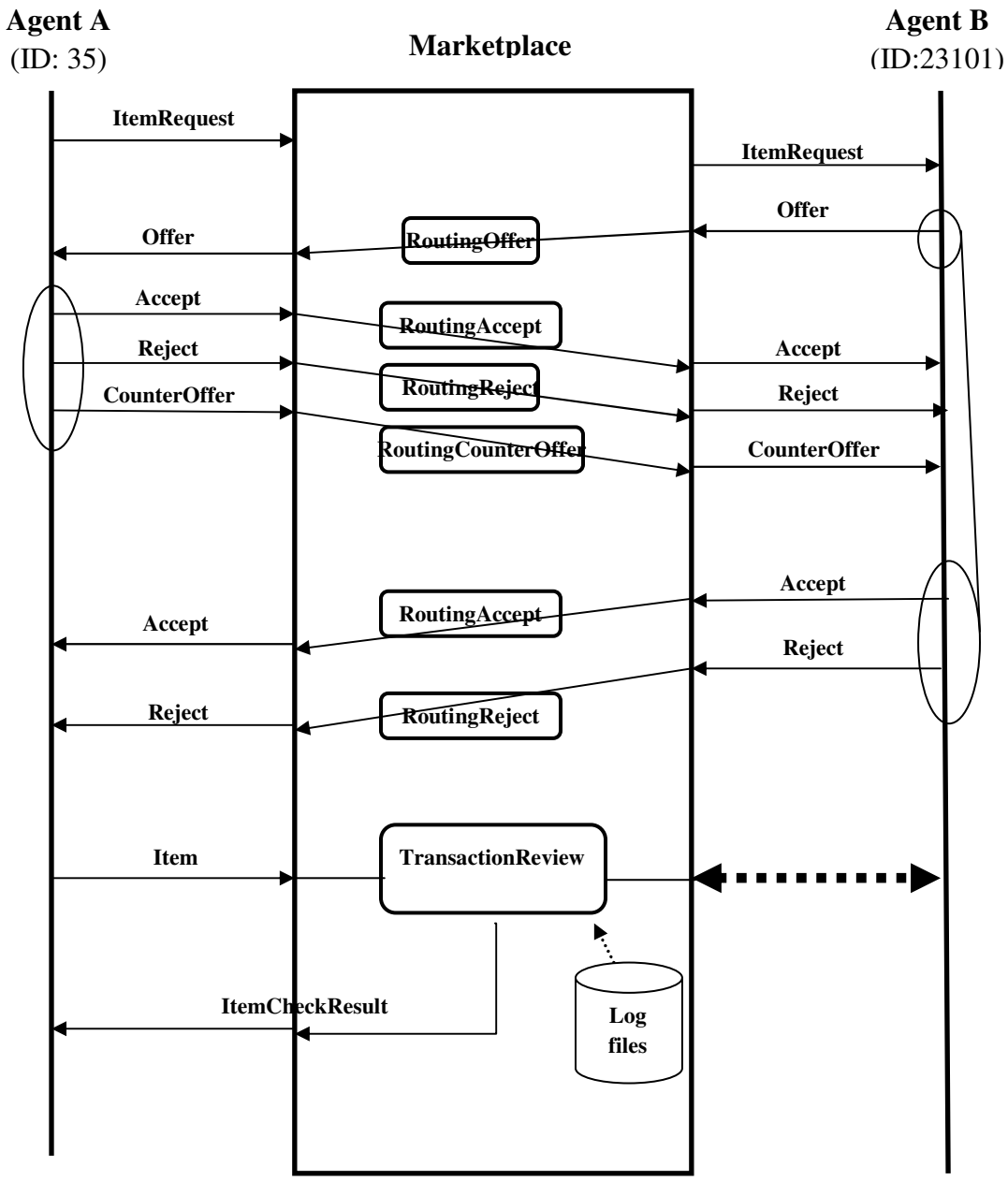


Figure 5.1.1: One-to-One negotiation protocol

Notice from the figure above that when agent **A** receives an "*Offer*" from agent **B**, then he can send back to agent **B** either "*Accept*", "*Reject*" or "*CounterOffer*" messages.

When agent **B** receives "*CounterOffer*", then he can send back to agent **A** either "*Accept*", "*Reject*" or a modified "*Offer*" on the same product/service. In any case, one of the agents has to send "*Accept*" or "*Reject*" sometime to end the negotiation process, otherwise they might go into an infinite loop. This is easy to resolve when designing customers and providers agents. One way to solve this problem is by having a timing counter, once it expires, the agent sends a "*Terminate*" message rather than wasting the time with an endless negotiation.

When the marketplace receives complaints regarding a transaction ("*Item*"), it interacts with the item provider to resolve the issue (the two ways dotted arrow in the figure above). Such an interaction depends on the regulations of companies. In e-commerce, usually the product provider (seller) refunds the buyer the item price and the buyer returns the item. Some companies provide products exchange option.

In figure 5.1.2, when the marketplace processes the "*ContractQuery*" message, it needs to decide on the appropriate receivers of the query (service providers who have the requested service available). Then it forwards the contract query to the chosen providers and waits for responses from them. After receiving the responses it selects the best that meets the requirements in the contract query and sends its ID back to the customer. At that time, the customer will choose whether to establish a link with that provider or maybe negotiate with the selected provider for a better deal using the One-to-One protocol (since the customer agent knows the service provider ID).

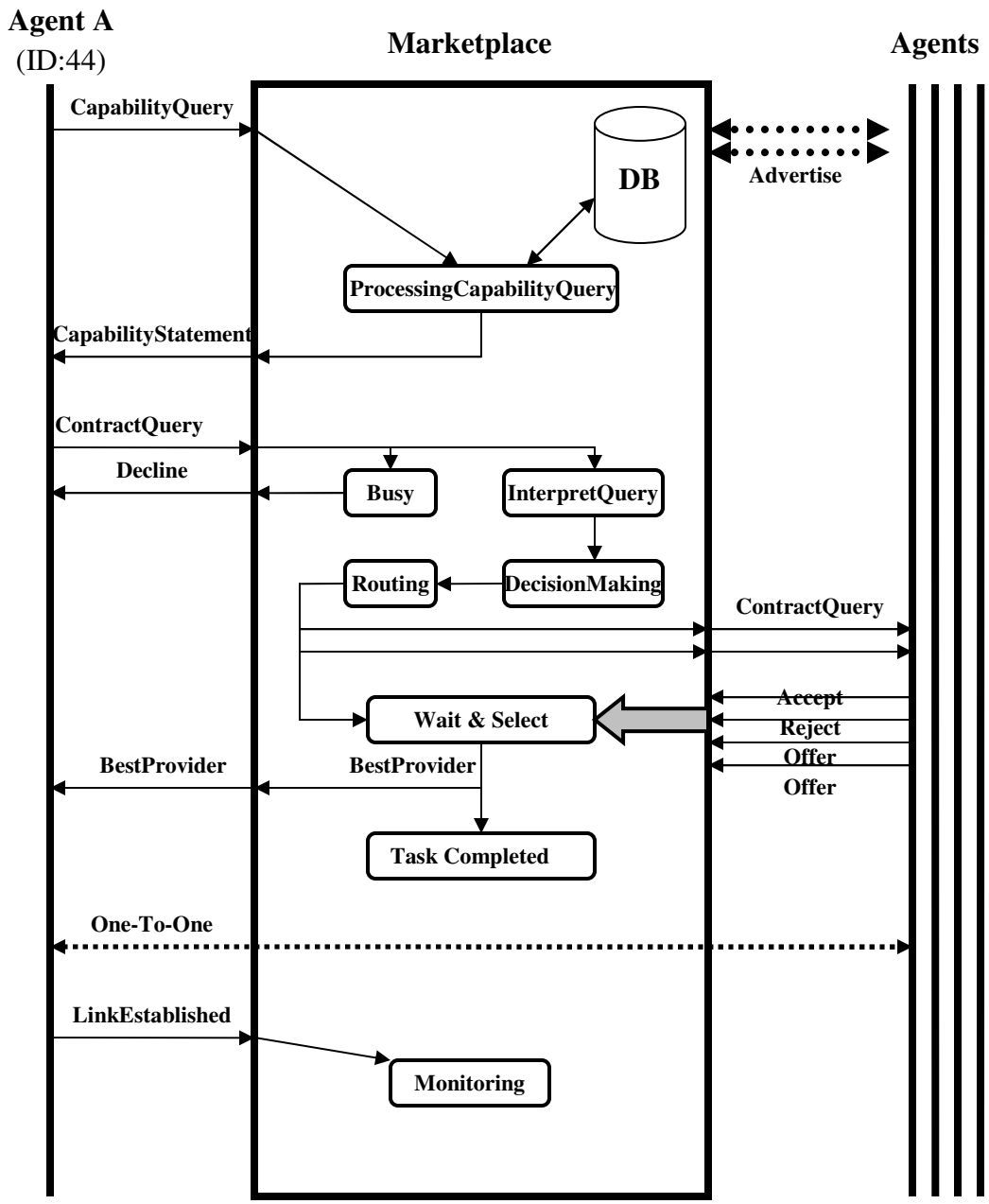


Figure 5.1.2: Service discovery negotiation protocol

We used the Finite Deterministic GUI tool version 0.6.0 to define the marketplace model. In using the tool, we need to specify three main categories which are shown in the

figures below. Figures 5.1.3, 5.1.4 and 5.1.5 show the states table, internal transition function and the external transition function respectively.

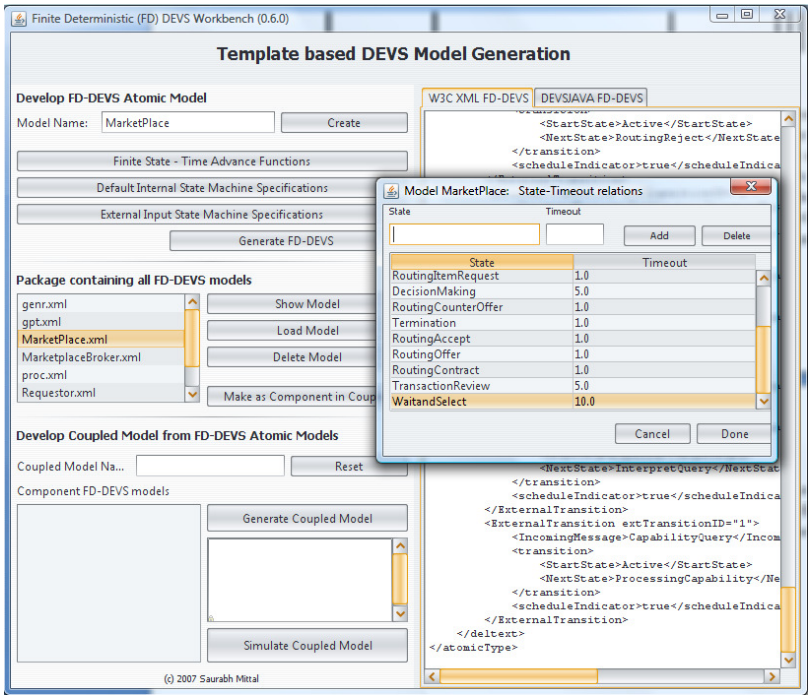


Figure 5.1.3: Marketplace model (states table)

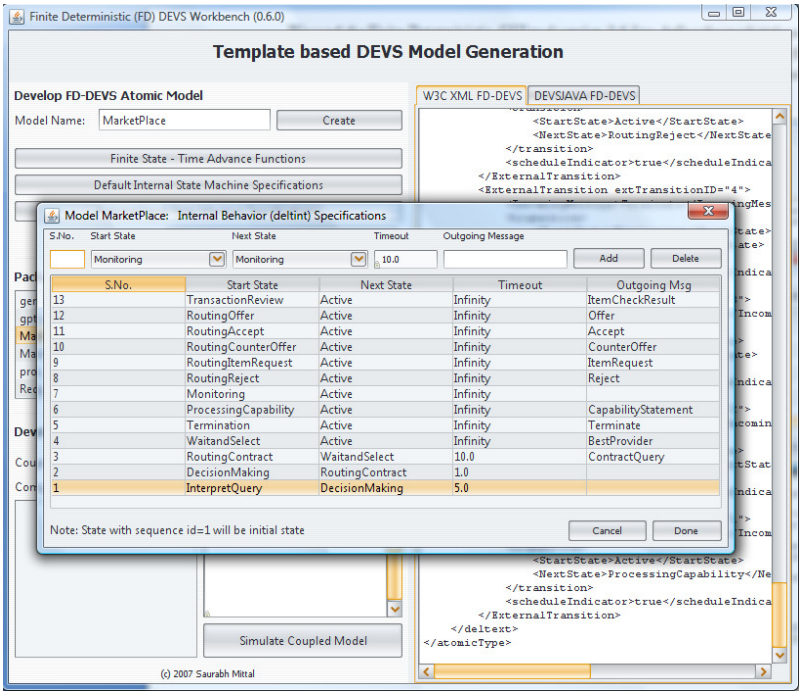


Figure 5.1.4: Marketplace model (internal transition function)

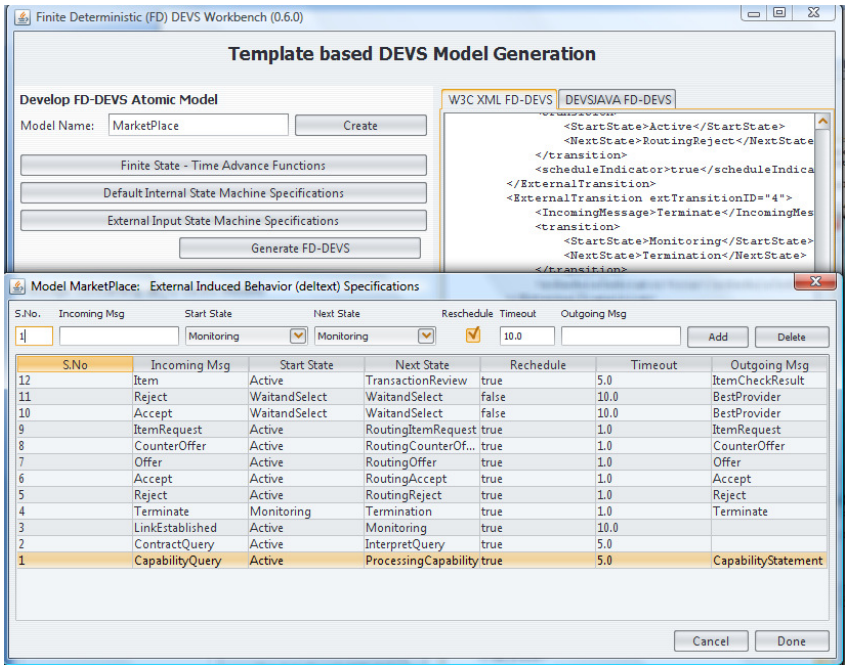


Figure 5.1.5: Marketplace model (external transition function)

The FD-DEVS tool generates an XML representation of the model. This XML file is very important since it can be used directly to into our automated version of the system as will see in the next chapter on automatic code generation of the marketplace model given a domain name. The in ports and out ports names are generated based on the name of the messages; for example, the message “Accept” will be received on in port “inAccept” and will be sent on out port “outAccept”.

5.2 SES and the Messages Structure Ontology

We discussed in the previous chapter in section 4.3 about the messages structure that it should be dynamic based on the domain of interest. This is because the information that needs to be sent through messages is different. For example in E-commerce domain, the agents consider parameters such price, shipping and handling, return policy for their products and services. However, agents in Oceanography or software services will have different parameters that they care about such as execution time, bandwidth, latency. In some cases, even under the same domain, the designer can construct the structure of a message to have more than one meaning. Mathieu and Verrons [74] in their attempt to provide a flexible negotiation protocol, they had to add more stages on the One-to-One negotiation protocol to provide “modification request” and “propose modification”. In order for our negotiation primitives to accommodate for varying capabilities under different domains, the messages structures must be dynamic and based on the domain under consideration. Hence, in our design we use an ontology structure for each type of

messages. The design of the ontology is shown in Figure 5.2.1 for message *ContractQuery* as an example. The message type (entity) has specialization relations to each of the domains defined (*SubdomainOfIntrestSpec*). Each domain entity has a decomposition relation with its "domainMsgStructure" which refers to the message structure. Each domainMsgStructure has variable slots (fields) that contain the different parameters of the message structure such as (Price, SellerID, Location, Roughness).

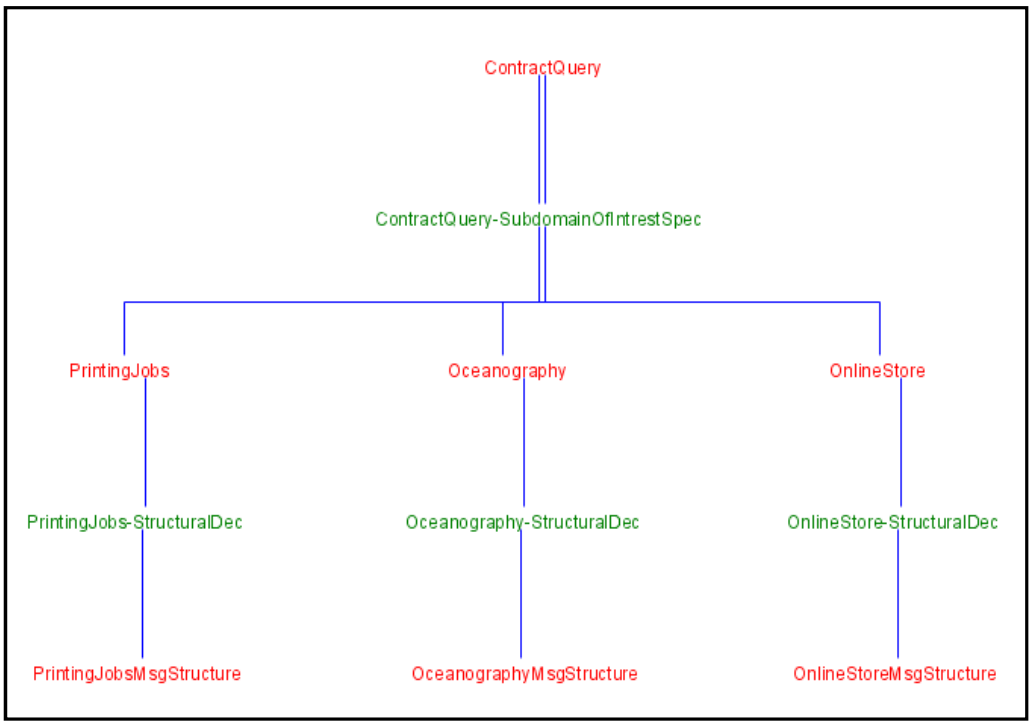


Figure 5.2.1: ContractQuery ontology tree

In the figure above, the message structure under PrintJobs domain consists of: PrintJob, TechnologyType, NoCopies, Deadline, Customer, PaperQuality, Duplex, PrintJobID, and Color. The message structure for Oceanography consists of: Speed, Roughness, Location, and Altitude. And for the OnlineStore we chose the structure to

have: SandH, BuyerID, Price, Return, and SellerID. In order to implement the dynamic message structure ontology, we used System Entity Structure formalism. SES is a useful ontological framework to define data engineering ontologies. In SES, Entities represent things that exist in the real world or in the imagined world. Aspects represent ways of decomposing things into more fine-grained ones [5]. In our ontology tree, the message type is an entity as well as the domain. SES has been applied to many different areas as a classification tool such as in [13]. More on XML and SES are discussed previously in chapter 4. Below is the XML document representation of the ontology in Figure 5.2.1.

----------XML Document ----------

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE entity SYSTEM "ses.dtd" []>
<entity  name = "ContractQuery">
<specialization  name = "ContractQuery-SubdomainOfIntrestSpec">
        <entity  name = "PrintingJobs">
                <aspect  name = "PrintingJobs-StructuralDec">
                        <entity  name = "PrintingJobsMsgStructure">
                                <var  name = "Duplex">
                                </var>
                                <var  name = "Customer">
                                </var>
                                <var  name = "NoCopies">
                                </var>
                                <var  name = "PrintJobID">
                                </var>
                                <var  name = "PaperQuality">
                                </var>
                                <var  name = "Deadline">
                                </var>
```

```
                        <var  name = "PrintJob">
                        </var>
                        <var  name = "Color">
                        </var>
                        <var  name = "TechnologyType">
                        </var>
                </entity>
        </aspect>
</entity>
<entity  name = "Oceanography">
        <aspect  name = "Oceanography-StructuralDec">
                <entity  name = "OceanographyMsgStructure">
                        <var  name = "Altitude">
                        </var>
                        <var  name = "Speed">
                        </var>
                        <var  name = "Roughness">
                        </var>
                        <var  name = "Location">
                        </var>
                </entity>
        </aspect>
</entity>
<entity  name = "OnlineStore">
        <aspect  name = "OnlineStore-StructuralDec">
                <entity  name = "OnlineStoreMsgStructure">
                        <var  name = "BuyerID">
                        </var>
                        <var  name = "SellerID">
                        </var>
                        <var  name = "Price">
                        </var>
```

```
                                <var  name = "Return">
                                </var>
                                <var  name = "SandH">
                                </var>
                        </entity>
                </aspect>
        </entity>
</specialization>
</entity>
```

We used SES builder to design the message ontologies. The SES builder is an easy to use tool and provides many features. The input is a restricted natural language designed for the system entity structure framework purposes. More on the natural language forms and syntax can be found in [5] and on the website www.devsworld.org [42]. The natural language input that resulted in the above ontology for *ContratcQuery* message is as in the following figure.

ContractQuery can be PrintingJobs, Oceanography, or OnlineStore in SubdomainOfIntrest!

From the Structural perspective, the PrintingJobs is made of PrintingJobsMsgStructure!

From the Structural perspective, the Oceanography is made of OceanographyMsgStructure!

From the Structural perspective, the OnlineStore is made of OnlineStoreMsgStructure!

The PrintingJobsMsgStructure has PrintJob, TechnologyType, NoCopies, Deadline, Customer, PaperQuality, Duplex, PrintJobID, and Color!

The OceanographyMsgStructure has Speed, Roughness, Location, and Altitude!

The OnlineStoreMsgStructure has SandH, BuyerID, Price, Return, and SellerID!

Figure 5.2.2: Natural language input for ContractQuery message

For each of the messages types in the language of encounter, we have a text file similar to that in Figure 5.2.2 that represents the message structure. It is obvious that the natural language interface gives very satisfactory options to the humans to express their ontological specifications. The book by B.Zeigler and P. Hammonds on simulation-based data engineering gives more insights into the natural languages and its usages [5]. For more on SES and ontology design refers to [45].

As we have seen in section 5.1, the marketplace architecture is a domain-independent design, where the language of encounter ontology is a domain-dependent structure. Combining both methodology results in an automated powerful negotiation model that provides enough expressiveness power while enforcing negotiation protocols to capture different user agents behaviors. Figure 5.2.3 shows the big picture of the two methodologies.

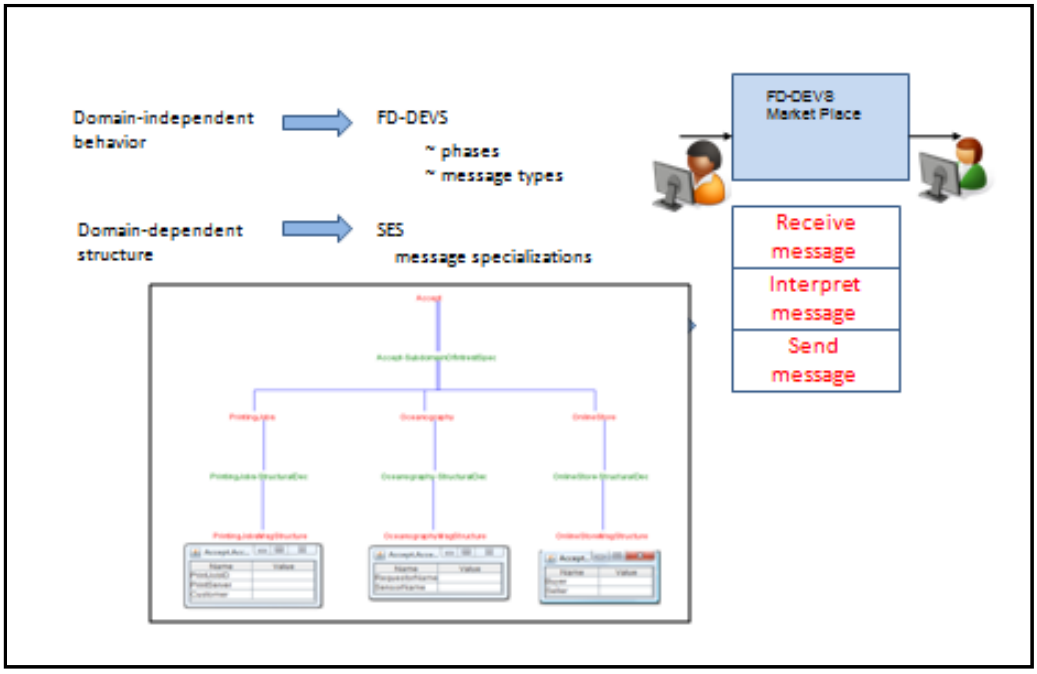


Figure 5.2.3: System negotiation modeling approach

5.3 Negotiation System Model Process Flow

In the previous two sections we explained the implementation of our approach using FD-DEVS and SES formalisms. In this section we will explain the negotiation system design process. Figure 5.3.1 shows the process flow of the negotiation model.

The system designer started the process by defining the domain-dependent message structure using a GUI tool that we implemented. The output of the GUI is a natural language for SES ontology structure where SES can be used to create the ontology representation in XML schema. The schema will be the input to the JAXB compiler which in turn results in Java classes defined for the domain of interest. Those

Java packages are ready to use but carry no information or data yet. The second pipeline in bottom starts by implementing our negotiation protocols (rules and requirements) in FD-DEVS specifications, which results in a generic domain-independent marketplace model. The tailored marketplace is a result of the designer choice of the domain of interest, more on that in the next chapter. The marketplace receives messages, interpret them by unwrapping them (unmarshal) and it might need to marshal them with data and send them. On the service provider side, the same scenario occurs.

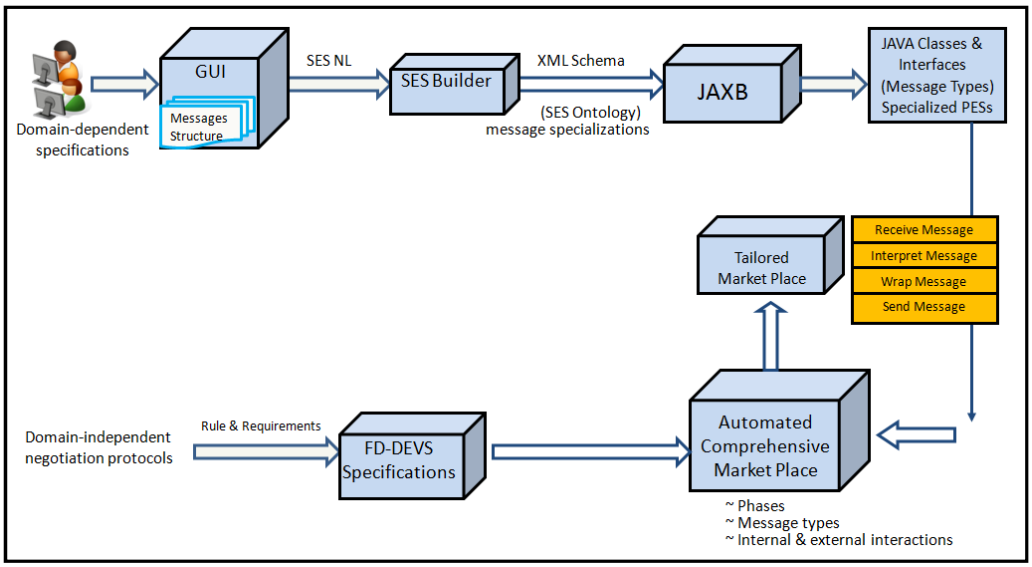


Figure 5.3.1: Negotiation model process flow

The process of unmarshalling and marshalling the language of encounter messages represented in Java classes between the requestors and the service providers is shown in Figure 5.3.2. On the service provider side, his data collections or services are represented in a pruned entity structures and XML instances. The pruned entity structures (PESs) are product descriptions such as (PrintJob = “Newspapers”,

TechnologyType=”Digital”, NoCopies=”1”, Deadline=”20”, Customer”RequestorName”, PaperQuality=”High”, Duplex”yes”, PrintJobID=”15382”, and Color=”BlackandWhite”). These variables are encoded in XML instances in the same formats of the XML schema for ContractQuery message. When the service provider uses JAXB data binding “unmarshaller” of the PESs on an empty *ContractQuery* message class, the returned message will be a *ContractQuery* with the above data inserted in the corresponding slots of the *domainMsgStructure* entity; after that, the message can be exchanged between agents

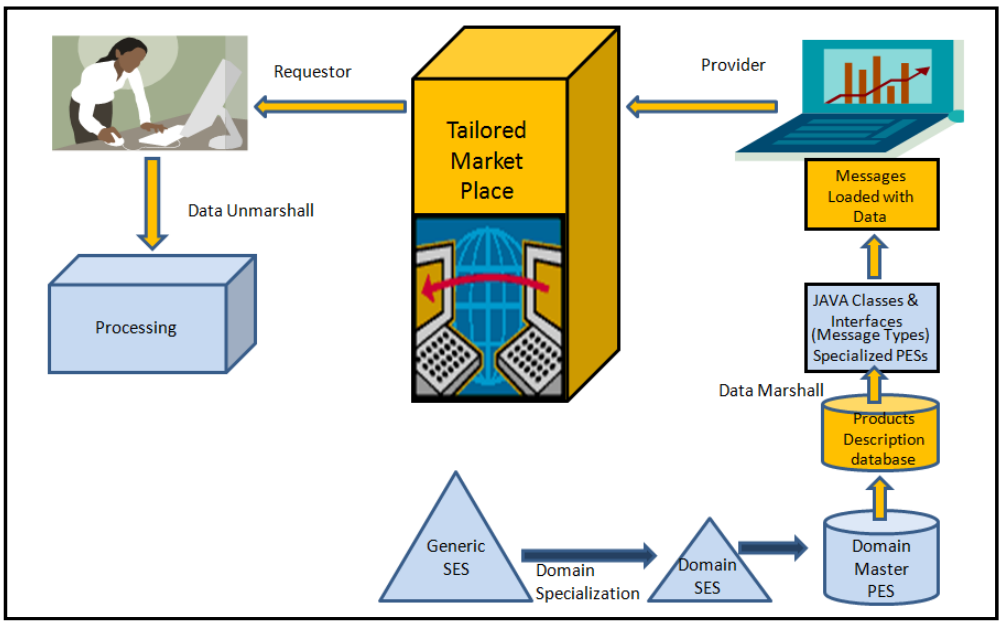


Figure 5.3.2: Unmarshalling and marshalling process between service providers and the service requestors.

CHAPTER 6. AUTOMATIC MARKETPLACE GENERATION FOR A SPECIFIC DOMAIN OF INTEREST

6.1 Steps in the Marketplace Generation

Designing the negotiation system is a very time consuming task which consists of many steps. We divided the steps here into two groups. The first group is regarding defining the dynamic message structure ontology. The second group is for designing the Marketplace phases, transitions and output in FD-DEVS formalisms. To design the language of encounter ontology for a specific domain, the system designer needs to follow the following steps:

1. Writing an SES natural language that describes the language of encounter' ontologies. This requires from the designer to write each message structure for each specific domain.
2. Using SES builder tool which was developed in our LAB (ACIMS LAB) [51], to create the ontology structure in SES XML schemas. The SES builder is an efficient tool for Knowledge Representation and data engineering and ontology design [26]. SES builder is also useful to prune SES XML files. For more information on pruning SES refer to [5].
3. The result of the second step associates each negotiation primitive with a SES schema. Java Architecture for XML Binding (JAXB) allows users to map Java

classes into XML representations and vice versa [32]. JAXB compiler takes XML schemas as inputs and produces Java classes and interfaces [33]. The negotiation system designer can use the JAXB compiler to create negotiation messages packages that can be plugged directly into Java files (our objective is to use them in the Marketplace implementation).

4. The output packages of the JAXB compiler can be used now in the Marketplace Java file.

To create the Marketplace negotiation protocols in FD-DEVS, the designer can use the FD-DEVS GUI tool, which is a useful tool to generate Java templates [27], to create the Marketplace states and transition specifications. The following steps are to be performed by the designer:

1. Use FD-DEVS GUI to define the Marketplace phases, the internal function and the external function tables. The tool will result in two files. One is an XML representation of the model and the second is a Java file.
2. Take the Java file which is a domain-independent generic marketplace template for the negotiation protocols.

In order to integrate the language of encounter Java packages with the domain-independent marketplace, the following steps must be carried out:

1. Importing the specific domain message classes into the marketplace model. For example, if the designer is developing an Oceanography domain negotiation system, then he must import the specific messages for the Oceanography domain. If another designer wants to develop online store negotiation system, then he much import the negotiation messages for OnlineStore domain. As a result, based on the domain of interest, the designer must manually import the same domain messages packages.
2. Remove the messages definitions of the generic marketplace model and define new messages classes based on step 1.
3. Unwrap messages classes and wrap them in the *deltext* method and the *out* method in order to provide the capabilities of sending data or receiving data (to be able to use using Set*variable* and Get*variable* methods).
4. The phase *ProcessingCapability* suggests that the marketplace receives a *CapabilityQuery* to find the appropriate providers for a specific job. Hence, the marketplace needs to access its database (in the form of pruned XML files) to unmarshal data in order to send them back to the requestor via a *CapabilityStatement* message. The designer must handle this process by adding the correct JAXB Unmarshalling code.

Figure 6.1.1 shows the flow of the manual steps that the negotiation system designer needs to follow. The figure shows five time consuming and tedious human interaction

tasks that each designer needs to go through before he starts tuning up dynamic coupling and decoupling in the hierarchical model. Writing SES natural language needs a lot of careful from doing syntax errors, in addition that each message in the language of encounter needs a separate SES natural language, which results in 17 different text files. The second step needs to import each of the 17 SES text files into the SES builder and create the SES XML schema. The third step will need a 17 system commands for each of the SES schemas to convert them into Java packages using JAXB compiler. In order to import a domain specific message structure, we need to write in the header file of the Marketplace Java file many lines of codes to import the correct messages. In the last step, a lot of work needs to be done. Unwrapping each of the messages in the *deltext* method and wrapping each message in the *out* method consumes a lot of time and effort.

The following section will show how we automate all these tedious and time consuming steps by developing a code generation tool that does most of the work on behalf of the designer. The tool reduces the human interactions into two very simple inputs from the designer. The designer then can do little of work tuning on the Marketplace model.

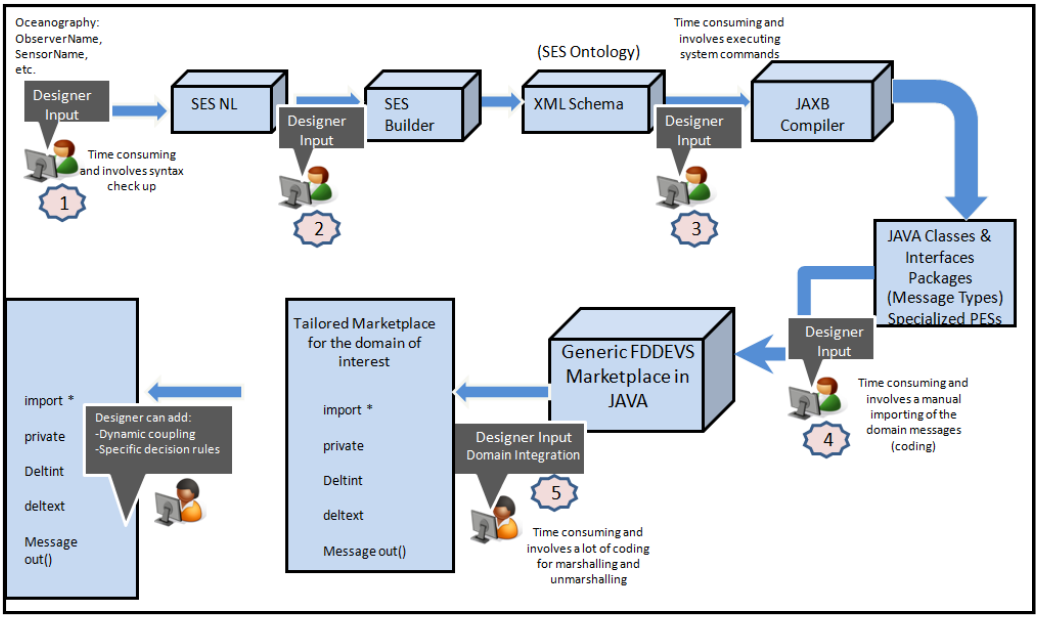


Figure 6.1.1: Manual steps in generating the negotiation system for a specific domain

6.2 Automatic Generation and Integration of the Negotiation Marketplace

In order to help the designer in defining the message structures, we have developed a simple, easy to use Graphical User Interface shown in Figure 6.2.1. The user of the GUI can add any subdomain he is interested in (for example PrintgJobs is a sub domain of domain Services). Then the tool asks the user to enter information about each message of the language of encounter. For example, it will ask how many fields does message “Accept” has, what are the names of each of them. The user or the designer might select two fields: CustomerName and PrintServerName. Then it will ask about the next message in the language of encounter and so on until all of the messages are defined.

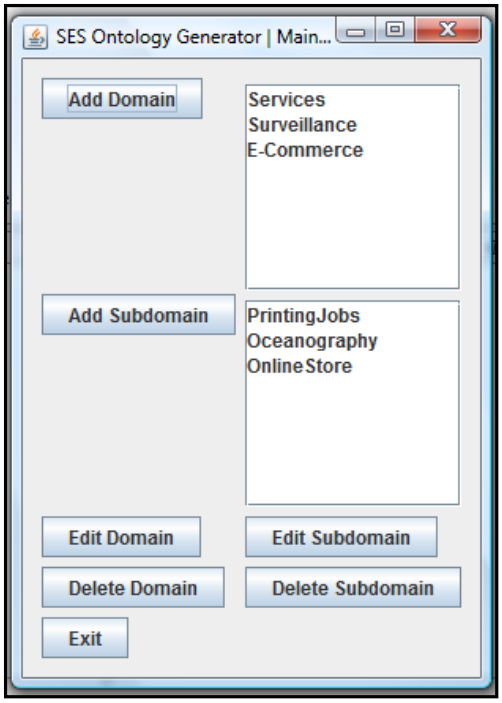


Figure 6.2.1: SES ontology creation GUI

The output of the SES ontology generation GUI tool is a collection of SES natural language text files, one for each message type. An example was given before. With the help of the source code of the SES builder we automatically generate an XML representation of the SES natural language by running the code:

sesinxml = NatToXml.getXML(SESnaturallanguagetext);

where SESnaturallanguagetext is the SES natural language and the NatToXxml.getXML is a method in the class NatToXml that generate an XML representation of a natural language input.

Then we convert the XML representation (String sesinxml) into an XML schema by running the line of code:

String schema = XmlToSes.getSchema(sesinxml);

The returned value of the above code is a schema stored in as a String variable. Then the tool writes each of the schemas of the messages into files with the extension ".xsd" to prepare them for the JAXB compiler to create the target Java packages. The SES schema is the representation of a master Ontology that contains all domains defined so far and pruned with their structures. An example is given in figure 6.2.2. If we add a new domain (say PrintingJobs) to the ontology it will be added automatically as a new specialization of the sub domains as in Figure 6.2.3.

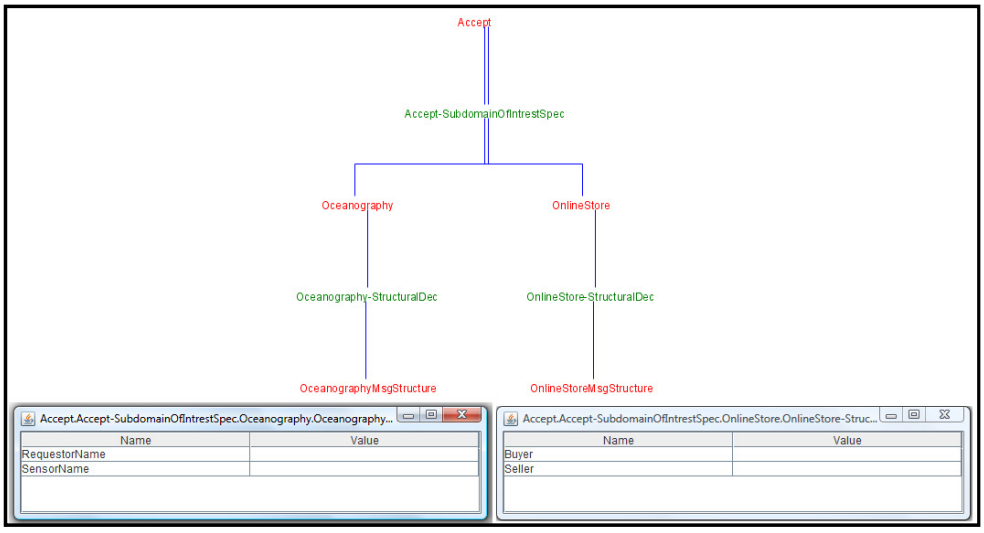


Figure 6.2.2: Accept message structure for Oceanography and OnlineStore

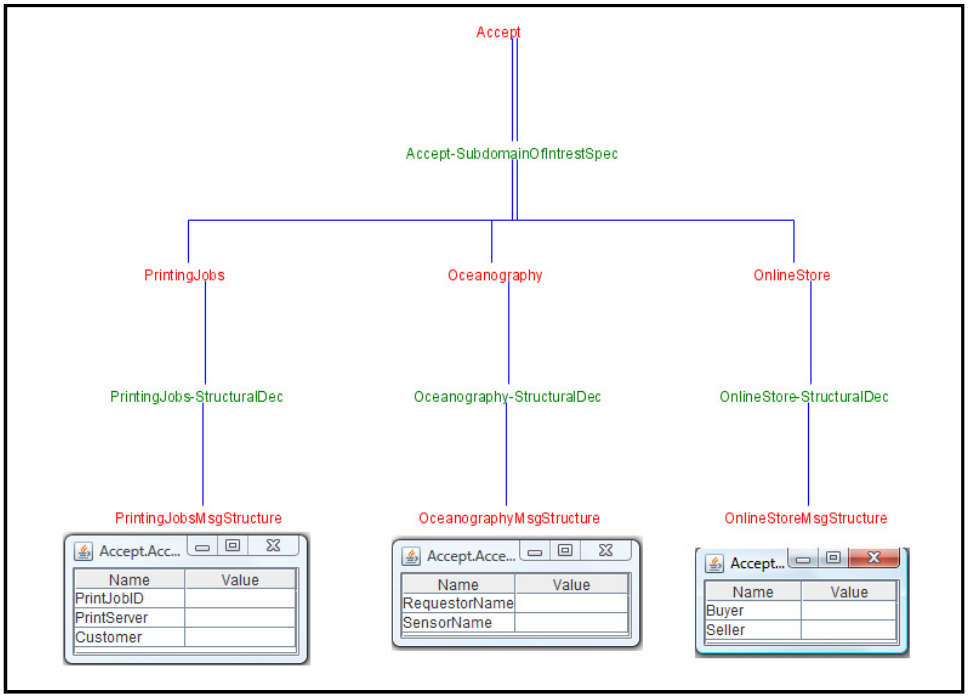


Figure 6.2.3: Accept message structure after adding PrintingJobs

Translation into Java Classes

The system syntax command for JAXB compiler is:

```
xjc schema.xsd –d dirName –p PackageName
```

xjc is the JAXB compiler.

schema.xsd is the SES structure representation in XML schema.

-d dirName is the name of the directory where to output the Java classes.

-p PackageName is the name of the Java files package name.

The method ExecJAXBSchemaCompiler as shown in Figure 6.2.4, executes the appropriate system command on each of the elements in the Set schemas (where each elements in the Set represents a message representation). The schemas files ("*.xsd") are saved under "currentPath/Messages/" and the output package name is NegotiationMessages. At the end of the method, a call to the method PostProcessingJavaClasses is needed. This method makes extends (*derived class*) each of the messages of class "*entity*" which is the base class for message exchanging in DEVS JAVA. Also it imports the package ("*import GenCol.*;*"). At this point, the Java packages are complete and can be used by the Marketplace Java model to declare the appropriate messages for the specific domain of interest.

Tailor of FD-DEVS for a Specific Domain

The second step that needs human interaction is very simple and all what it needs is to call a Java method (namely CreateFDDEVSModelFor) with the domain of interest as a String input such as "Oceanography or PrintingJobs". Since our Marketplace architecture is standard and implements the negotiation protocols we defined early, one time definition of the states, deltint table and deltext table in the FDDEVS GUI tool is enough. The XML model representation is very important can be stored somewhere for the tool to access. The in ports and out ports of the Marketplace model are also defined in the XML file. So the XML model file is an input also to the Java method CreateFDDEVSModelFor. Hence the standard calling of the Java method is as follows:

```
CreateFDDEVSModelFor("PrintingJobs");
```

where the MarketPlace.xml is the standardized design of the Marketplace model. Executing the line of the code above generates a tailored Marketplace Java model for PrintingJobs domain. The model has PrintingJobs language of encounter structure classes and ready to be used. Similarly for Oceanography domain:

CreateFDDEVSModelFor("Oceanography");

```
public void ExecJAXBSchemaCompiler( Set schemas ){
            Iterator itrschema = schemas.iterator();
             while(itrschema.hasNext()){
               String schema = (String)itrschema.next();
                   String[] command = new String[6];
                     command[0] = "xjc";
                      command[1] = "./Messages/" + schema + ".xsd";
                      command[2] = "-d";
                     command[3] = "./src/";
                     command[4] = "-p";
                      command[5] = "NegotiationMessages." + schema;
                    try{
                                Runtime.getRuntime().exec(command);
                                }
                    catch (Throwable t)
                       {
                         t.printStackTrace();
                       }         }
     PostProcessingJavaClasses(schemas);
  }
```

Figure 6.2.4: Class ExecJAXBSchemaCompiler to execute the compilation commands

Figure 6.2.6 shows the implementation of method CreateFDDEVSModelFor. The method is using class *AtomicFDD*, which is a class used in FDDEVS GUI that was developed in our Lab. The class is modified to create the following code generation steps for any negotiation Marketplace model:

1. Import the required Java classes for the negotiation process to take place, and the appropriate message package for the domain of interest.
2. Declare an instance of each of the negotiation primitives (language of encounter message) as shown in Figure 6.2.5.
3. In the deltext method, get message X when received on the corresponding in port "inX" that was designed to receive messages of type X. After receiving a message, store it in the corresponding local variable produced in step 2 and then generate the appropriate code to unwrap the message to get DomainMsgStructure class that has the get and set methods to allow the designers to access the data received or set variables to be sent into a message. The objective of storing the messages into local variables provides the capabilities for future data accesses and processing.
4. Create the JAXB Unmarshaller code to provide the Marketplace to access its database during the phase *ProcessingCapability*.
5. Prepare DomainMsgStructure classes and wrapping them into the corresponding language of encounter primitive; and then send it through the appropriate out port. This step simplifies the designer job into adding setV methods to marshal the messages with data that he would like to send.

```
private Accept accept;
private BestProvider bestprovider;
private Busy busy;
private CapabilityQuery capabilityquery
private CapabilityStatement capabilitystatement
private ContractQuery contractquery;
private CounterOffer counteroffer;
private Decline decline;
private Item item
private ItemCheckResult itemcheckresult;
private ItemRequest itemrequest;
private LinkEstablished linkestablished;
private NotMet notmet;
private Offer offer;
private ProvidersChosen providerschosen;
private Reject reject;
private Terminate terminate;
```

Figure 6.2.5: Local messages declaration variables for the marketplace model

```
public void CreateFDDEVSModelFor( String thesubdomain, String atomicXMLFile){
    String filename = atomicXMLFile;
   AtomicFDD atomicFDD;
   try {  AtomicJAXB atJaxb = new AtomicJAXB();
      atJaxb.initializeModel(filename);
      atomicFDD = atJaxb.atomicFDD;
   }
   finally {
   }
   if(atomicFDD != null){
      GenerateLanguageofEncounter();
      atomicFDD.generateDEVSModel("", LanguageofEncounter, thesubdomain);
      atomicFDD.writeDevsjavaModel("./models/java/");
   }      }
```

Figure 6.2.6: Class CreateFDDEVSModelFor for the domain of interest

Summary

In this chapter we showed how we automated the process of generating the Marketplace model given a message type of the language of encounter and the domain of interest. For example, if the message is "*ContractQuery*" and the domain is "*Oceanography*", the tool will select the pruned sub SES of the *ContractQuery* ontology that defines the message structure under the domain *Oceanography*. The overall

automated pipeline of the Marketplace generation is shown in the figure below (Figure 6.2.7).

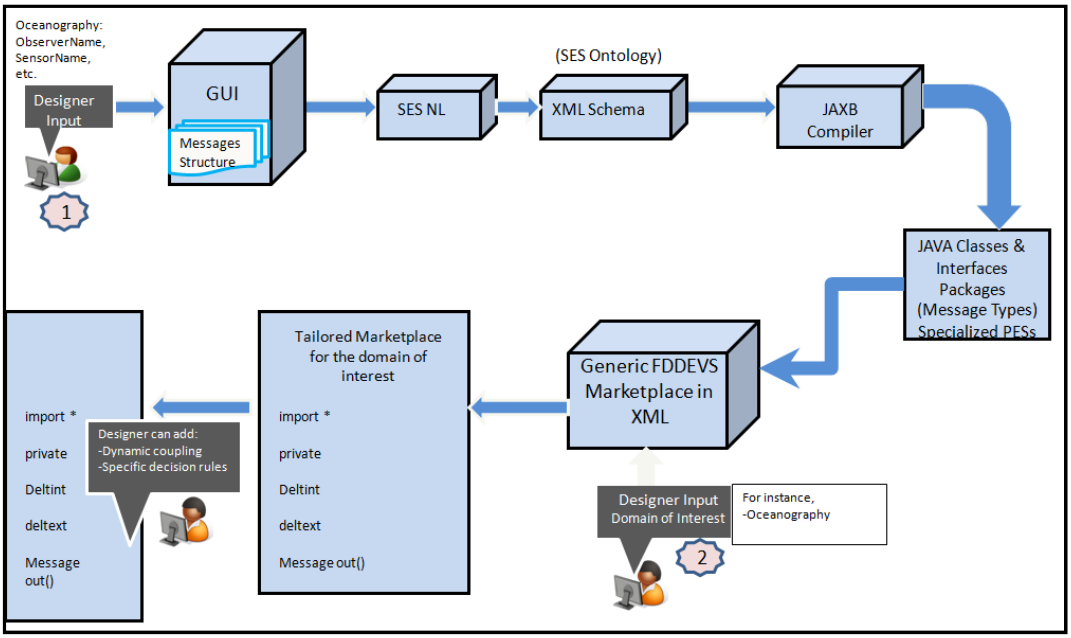


Figure 6.2.7: Marketplace generation flow

CHAPTER 7. EXPERIMENTS AND RESULTS

The application of the negotiation activity can be applied into many multi-agent disciplines where a user or an agent initiates the process by asking a query or a request to be fulfilled. The user seeks to find either the best provider for his request or a provider that can meet his requirements. In this chapter we will show two scenarios of interactions where the negotiation model is an essential to the success of requirements fulfillment. The first experiment is concerning surveillance systems in which observers negotiate with active or passive sensors to find the right sensor that can provide the right data and measurements over a specific region. The marketplace intermediates the interactions to find out the best data provider on behalf of the observer. The second experiment occurs very frequently in distributed engineering applications. A user or an engineer tries to find computing resources, where he can deploy his jobs and gets responses from the service provider within a specific deadline. The marketplace helps all negotiating party to reach an agreement. These examples show that the service provider can change dynamically and also how dynamic coupling and decoupling can be added in DEVS environment with the appropriate provider.

7.1 Oceanography in Surveillance domain

The problem of finding the best data source has been widely studied in the research. The objective is to find either the shortest path or the most efficient solution which takes into account Link Bandwidth, how fast does the source process data, etc. In

this section, we will focus on how a requestor of data can find the right data provider for his specifications and how the data providers can be selected dynamically over time. The marketplace permits requestors to communicate with the appropriate data providers based on its database records. Also, the marketplace can decide on behalf of the requestor on who is the best provider. Such a situation occurs if the designer of the domain implements some decision making to compare different offers from the service providers to pick the best out of them. On the other hand, in most of the situations, the decision making is made by the user. However, in this example, the marketplace receives *Reject* and/or *Accept* messages, and then it chooses the best of them. In the next section on distributed services environments, we will show how the marketplace receives *Offer*s messages and routes them to the correct destination (requestor). No decision making will be made by the marketplace except in finding the appropriate service providers.

We applied our system to the Oceanography field in surveillance systems in which experts observe different kinds of nature phenomena that might occur in the ocean. Monitoring the sea level is critical in order to be prepared for any of destruction phenomenon that could affect our cities and maybe causing a terrible impact on our life such as in Tsunami effects. Many authorities and governments have radars and sensors collecting data above the oceans all day time trying to detect any Oil slicks, Tsunami, earthquakes, volcanoes activities, etc. Sensors are divided into two types: namely *active sensors* and *passive sensors* [55] [62]. Passive sensors depend on the solar radiation; they can detect different object properties such as reflections, roughness of the surface, speed. However, passive sensors cannot measure the distance to the objects or the sea level. On

the other hand, active sensors are independent of the solar radiation; they operate by sending different wavelengths and detect how much of the waves are reflected from the object. This feature gives them the ability to measure the distances to objects [59]. Active sensors are capable of measuring sea level and can be used to detect the changes that Tsunami can cause on the ocean level. For more details on Radar sensors and their operational specifications refer to the European Southern Observatory (ESO) site [63].

In this experiment we will show that our negotiation model can provide observers the required capabilities to discover, locate and establish data links with the appropriate sensors. After that, data and information can be exchanged.

7.1.1 Language of Encounter Structure

We have defined the message structure in the language of encounter ontology as shown in table 7.1.1. We compiled the schemas of each of the message types into a Java package and we named it OceanographyMessages. The table below shows that some of the messages carry no information other than its type, which is all what it is needed for the marketplace to transit from one phase to another. Some messages carry information as needed by the experiment.

Message Type	Contents
Accept	SensorName, and RequestorName
BestProvider	SensorName
Busy	-
CapabilityQuery	AltitudeThreshold
CapabilityStatement	Sensors
ContractQuery	Speed, Roughness, Location, and Altitude
CounterOffer	-
Decline	-
Item	-
ItemCheckResult	-
ItemRequest	-
LinkEstablished	SensorName, and RequestorName
NotMet	-
Offer	-
ProvidersChosen	SensorsNames
Reject	-
Terminate	SensorName, and ObserverName

Table 7.1.1: Language of encounter structure for Oceanography domain

7.1.2 Observer Model

The Observer model starts the negotiation process in *ServiceDiscovery* phase causing the transmission of a *CapabilityQuery* message to the Marketplace asking if any of the sensors can provide a sea level altitude greater than a pre-defined threshold. The marketplace replies by sending the names of the sensors who can provide such data (need to be an active sensor type). After receiving the *CapabilityStatement* with the names of the sensor from the marketplace, the Observer model transits into *IssueContract* and marshals his specifications in a *ContractQuery* message and sends it to the marketplace. The *ContractQuery* will contain the different types of data that the Observer is interested in (namely Speed, Roughness, Location and Altitude). The marketplace then informs the Observer of the best provider sensor by sending a *BestProvider* message to it. After knowing the best provider, the Observer issues a *LinkEstablished* message asking the marketplace to setup a communication channel with the chosen data sensor. Then the sensor starts sending data periodically to the Observer until the collected data does not meet the specifications (this occurs when the altitude is less than the threshold). Once the dedicated sensor announces that he does not have the appropriate data. The Sensor will ask the marketplace to terminate the channel to the Observer, after that the Observer starts a new cycle looking for the next best provider. Figure 7.1.2.1 shows the scenario.

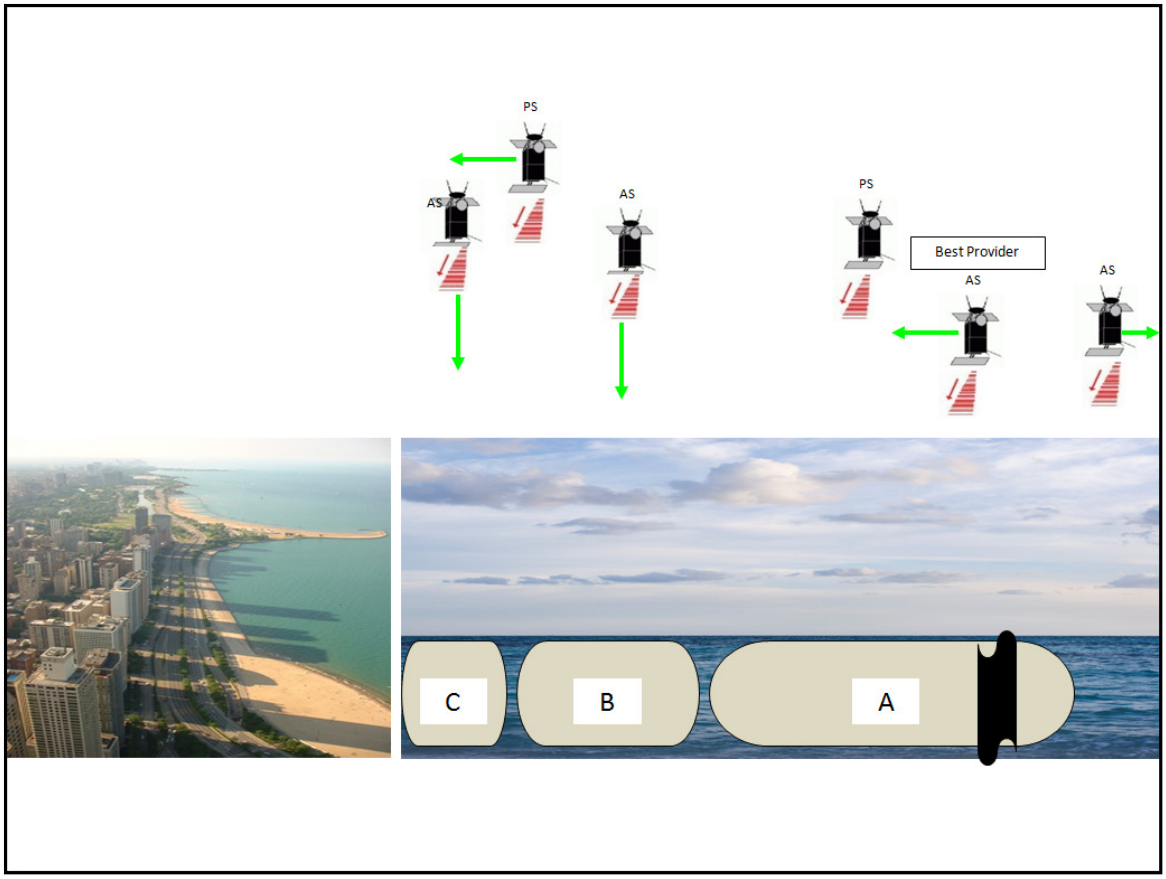


Figure 7.1.2.1: Oceanography best provider changes over time

In this example, we assumed that there are three regions on the ocean, region A, region B and region C. in region A, the sea level is above the threshold, at that time, Sensor 1 is the best provider of the data and he can provide it for while because he is covering a large region (A). In region B, Sensor 2 is the best provider. Since the waves move forward leaving the angle view of Sensor 2 at region C, then Sensor 3 becomes the next best provider. Figure 7.1.2.2 shows the atomic model of the Observer along with its input ports and output ports. Figure 7.1.2.3 shows the state transition diagram.

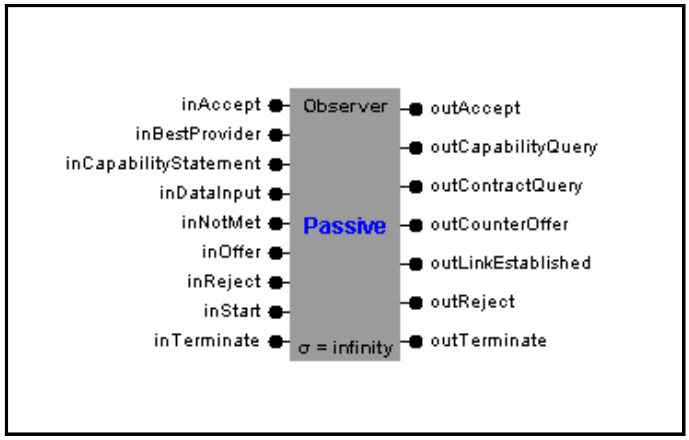


Figure 7.1.2.2: Observer atomic model

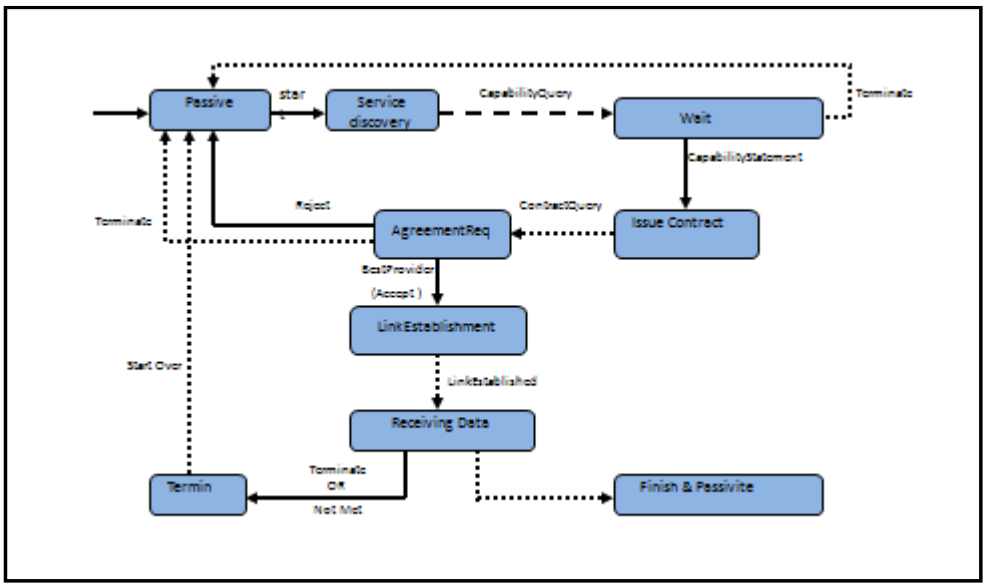


Figure 7.1.2.3: Observer state transition diagram

7.1.3 Marketplace Model

The Marketplace receives a *CapabilityQuery* from the Observer to find out the sensors who are capable of measuring the sea level altitude. The Marketplace replies with the active sensors names since all of them can provide altitude measurements. Then it forwards the *ContractQuery* message to the same chosen sensors in the *CapabilityStatement* which is the output of *ProcessingCapability* phase. After that it waits to receive from the Sensors either: *Accept*, *Reject* or *Offer* messages. In this experiment we have three active sensors, Active Sensor 1, Active Sensor 2 and Active Sensor 3. If it receives two *Reject*s and one *Accept*, then it will choose the one who responded with *Accept* as the best provider and sends its name in a BestProvider message to the Observer. If it receives all Reject, it will send an empty *BestProvider*. If it receives more than one *Accept*, then it will send the last one who replied with *Accep*t as the best provider. Once the Observer receives a best provider the Marketplace will establish a link between them. When one of the two communicated parties sends a *Terminate*, the Marketplace handles that by removing the communication link between them. Figure 7.1.3.1 shows the atomic model of the Marketplace along with its input ports and output ports. Figure 7.1.3.2 shows the main state transitions for the Marketplace model for this experiment.

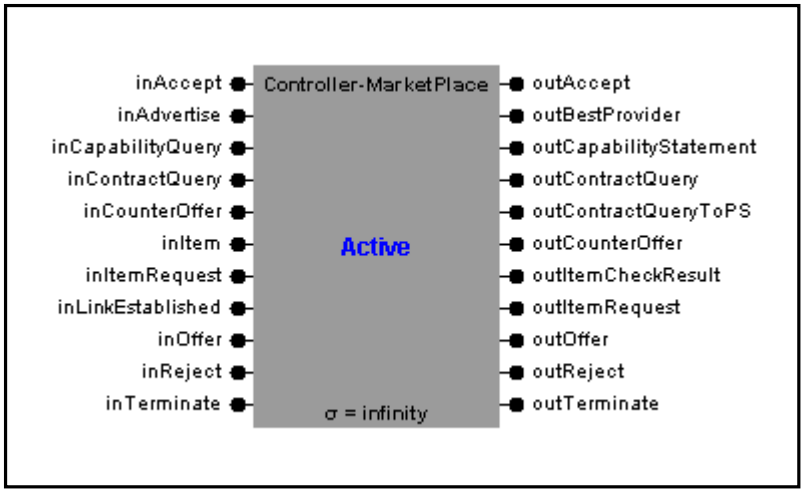


Figure 7.1.3.1: Marketplace atomic model

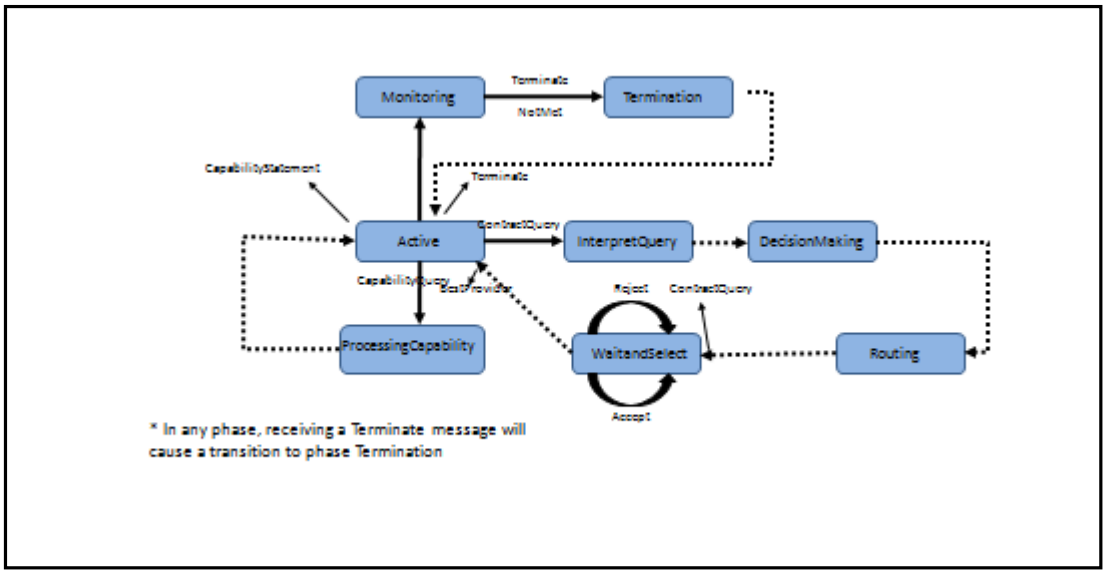


Figure 7.1.3.2: Marketplace main state transitions

7.1.4 Sensor Model

The Sensor model has database in the form of pruned SES files of the *ContractQuery*, which has fours variable as mentioned above (Speed, Roughness, Location, and Altitude). These data are a proposed data and not real, because of the lack of having real Radar sensors. However, the model gives useful insights on how collected datasets can be used; and no matter how the data is stored in the real sensors, it easily can be mapped into pruned SES files. The proposed XML files have time stamps based on the simulator clock. So, if the simulator clock is 55, then the sensors will access file "data55.0.xml" which is stored under the corresponding directory (Active Sensor 1 has directory "AS1"). When the sensors receive the *ContractQuery* message, they unmarshal their corresponding pruned XML files and check whether the variable "altitude >= Threshold". If the statement is true, the sensor will send *Accept*, otherwise it will send *Reject*.

If one of the sensors who responded with *Accept* is chosen as the best provider, the communication link will be established to it. After which it keeps retrieving the data from its own pruned XML files every 2 simulation clock and sending the data to the Observer model. The process proceeds as long as the data he is collecting is greater than the Threshold. Once the Altitude is less than the Threshold, the sensor will send *Terminate*. Figure 7.1.4.1 shows a pruned XML sample file for Active Sensor 1. Figure 7.1.4.2 shows the atomic model of the Sensor model and Figure 7.1.4.3 shows the state transition diagram.

```
<?xml version="1.0" encoding="UTF-8"?>
<ContractQuery xmlns:xsi =  "http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="xmlinSCH.xsd">
<ContractQuery-SubdomainOfIntrestSpec>
<Oceanography>
        <aspectsOfOceanography>
                <Oceanography-StructuralDec  coupling = "">
                        <OceanographyMsgStructure  Altitude = "8" Location = "20" Roughness = "30" Speed = "40">
                        </OceanographyMsgStructure>
                </Oceanography-StructuralDec>
        </aspectsOfOceanography>
</Oceanography>
</ContractQuery-SubdomainOfIntrestSpec>
</ContractQuery>
```

Figure 7.1.4.1: Pruned XML file for active sensor 1 -ContractQuery

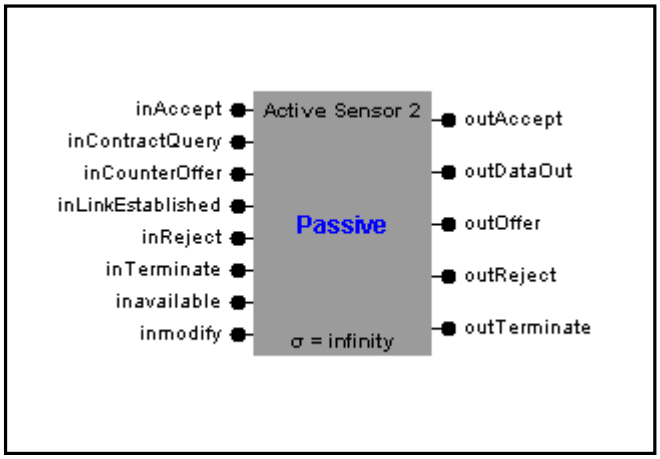


Figure 7.1.4.2: Sensor atomic model

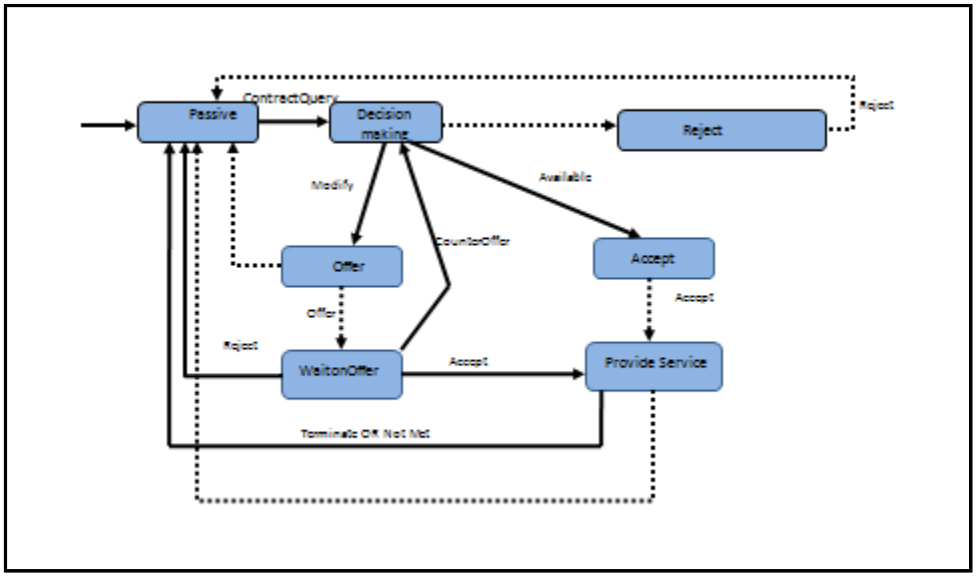


Figure 7.1.4.3: Sensor state transition diagram

7.1.5 Coupled Model and Simulation

Figure 7.1.5.1 shows the coupled model of the simulation. The system consists of: Observer model, Marketplace model, Passive Sensor 1, Passive Sensor 2, Active Sensor 1, Active Sensor 2 and Active Sensor 3. The simulation of the negotiation process results in the same behavior as we expected. From simulation time 22 until 70, Active Sensor 1 is the best provider and is chosen to be the data source for the Observer. From simulation time 94 until 116, Active Sensor 2 is the best provider and is chosen as the data source for the Observer. And finally, from simulation time 142 until 172, Active Sensor 3 is the best provider and the appropriate data source. The Figures 7.1.5.2 and 7.1.5.3 show snapshots of the simulation at running.

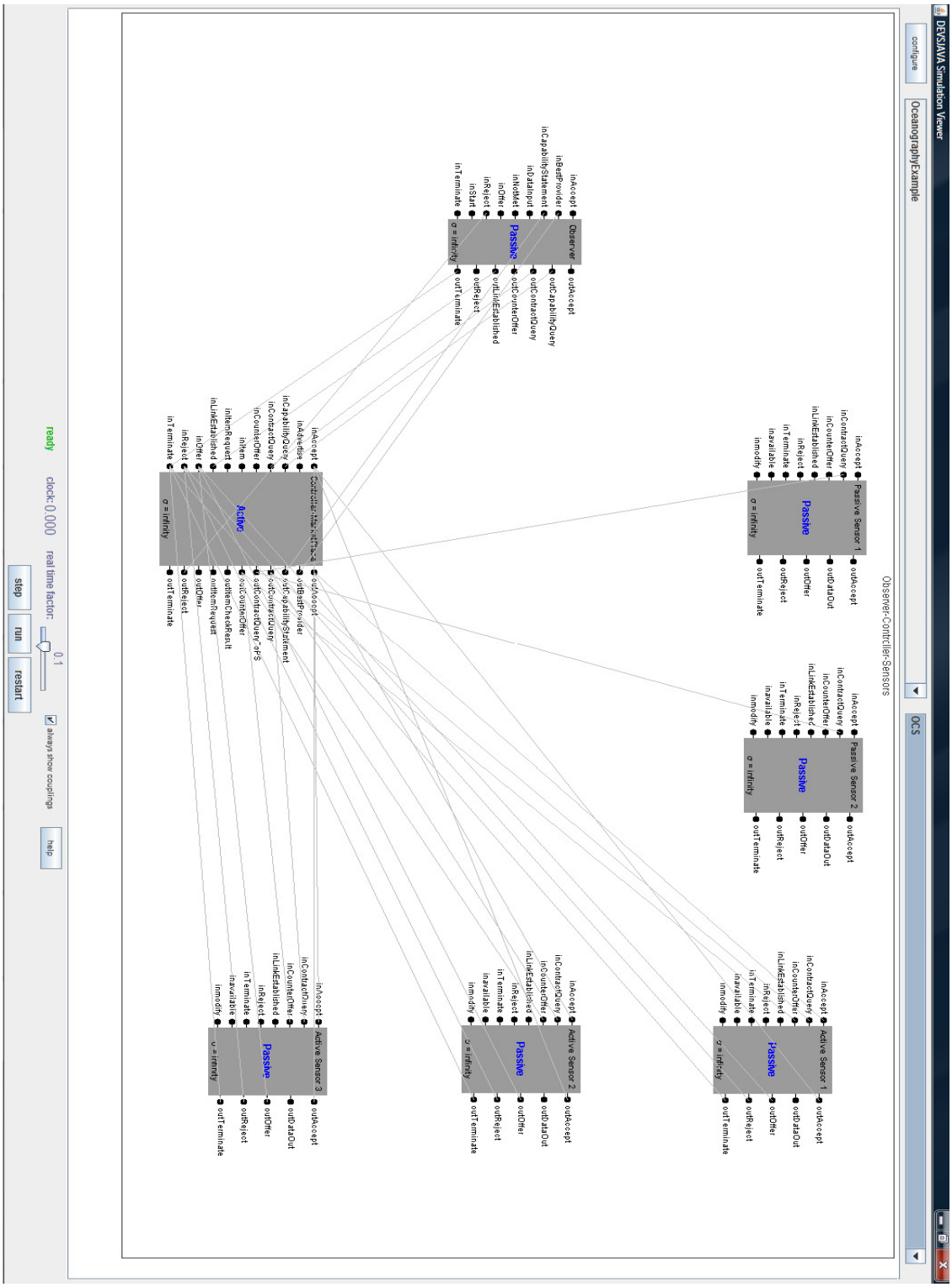


Figure 7.1.5.1: The coupled model

Figure 7.1.5.2: Routing ContractQuery to the active sensors

Figure 7.1.5.3: Active sensor 1 is the best provider and the data source

7.2 Distributed Services Environment

Exploiting service providers in a distributed services environment has been a tedious task to achieve. That is because of the fact that service providers are geographically distributed and loosely coupled [21]. Users or engineers have been always try to share computing resources because many of distributed systems are costly and expensive to design and maintain. Hence, whenever it is possible, different companies and other parties prefer to have software services that are optimally utilized where they can deploy their models and jobs on the grid on demand. In these environments, the users concern about different parameters such as the execution time, deadline until they get responses, the quality of the data they need, the solution efficiency.

Having the services distributed brings the following challenges into systems management techniques. First, users will need help from a third party to locate and find out the appropriate providers among many of them. Second, privacy and transparency where users do not like to publish their interests to every provider registered in a multi agent environment. Third, users do not want to waste time and money to discover their candidates. For example, in [15] an investment banking system based on web services have been discussed where semantic ontologies were developed to represent services in an attempt to close the gap and match between requesters and providers. The point here is that you have many distributed and loosely coupled investment systems and the users cannot locate the provider who can meet their requirements. As a result, a service model based on the semantics is used to make the users understand and choose their best match.

In this section, we will show printing jobs scenarios in which users sends different kinds of printing jobs and negotiate on different aspects of the job specifications until they reach an acceptable agreement within their range. The problem is very close into its definition to the problem of deploying computing jobs (or programs) into distributed computing grid. This scenario captures most of the issues that could be found in such engineering service environments.

7.2.1 Language of Encounter Structure

We used the GUI that we developed to define the structure of each of the messages in the language of encounter. The result of the automation tool is a Java package that we gave it the name PrintingJobsMessages. In designing the message structures for this domain, we chose some selections of the types and technologies in current printing servers. The following is a list of the printing technology along with their applications.

Digital Printing

- Brochures
- Journals
- Booklets

Embossing Printing

- Greeting Cards
- Metals
- Garments

Flexography Printing

- Milk and Beverage Cartons
- Disposable Cups
- Containers
- Adhesive Tapes
- Envelopes
- Newspapers
- Food and Candy Wrappers

Letterpress Printing

- Business Cards
- Company Letterhead
- Proofs
- Billheads
- Forms
- Posters
- Embossing

- Hot-leaf Stamping

Engraving Printing

- Stationery
- Wedding Cards
- Business Cards
- Letterhead

Gravure Printing

- Label
- Flexible Packaging
- Cartoning

Thermography Printing

- Fax Printers
- Business Cards
- Letter Head
- Invitation

For instance, if a customer is concerning with printing business cards, he might choose thermography, Engraving or Letterpress technology. Also, we defined different aspects for paper quality, deadline, color and duplex. The table below shows each message type and the contents/information that it carries.

Message Type	Contents
Accept	Customer, PrintServer, and PrintJobID
BestProvider	-
Busy	-
CapabilityQuery	PrintJob, and Customer
CapabilityStatement	PrintJob, and PrintServer
ContractQuery	PrintJob, TechnologyType, NoCopies, Deadline, Customer, PaperQuality, Duplex, PrintJobID, and Color
CounterOffer	PrintJob, TechnologyType, NoCopies, Deadline, Customer, PaperQuality, PrintServer, Duplex, PrintJobID, and Color
Decline	-
Item	-
ItemCheckResult	-
ItemRequest	-
LinkEstablished	Customer, and PrintServer
NotMet	-
Offer	PrintJob, TechnologyType, NoCopies, Deadline, Customer, PaperQuality, PrintServer, Duplex, PrintJobID, and Color
ProvidersChosen	-
Reject	Customer, PrintServer, and PrintJobID
Terminate	-

Table 7.2.1: Language of encounter structure for PrintingJobs domain

We assumed also that if a new printing server would like to join the printing services community, he should send a "*MyCapability*" message to the Marketplace to register himself. *MyCapability* message should contain at least the provider ID and name along with what printing capabilities he can provide such as: Business Cards, Wedding Cards.

7.2.2 User/Customer Model

The user of the printing services system starts the negotiation process by sending a service discovery request to the marketplace asking whether his job can be serviced by any of the printing servers. The marketplace replies with whoever can provide the service for that specific job, for instance, print server 3 provides Business Cards printing. After discovering the appropriate service providers, the user starts to negotiate with the selected providers by exchanging offers and counter offers on different printing attributes such as paper quality, color, the deadline to finish printing. Once an agreement is reached, the user will be satisfied with that specific job specification and sends *Accept*. In modeling such an interaction behavior, A DEVS Java model is developed with the following decision making rules.

- The User model is searching for a provider who has the Business Cards printing capability.
- The user would accept an offer if one of the following conditions is satisfied:

1. If the paper quality is medium or high, the color is full HD and the deadline is less than 80.
2. If the paper quality is medium or high, the color is RGB and deadline is less than 30.
3. If the paper quality is medium or high, the color is grayscale and the deadline is less than 20.

- If the offer does not match any of its acceptable ranges, the user sends back a counter offer asking either his first preference or a modified one based on the history of the offers he was receiving. In our model, we chose that the user sends his first preference.

Figure 7.2.2.1 shows the User/Customer atomic model along with its input ports and output ports.

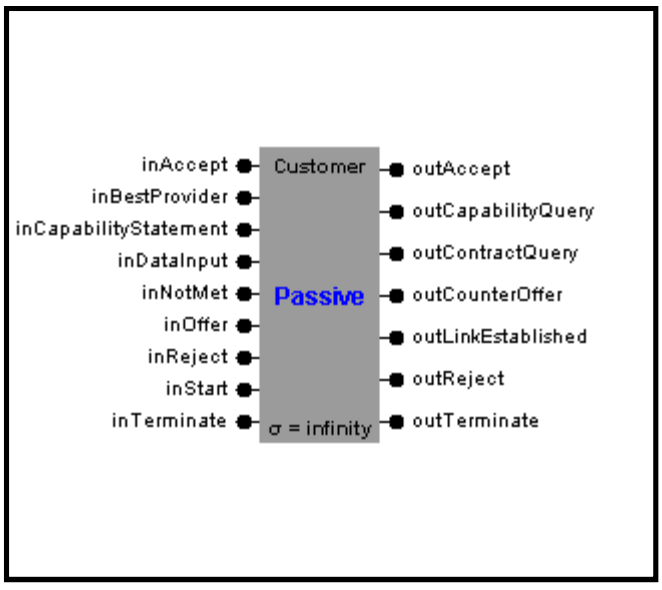


Figure 7.2.2.1: User/Customer atomic model

Figure 7.2.2.2 shows the state transitions. At the beginning a start message is injected into the User model causing it to transit into *ServiceDiscover* phase. In this phase, the User puts its printing job type and its name into a *CapabilityQuery* message and sends it to the Marketplace model at the end of the phase (internal transition). Then the User waits for a *CapabilityStatement* in phase *Wait*. After receiving the *CapabilityStatement*, it gets the selected providers for his job and transits to state *IssueContract*, where a *ContractQuery* message is prepared with different printing job specifications and attributes to be sent to the selected providers. Note here that if *CapabilityStatement* that the user has just received from the Marketplace does not contain any providers, then even if the User sends a *ContractQuery* message to the Marketplace it will not be routed to any of the providers since none of them supports the User requirements. The internal transition from *IssueContract* outputs *ContractQuery* to the Marketplace and the User goes into state *Agreement* waiting for an agreement with any of the appropriate providers. While the User in the *Agreement* phase, he will be receiving different *Offers* from the selected providers. It will wait in the *Agreement* state for a specific time (we selected it to be enough until all the providers complete sending their offers). The internal transition function causes the User to transit into *DecisionMaking* phase, in which it starts pulling each *Offer* he received and decide whether it meets his acceptable range or not. In this state, the User unmarshals the data he needs from the *Offer* message to help him decide on that offer, this include the different fields in the message such as: PaperQuality, Color, Deadline, TechnologyType. If the *Offer* does not meet his interests, the User goes into *IssueCounterOffer* state where the internal function

cause a *CounterOffer* message to be sent at the end of that phase to the source of that specific *Offer*. After sending all *CounterOffers* to the providers involved in the negotiation process, the User waits in state *Wait*. The internal transition function takes the User from *Wait* into *Agreement* again and the same cycles of *Offers –CounterOffers* proceeds until an acceptable *Offer* is detected. If the User receives an *Offer* that is acceptable to him, then during the *DecisionMaking* state the User will decide to transit to phase *Acceptance*. The internal transition from *Acceptance* causes a message *Accept* to be sent to the Marketplace and then to the provider who owns that *Offer*. Immediately after that, a transition to phase *LinkEstablishment* occurs. The internal transition from *LinkEstablishment* causes an output of message *LinkEstablished* to be sent to the Marketplace and the appropriate provider in order to inform them that the user is ready to receive the service. The User transits into *Receiving Data* until the provider processes his job and send him back an acknowledgment (*DataOut*) that he finished processing his job. Once the User is informed that his job is finished, he goes into *Termination* state causing message *Terminate* to be transmitted to the Marketplace.

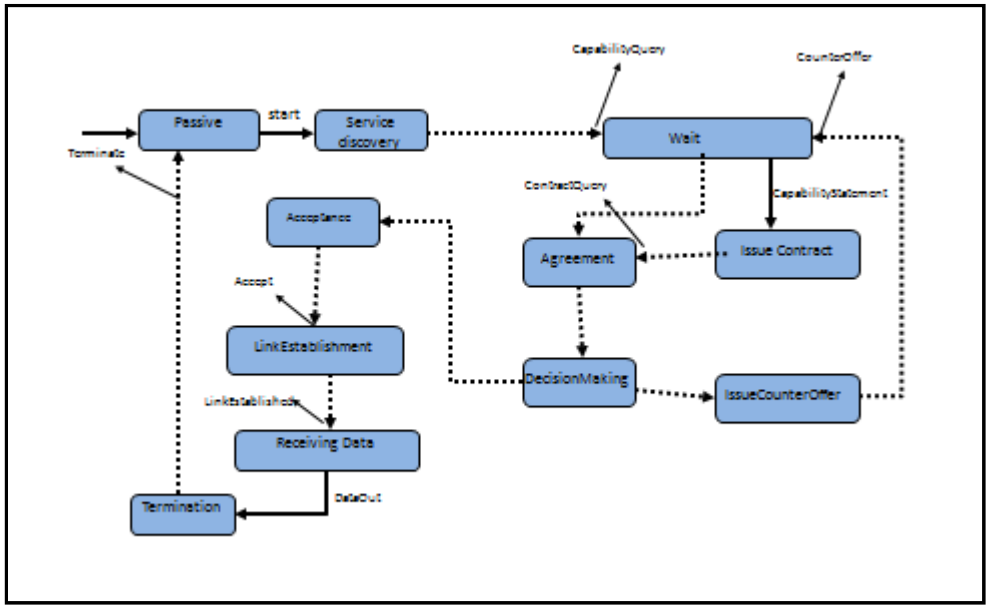


Figure 7.2.2.2: State diagram for User/Customer model

7.2.3 Marketplace Model

The generic automated Marketplace model is used here. However, we added two more functionalities to permit the Marketplace to intermediate the negotiation to enhance the performance and efficiency. The two functionalities are:

1- Dynamic coupling and decoupling to setup channels between the User model and the service providers based on the message source and destination. For example, if a *CounterOffer* is aimed to be delivered to Print Server 6, then a channel should exist between the User and the Print Server 6 to enable them of exchanging the messages. At the same time,

there is no need to have a coupling between Print Server 4 and the User since no *CounterOffer* with his name as a destination.

2- When receiving a *ContractQuery* message from the User to be forwarded to the appropriate providers. The Marketplace unmarshals it and adds a unique *PrintingJobID* field. The purpose of this field is to enable the Marketplace to keep track of all the jobs that goes between users and providers, and to differentiate between all of the jobs, it will be helpful to have the Marketplace adding a unique ID for each job in order for future purposes such as resolving an a agreement. For example, when a User complains about an agreement violation, the Marketplace can access its own database and find out the job that needs to be resolved.

The rest of the Marketplace behaviors follow the same rules and specifications as mentioned previously when we discussed the Marketplace architecture and its functionalities. We will point out here that when the Marketplace receives a *ContractQuery* from users, it will forward it to the appropriate providers based on their capabilities that were published in the past. After which the Marketplace waits for responses from the providers. When it receives offers from the providers, it routes them back to the destination of the Offer messages. Figure 7.2.3.1 shows the Marketplace atomic model along with its input ports and output ports.

The Marketplace database consists of XML files in the project path under directory “MarketplacePrunedDB”. These XML files contain the printing job type name

and the names of the providers who can provide that printing type. For example, Figure 7.2.3.2 shows a sample XML file for Business Cards printing types and the provider names which are: Print Server 1, Print Server 3 and Print Server 6.

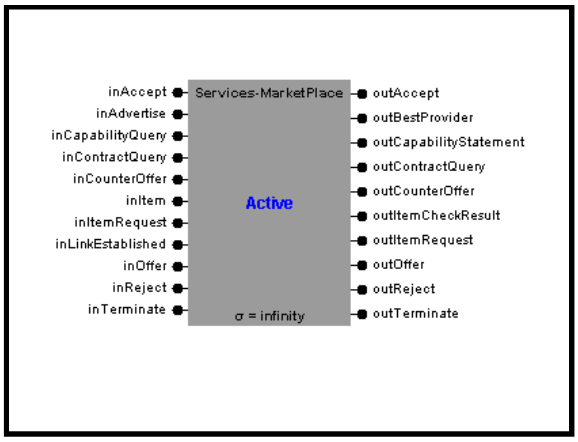


Figure 7.2.3.1: Marketplace atomic model

```
<?xml version="1.0" encoding="UTF-8"?>
<CapabilityStatement xmlns:xsi =  "http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="xmlinSCH.xsd">
<CapabilityStatement-SubdomainOfIntrestSpec>
<PrintingJobs>
      <aspectsOfPrintingJobs>
            <PrintingJobs-StructuralDec  coupling = "">
                  <PrintingJobsMsgStructure  PrintJob = "Business Cards" PrintServer = "Print Server 1;Print Server 3; Print Server 6">
                  </PrintingJobsMsgStructure>
            </PrintingJobs-StructuralDec>
      </aspectsOfPrintingJobs>
</PrintingJobs>
<Oceanography>
      <aspectsOfOceanography>
            <Oceanography-StructuralDec  coupling = "">
            <OceanographyMsgStructure/>
            </Oceanography-StructuralDec>
      </aspectsOfOceanography>
</Oceanography>
<OnlineStore>
      <aspectsOfOnlineStore>
            <OnlineStore-StructuralDec  coupling = "">
            <OnlineStoreMsgStructure/>
            </OnlineStore-StructuralDec>
      </aspectsOfOnlineStore>
</OnlineStore>
</CapabilityStatement-SubdomainOfIntrestSpec>
</CapabilityStatement>
```

Figure 7.2.3.2: Business Cards.xml file

7.2.4 Service Provider Model

The Print Server model or Service Provider accesses its own XML files database in the same way the Marketplace accesses its database. Each of the Print servers has different printing capabilities that are stored in the XML files, which are pruned SES files. For example, Print Server 1 has the capability to print Business Cards, Brochures, Newspapers and Posters. The specifications of each of these printing capabilities will be stored under the corresponding PES file for that printing capability; for example, "Business Cards.xml". We assume that each of the print servers can update or change on these specifications such as Deadline in order to match user requirements. The modifications process of the aspects follows some rules which were defined for each of the Print Servers models. The scope of this research is not on how the decision making occurs on the Print Server side or the user side. It could be a manual user interaction, or an automated mathematical model that captures the user objective function. Hence, in our implementation we have assumed some random updates on different printing jobs specifications, for instance, we used that CurrentDeadline = PreviousDeadline – Update.

If a new Print Server would like to join the printing services community, he sends a "MyCapability" message including his name and the printing capabilities he provides. Then the Marketplace will add him to its database along with his printing capabilities. When the Print Server model receives a *ContractQuery* message, he transits into *DecisionMaking* state, where a decision will be made on whether he can meet the requirements of the printing job in the *ContractQuery* message or not. If he can, then he will send *Accept* and an agreement will be reached. However, if he cannot meet the

customer specifications, he will send an *Offer* message to the Marketplace including his current offer and his name. The Marketplace receives the message, find out the customer name by unmarshalling the message, and then routes it to the appropriate receiver. The way we designed the decision making rules in this experiment is to show how negotiation cycles of *Offer-CounterOffer* occur. On the other hand, in the previous experiment as we explained, the decision making was direct with best provider chosen.

The internal transition function causes the transition from *DecisionMaking* to *Offering* phase, the output of *Offering* phase is an *Offer* message. After that, the Print Server model holds in *WaitonOffer* phase; in which the Print Server waits to receive *Accept*, *Reject* or *CounterOffer*. If he receives a *CounterOffer*, he goes into the same cycle of *DecisionMaking->Offering->WaitonOffer*, or he can accept and goes into *Acceptance* state which results into sending *Accept* message. In this implementation, we assumed that if a Print Server sends an *Offer* to a Customer and the customer accepts the offer, then an agreement is reached. No need to go back to the Print Server and asking him whether he accepts or no.

If the Print Server receives *Accept*, he will hold in state *ProvideService* for the time defined in the *Offer* Deadline. Internal transition causes the model to transits from *ProvideService* to *Passive* and an output of *DataOut* will be sent to the Customer informing him that the processing of his job has finished. Figure 7.2.4.1 shows the atomic model of the Print Server model (or Service Provider model) along with its input ports and output ports.

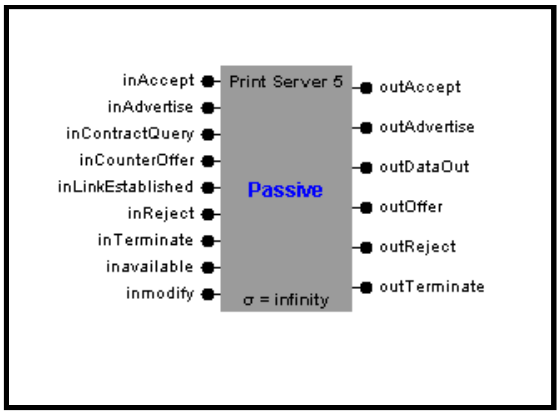


Figure 7.2.4.1: Print server atomic model

The state transition diagram of the Print Server is shown in Figure 7.2.4.2.

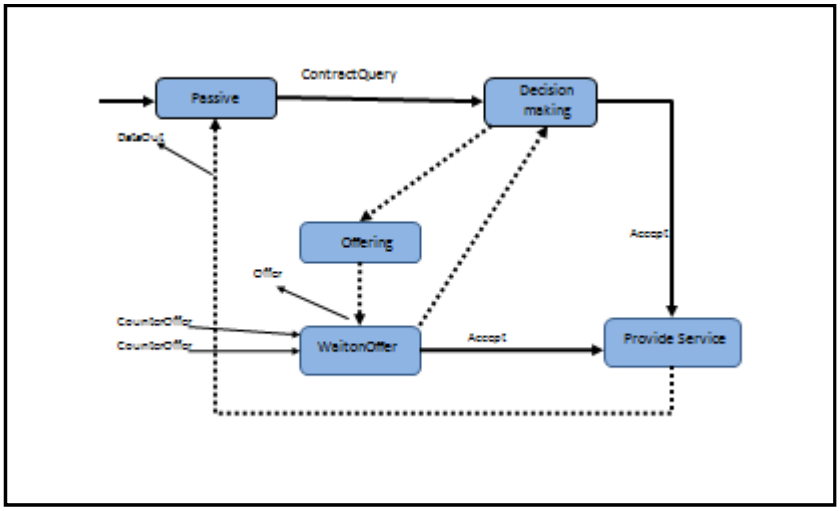


Figure 7.2.4.2: Print server state diagram

7.2.5 Coupled Model and the Simulation

Figure 7.2.5.1 shows the coupled model which is a higher hierarchical level of the atomic models. The output ports of the User model is connected to the input ports of the Marketplace. None of the output ports of the Marketplace model is connected to any of the input ports of the service providers. Where we aim to add the coupling or remove it dynamically based on the destination of the messages or the capabilities of the providers. For example, when a *ContractQuery* is received from the User model, the Marketplace add coupling with those of the providers who provides that printing service defined in the *ContractQuery* (Figure 7.2.5.2). Since we have in our simulation only one customer, we connected the output port “outCapabilityStatement" of the Marketplace to the User input port “inCapabilityStatement".

addCoupling(M,"outCapabilityStatement",U,"inCapabilityStatement");

We have seven Print Servers each of which has its own printing capabilities which are defined in his own PESs database. When the Marketplace needs to send a message to Print Server X, it adds coupling to it, sends him that message and removes the coupling unless its needed in the next step of the simulation. The Print Servers can easily send their messages to the Marketplace because their output ports are connected to the input ports of the Marketplace model. Print Servers models and the User models exchange their *Offers-CounterOffers* through the Marketplace model (Figure 7.2.5.3 and Figure 7.2.5.4).

If a Print Server model and the User model needs to communicate, they inform the Marketplace and then the Marketplace add the required coupling permitting them to negotiate. This situation occurs when they reach an agreement, the customer will ask for a link to be established resulting in the Marketplace adding a link between the two parties of the agreement. The link will be removed once the job processing is done (Figure 7.2.5.5 and Figure 7.2.5.6).

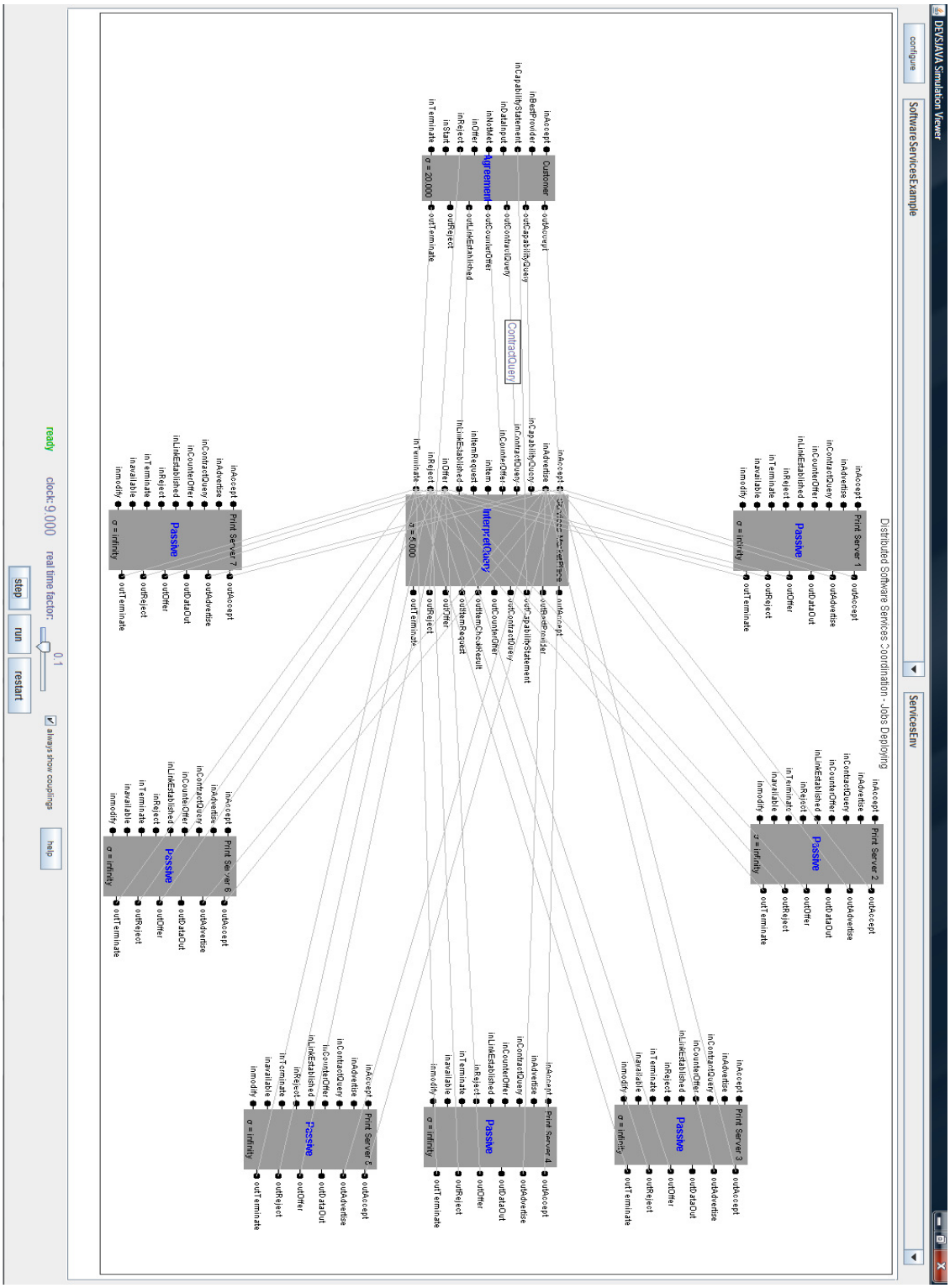


Figure 7.2.5.1: PrintingJobs coupled model

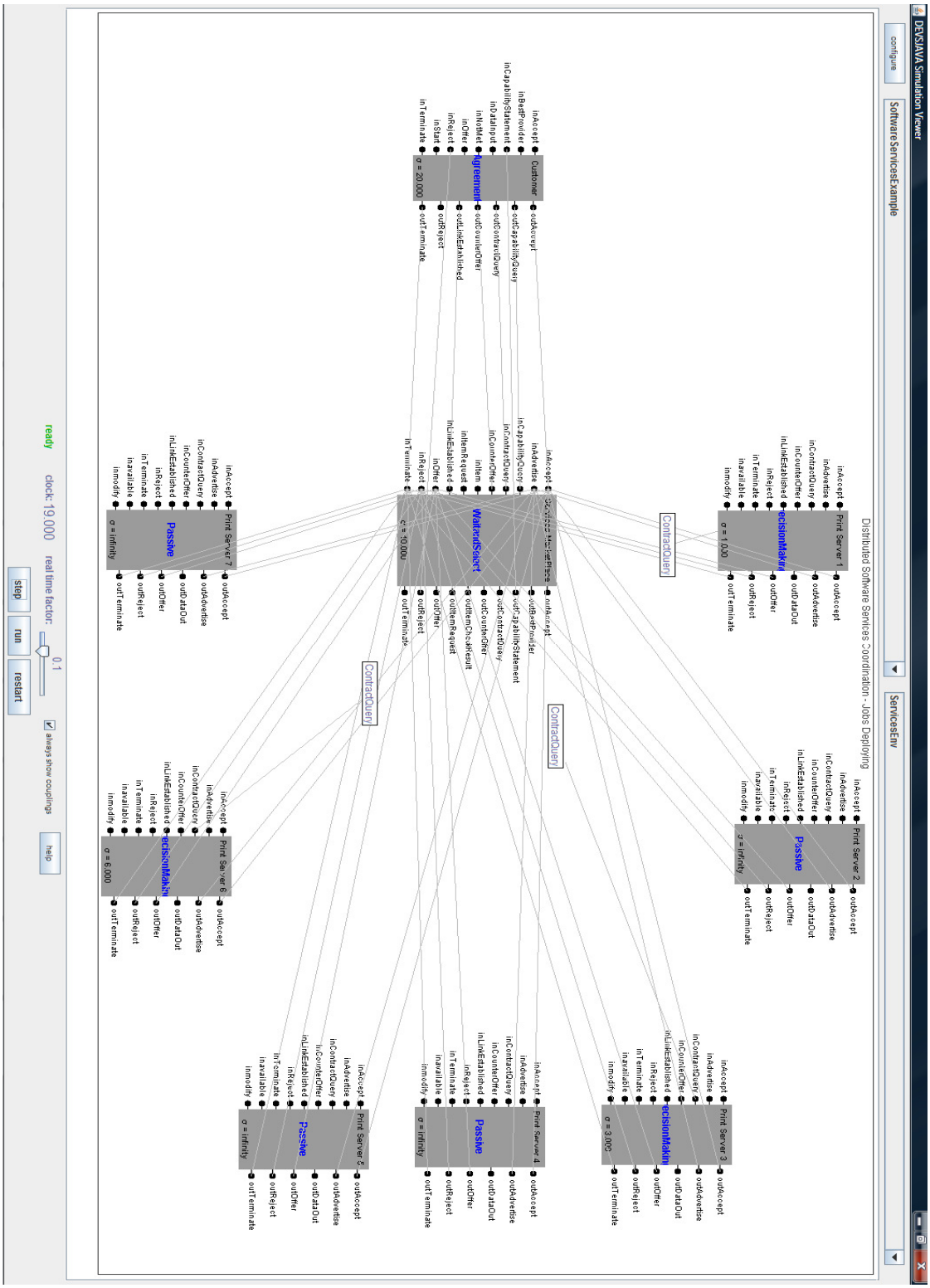


Figure 7.2.5.2: Dynamic coupling of ContractQuery exchange

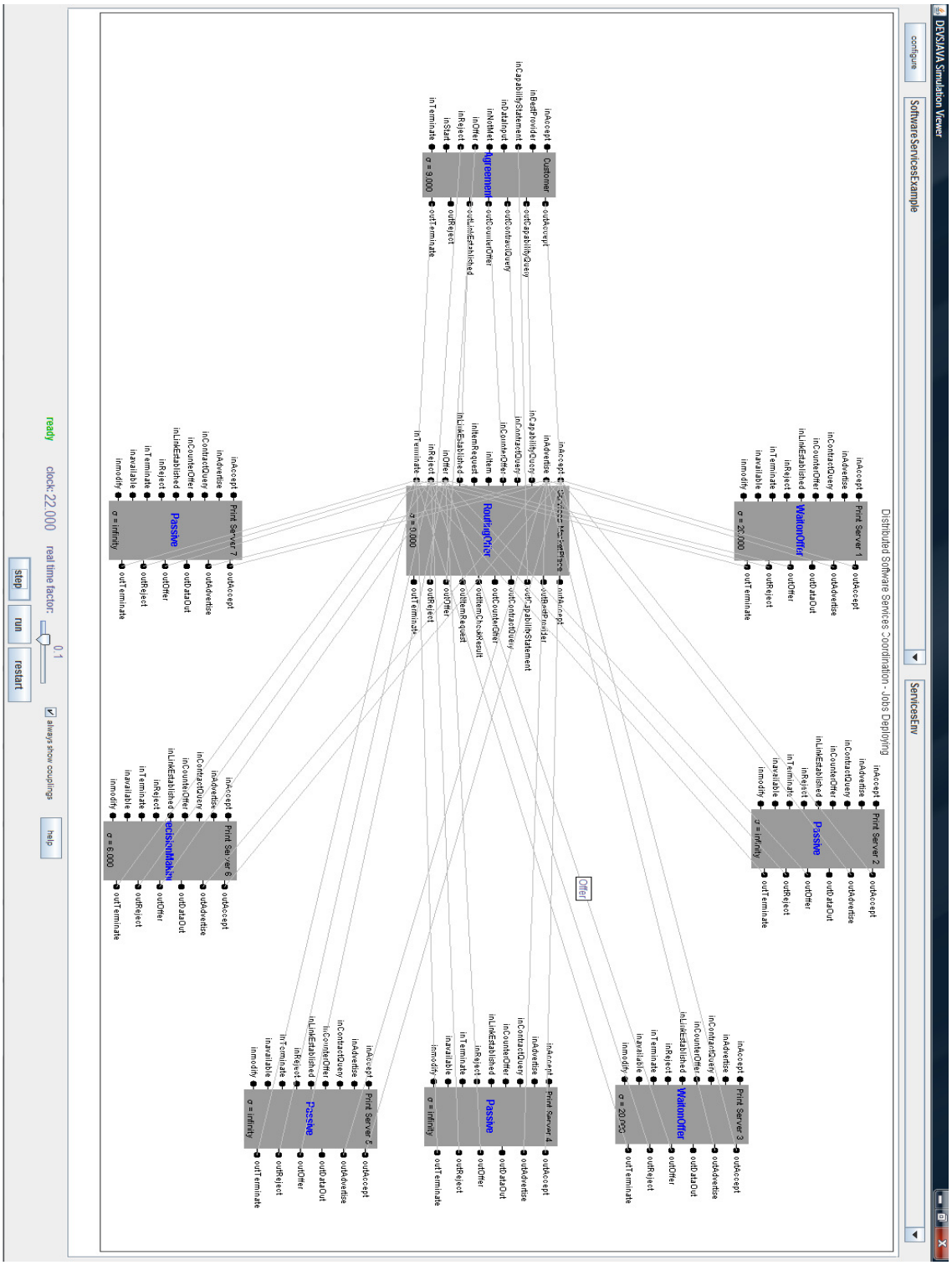


Figure 7.2.5.3: Negotiation through exchanging Offer messages

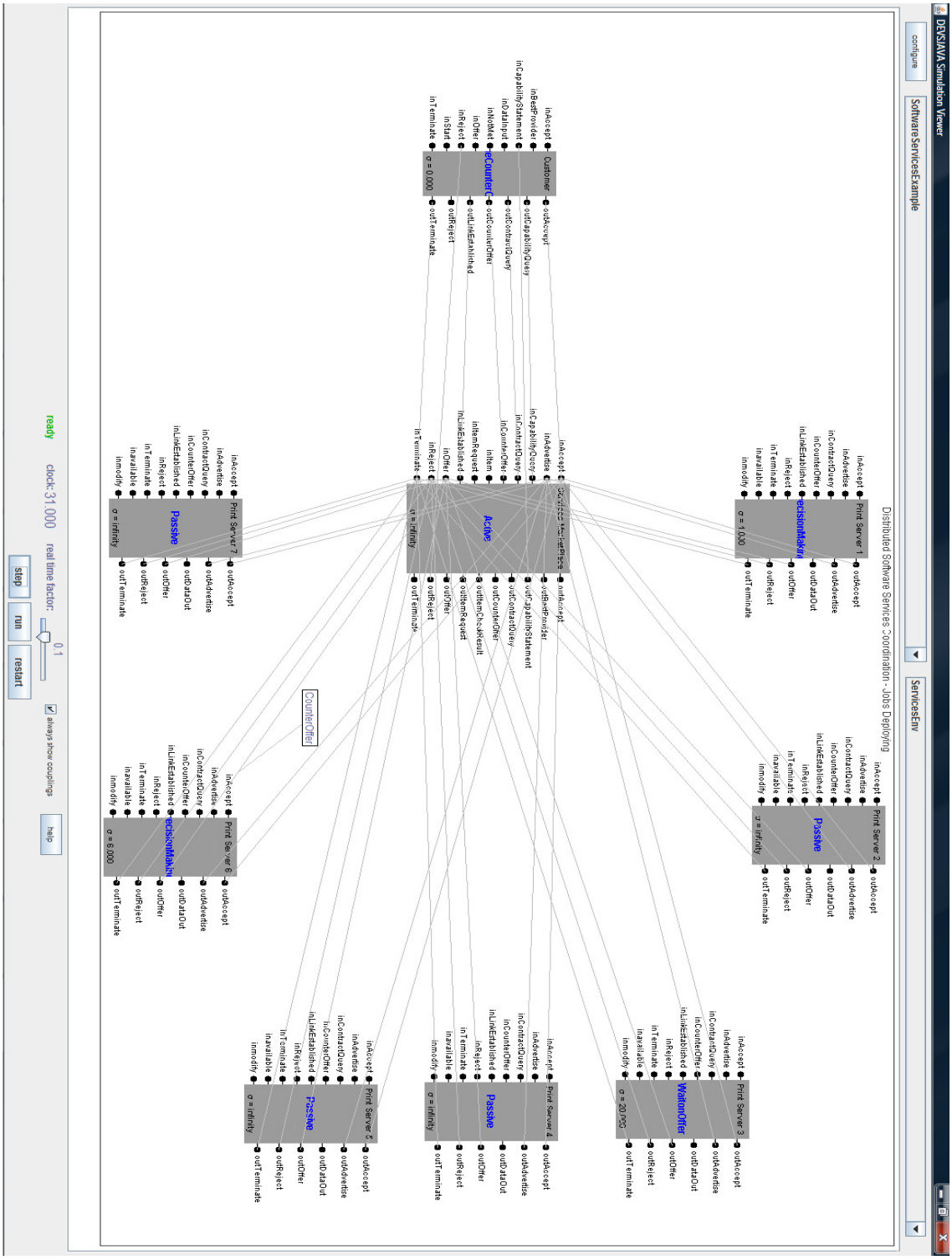


Figure 7.2.5.4: Negotiation through exchanging CounterOffer messages

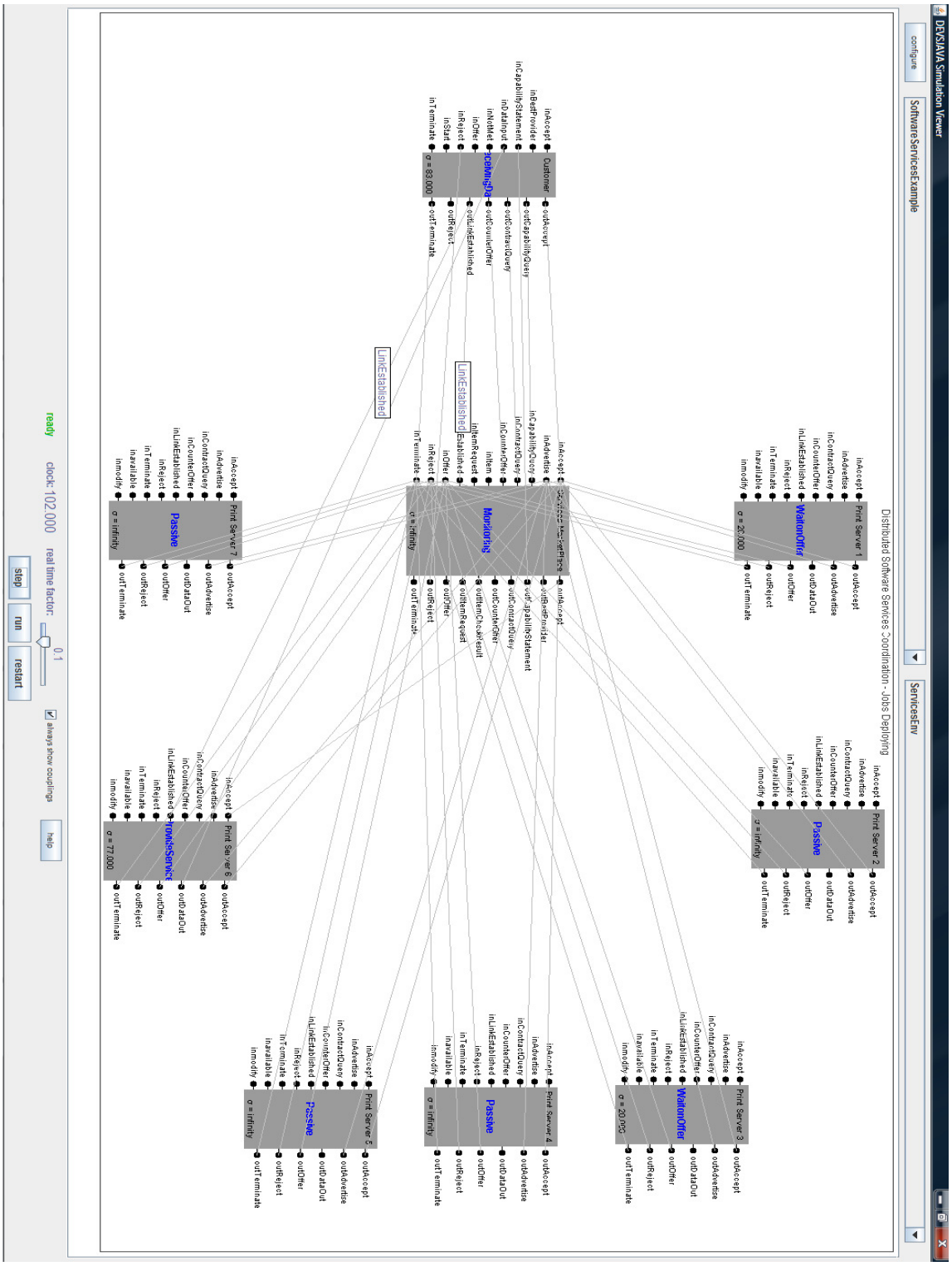


Figure 7.2.5.5: Link establishment messages

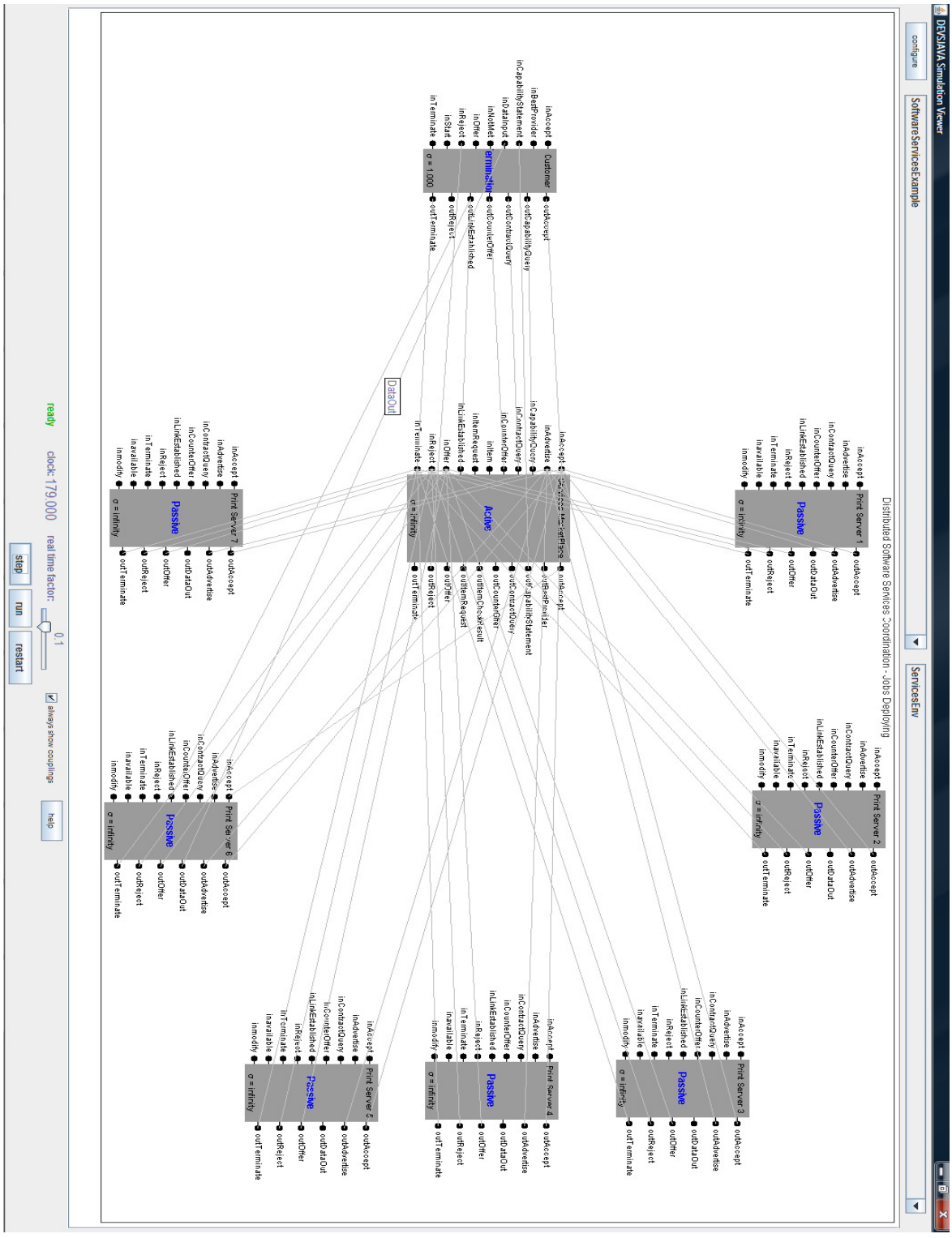


Figure 7.2.5.6: PrintingJob processing is finished

The output of the negotiation activity is an agreement as shown below. When we started the simulation, we did not know the result ahead of time. After the simulation is done, we compared the negotiation result with the Customer decision making options mentioned above in section 7.2.2. The terms of the agreement match the first condition of the User model decision making:

1. If the paper quality is medium or high, the color is full HD and the deadline is less than 80.

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>Agreement Offer information is:

Customer : Customer

Job Type : Business Cards

Print Server : Print Server 6

Color : FullHDColor

Paper Quality : High

Deadline : 78

Duplex : Yes

Number of Copies : 1

Technology Type : Thermography

>>>

Summary:

In this experiment we showed how the negotiation activity can be applied to the domain of distributed services environments. The marketplace agent plays a key role into discovering services providers and supervising the interaction between user agents and service providers. Having a trusted third party (Marketplace) gives the collaborative agents the confidence to deploy their jobs. A designer might want to use the same system to deploy programs or jobs into some computing resources to improve utilization. In such a situation, the designer might need to include Bandwidth, Execution Time, I/O tasks, etc. After that, the application of our system is straightforward and automated to generate the correct code that the designer will need. Hence, our approach is valid to be used under any of software and hardware multi-agent environments.

CHAPTER 8. PROOF OF CONCEPT (DEVS/SOA)

8.1 DEVS/SOA Environment

DEVS Service Oriented Architecture is a web services multi-server environment to support DEVS simulator. The system consists of two services, namely *MainService* and *Simulation* Service. Our concern in this section is the *MainService* and how can we deploy our models in the system. The *MainService* has four functionalities, *Upload* DEVS models, *Compile* DEVS models, *Simulate* DEVS models and *Get* results of the simulation. In order for our models to upload, compile and simulate correctly under the DEVS simulator, some minor modifications are needed to be done, which are:

- Atomic models need to inherit “atomic” class rather than “ViewableAtomic”, and the coupled model needs to inherit “digraph” class rather than “ViewableDigraph” class.

- DEVS Service Oriented Architecture was designed to support interoperability between different platforms and for heterogynous servers. In order to support that, the system nodes exchange messages among each other as strings in XML formats. For us to use such capability, we created a new class type of each of the language of encounter that has a String local variable where we send the pruned

XML structure of a specific message as a string. Figure 8.1.1 shows the *ContractQuery* primitive class.

The DEVS/SOA system we used is a centralized distributed simulation, which means, a coordinator controls the time for the next event t_N. The coordinator asks each node in the distributed environment for their local next time event t_N and collects them all. Then the coordinator calculates the minimum t_N, and informs each of the servers to change their next time event to the minimum t_N that was just computed. The following section shows the steps in deploying our models in DEVS/SOA and the output results of the distributed simulation. For more details on DEVS/Service Oriented Architecture system specifications and services, refer to [22][48][49]

```
/**
 *
 * @author Moath
 */
import GenCol.*;

public class XmlContractQuery extends entity{

    private String XmlContent;

        public XmlContractQuery(){
                super("ContractQuery");
        }


        public void setXmlContent(String str){
            XmlContent = str;
        }

        public String getXmlContent(){
            return XmlContent;
        }


        public String toString(){
                return getName();
        }

}
```

Figure 8.1.1: ContractQuery class implementation for DEVS/SOA

8.2 Printing Jobs Models Deployment in DEVS/SOA Environment

After preparing the Print Jobs experiment to run on DEVS/SOA environment, we chose five different machine servers to deploy the models. The first step of the models deployment is the IP assignments of each of the models Figure 8.1.2. The assignment does not need to be one-to-one as shown in table 8.1.1. The second step is to upload the models to the servers, where a copy of each of the models (client) will be sent to the

appropriate machine that has the IP address assigned to Figure 8.1.3. The third step is to compile the models and then the last step is to run the simulation.

IP	Model
150.135.218.200	Customer, Print Server 2, Print Server 4, Print Server 5 and Print Server 7
150.135.218.201	Print Server 1
150.135.218.203	Print Server 3
150.135.218.204	SOAMarketPlace and the Coupled model (ServicesSOAEnv)
150.135.218.206	Print Server 6

Table 8.1.1: Models assignment to the machines

We assigned Print Server 1, Print Server 3 and Print Server 6 to different machines dedicated to run their models because we knew from the beginning that those three print servers are the only ones capable to provide the customer request. Hence, in order to show that the negotiation occurs between separate machines, we chose this assignment.

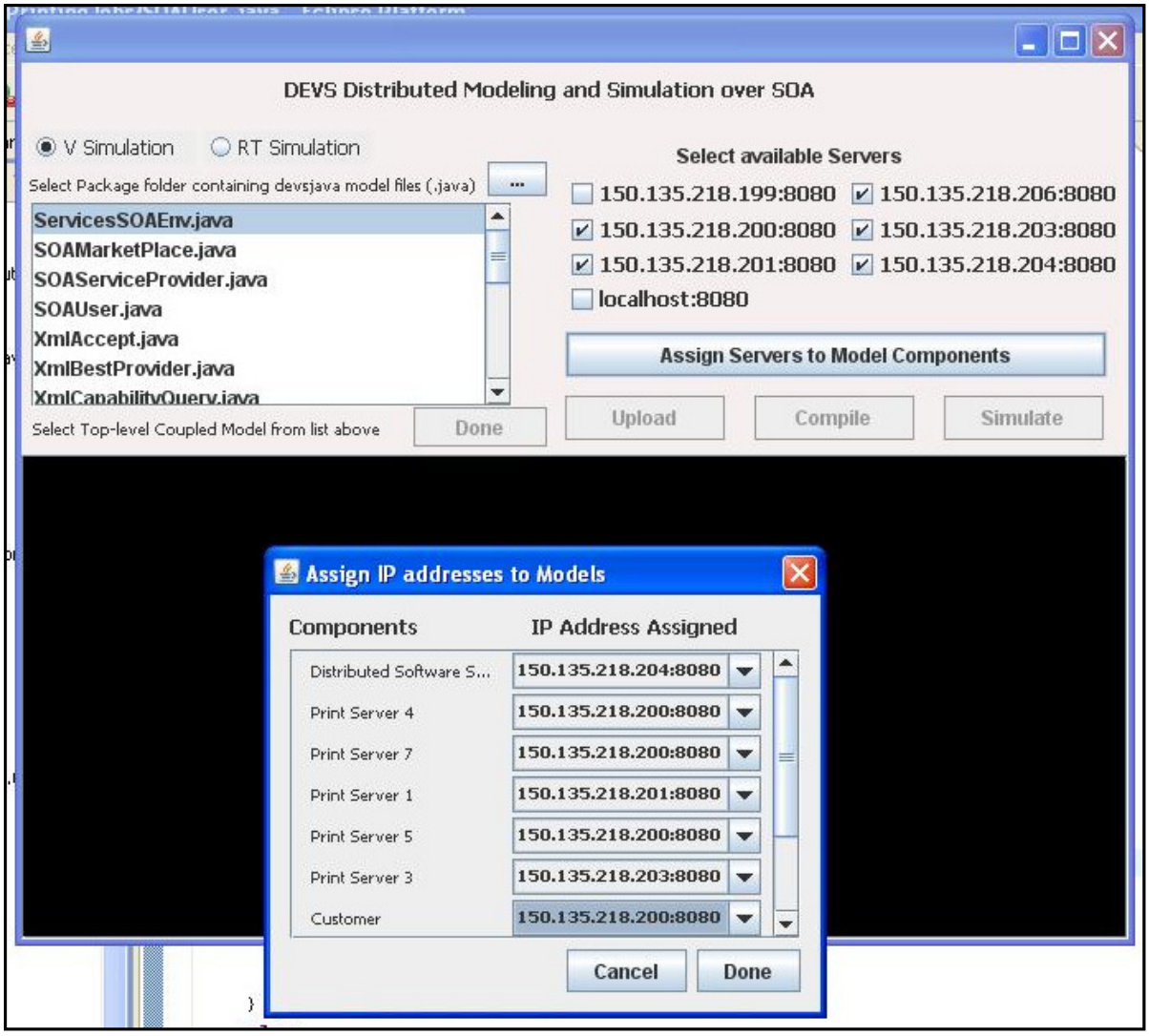


Figure 8.1.2: DEVS/SOA IP assignment

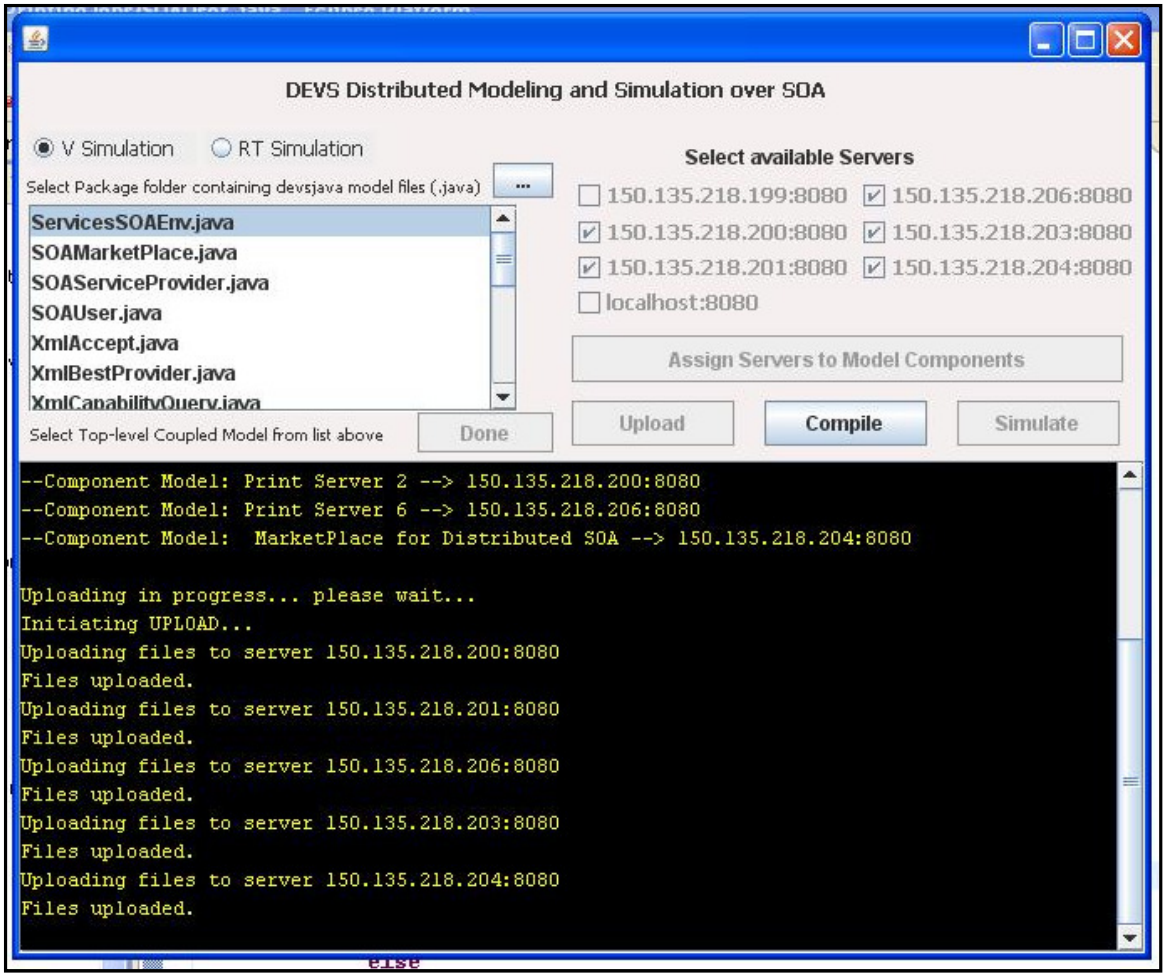
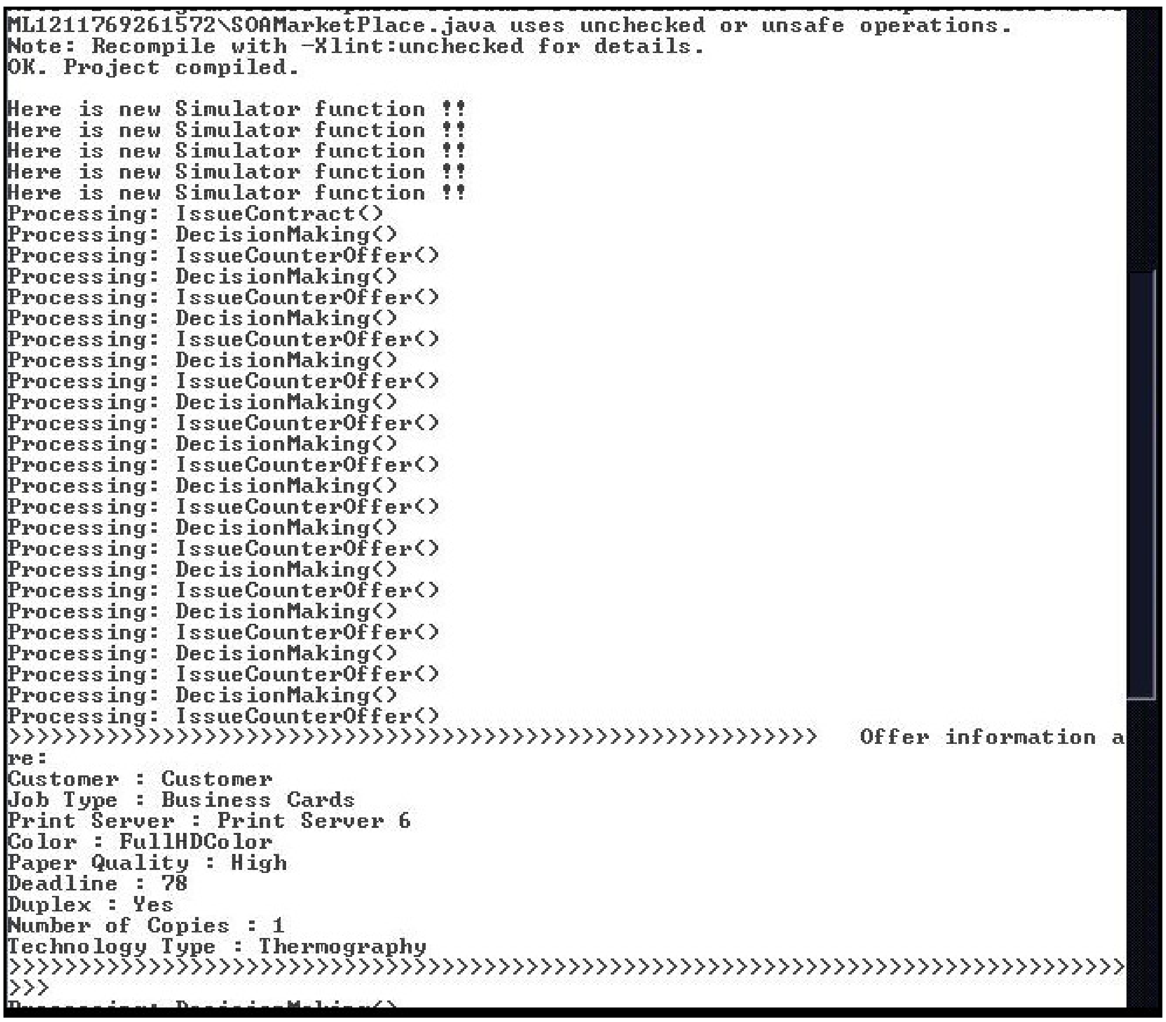


Figure 8.1.3: Models uploading process

After the simulation is over, we got the results as we expected. In the following figures, we will explain each of the server machines outputs. Figure 8.1.4 is the Customer model on machine 150.135.218.200. The output shows that the Customer sent a *ContractQuery* message to the SOAMarketplace asking for Business Cards Printing Job. Then he starts getting *Offers* which transits him to *DecisionMaking->IssueCounterOffer->DecisionMaking-> IssueCounterOffer* …and so on, until he receives an *Offer* that is acceptable to his decision making rules. After that, he establishes a link with the provider.

Also, we implement the Customer to print out any Offer he accepts. The output that says "Offer information …" is the terms and Offer specifications that the Customer agreed upon. Notice here, that although the rest of the other Print Servers (2, 4, 5, 7) are deployed and running on this machine, none of them produced output that is because they are not part of the negotiation since they do not provide Business Cards printing capability.



```
ML1211769261572\SOAMarketPlace.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
OK. Project compiled.

Here is new Simulator function !!
Here is new Simulator function !!
Here is new Simulator function !!
Here is new Simulator function !!
Here is new Simulator function !!
Processing: IssueContract()
Processing: DecisionMaking()
Processing: IssueCounterOffer()
Processing: DecisionMaking()
Processing: IssueCounterOffer()
Processing: DecisionMaking()
Processing: IssueCounterOffer()
Processing: DecisionMaking()
Processing: IssueCounterOffer()
Processing: DecisionMaking()
Processing: IssueCounterOffer()
Processing: DecisionMaking()
Processing: IssueCounterOffer()
Processing: DecisionMaking()
Processing: IssueCounterOffer()
Processing: DecisionMaking()
Processing: IssueCounterOffer()
Processing: DecisionMaking()
Processing: IssueCounterOffer()
Processing: DecisionMaking()
Processing: IssueCounterOffer()
Processing: DecisionMaking()
Processing: IssueCounterOffer()
Processing: DecisionMaking()
Processing: IssueCounterOffer()
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>   Offer information a
re:
Customer : Customer
Job Type : Business Cards
Print Server : Print Server 6
Color : FullHDColor
Paper Quality : High
Deadline : 78
Duplex : Yes
Number of Copies : 1
Technology Type : Thermography
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
>>>
```

Figure 8.1.4: The output of the customer machine

Print Server 1 and Print Server 3 outputs are almost the same except that each one of them outputs whatever *Offers* they are sending to the *Customer*. The offers information is for Business Cards printing. Notice here also that the Deadline does change from time to time since we designed them to update their Deadline such as:

CurrentDeadline = PreviousDeadline – Update

Print Server 6 is the winner provider of the negotiation process since he replied to the customer with an *Offer* that is acceptable to the Customer satisfaction. Hence, we can see in the output of the Print Server 6 that it goes into phase *ProvideService*. Print Server 1 and Print Server 3 outputs do not show that they provided any service to the Customer. Figure 8.1.5 shows a snapshot of the outputs of Print Server 1 and Print Server 3. Figure 8.1.6 shows the output of Print Server 6.

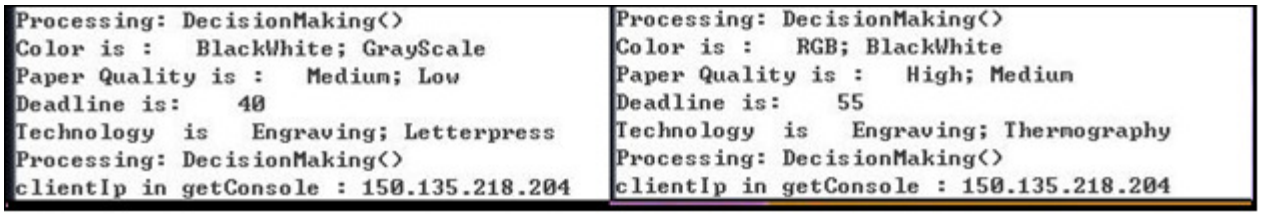


Figure 8.1.5: Print server 1 and print server 3 outputs side by side

Figure 8.1.6: Print server 6 output, showing providing service

Figure 8.1.7 shows a snapshot of the SOAMarketplace output on machine 150.135.218.204. the output shows that after the marketplace received a *CapabilityQuery* for Buisness Cards job, it accessed it's XML files database and found that Print Server 1, Print Server 3 and Print Server 6 are the only providers for Business Cards. Then it received a *ContractQuery* and transits to *InterpretQuery* to interpret the message. Then the market place went through *RoutingOffer*-> *RoutingCounterOffer*-> and so on, until it

received an *Accept* message, it forwarded it to the appropriate provider (Print Server 6) and then it transited into *Monitoring* after receiving *LinkEstablished* message. After Print Server 6 finished processing the Customer printing job, the Customer sends *Terminate* to the Marketplace causing its transition from *Monitoring* phase into *Active* phase.

```
minate.java"
OK. Project compiled.
Here is new Simulator function !!
The Printing Job is:   Business Cards
The print servers are:    Print Server 1;Print Server 3; Print Server 6
Processing: InterpretQuery()
Processing: RoutingOffer()
Processing: RoutingOffer()
Processing: RoutingOffer()
Processing: RoutingCounterOffer()
Processing: RoutingCounterOffer()
Processing: RoutingCounterOffer()
Processing: RoutingOffer()
Processing: RoutingOffer()
Processing: RoutingOffer()
Processing: RoutingCounterOffer()
Processing: RoutingCounterOffer()
Processing: RoutingCounterOffer()
Processing: RoutingOffer()
Processing: RoutingOffer()
Processing: RoutingOffer()
Processing: RoutingCounterOffer()
Processing: RoutingCounterOffer()
Processing: RoutingCounterOffer()
Processing: RoutingOffer()
Processing: RoutingOffer()
Processing: RoutingOffer()
Processing: RoutingCounterOffer()
Processing: RoutingCounterOffer()
Processing: RoutingCounterOffer()
Processing: RoutingOffer()
Processing: RoutingOffer()
Processing: RoutingOffer()
Processing: RoutingAccept()
Processing: Monitoring()
Processing: Termination()
clientIp in getConsole : 150.135.218.204
```

Figure 8.1.8: The output of the SOAMarketplace machine

This section ends our objective of the DEVS/SOA implementation which is a proof of the concept that our system can be used in different distributed engineering applications. Whether the distributed nodes are sensors who collect data and information, computing resources who provides an environment for software and hardware resources, print servers who provides different printing capabilities or online stores who provide products; all these and other domains can use the system to support different interaction behaviors. This can be done by using flexible negotiation protocols that are enforced by the trusted third party marketplace architecture we developed. The language of encounter, which was designed to be dynamic in structure, gives the domains enough expressive tools and capabilities to define their own messaging system so that users of the domain under consideration can simply understand and use them in the correct manner. Negotiation with service providers can take couple of minutes at the beginning to find the best (or an appropriate) provider; but once it is found, it could save hours and even days of data transformations or jobs processing.

CHAPTER 9. CONCLUSION AND FUTURE WORK

We believe that the negotiation process is an essential activity that needs to be used widely and correctly in today complex distributed systems. The complexity comes in having many parameters that manage computing resources in geographically distributed systems. Such systems need to provide negotiation capabilities on these parameters in order to reach agreements and behaviors that are efficient and intelligent. For example, a programmer that needs to deploy a task on a busy computing resource might keep on rechecking the resource availability every 1 minute. However, if we let the programmer negotiates with the computing resource; he might find out that the resource will be available until after 1 hour. As a result, he will wait and come back to deploy his task after 1 hour which is less costly and more efficient for both parties.

We have constructed an agent-based negotiation system that supports brokering between service providers and requestors. Two powerful and yet flexible negotiation protocols are used to enforce the rules of interactions. The rules are implemented in a trusted third party marketplace model which supervises the whole negotiation process while preserving privacy and transparency among the system users. Discrete event modeling and simulation environment (DEVS formalism) is used to implement the generic marketplace model. In order to accompany the negotiation protocols with flexible expressive primitives to handle negotiation behaviors in complex distributed systems, a dynamic structure of the language of encounter is implemented in SES

ontological framework. Each negotiation message has a separate ontology that defines its structure under different domain specialization entities.

The domain-independent marketplace design integrated with the domain-dependent language of encounter ontology gives system designers a very powerful tool to benefit from. With the automated code generation tool, given the language of encounter structures under a specific domain of interest and the domain name (both as inputs) produces a tailored negotiation marketplace model that is ready to be used. System designers usually need to add specific decision and behavioral criterions such as dynamic coupling to realize their wishes about the system they are interested in. This automated marketplace code generation results in a huge reduction amount in the software development time.

The negotiation system is evaluated by showing two different experiments for two different domains (applying it to other domains will have similar scenarios). The first one shows how brokering can lead to a data transformation contract from a data collector (such as a sensor) to a data requestor. Also we showed how the data collector can be changed dynamically through the use of the negotiation protocols. In the second experiment, we applied our system to the domain of distributed software services environment in which, services providers can do different job capabilities. In this context, we used print servers as our services providers. Since some print servers provide similar capabilities as others, and some provide services that none of the other can provide, negotiation over the capabilities is necessary. In order to have a proof of the concept in

distributed computing systems, we deployed our negotiation framework in Web Services environment (DEVS/SOA). Each one of the nodes has its own data (PESs) and running one of the print server capabilities. The system behaviors confined with our objectives and expectations. Our system provides the infrastructure that supports different domain with different negotiation requirements.

For the future work, we aim to add more functionality on the Ontology design GUI. The GUI right now is very basic and supports adding and deleting a domain along with its language of encounter structure. It will be useful for the designers to edit and modify on the structure of a domain. For example, if the designer makes a mistake in entering one or more of the slots names and he need to go back to fix it, currently he needs to redo the whole process. We aim to provide an "Edit" capability where the user can keep whatever he needs, delete whatever he does not need and modify errors in the names and the number of slots in the structure.

In some situations, providing manual interactions with *Offers-CounterOffer* message with the information that they are carried would be useful for systems users to understand what is happening. Another goal in our future work is to develop a decision making user interface under the DEVS/SOA environment where Web Services providers can manually modify and understand what the agreements terms are. The interface on the service provider side needs to support reading a received *CounterOffer* or a *ContractQuery*, unmarshal the data carried by the message and display it to the service provider. Then the provider can enter an offer information manually through the

interface, and then the interface will marshal it into an *Offer* message and sends it back to the requestor. In this case, the interface needs to be connected to the service providers XML data files (PESs) where it can read them, parse them and display them in a friendly way. Also it needs to be able to write and modify on these PESs. The interface on the requestor side needs to do the opposite. It need to marshals a *ContractQuery* from manually entered data and starts the negotiation, and then receive *Offer*s messages, unmarshal them and display them to the requestor to decide whether to accept the offer or start a *CounterOffer*. Either case the requestor enter manually data to be marshaled in the corresponding message, and then sends it to the service provider. The objective here is to support more features that some users can benefit from depending on their needs and convenient.

The framework provided here focused on the language of encounter as a support for the designer to implement more detailed negotiation protocols. Future work could extend the automation modeling to include such protocols and their properties. As stated earlier, the termination of the negotiation process is an important consideration. Future work might provide tools to support methods to guarantee termination of the negotiation so that it does not go forever. One approach can be implemented by including a timing counter in the marketplace structure which is decremented after each negotiation cycle of offers and counter offers. Once the timer hits zero, the agent can send a terminate message. Another approach that users might consider is to associate a timer each time the user start a negotiation process and compare the timer with the current DEVS simulation

clock. Once the clock reaches the timer value, the user terminates the corresponding negotiation process by sending a *Terminate* message.

Currently, researchers are concerned in developing techniques to process ontologies under different domains. One useful step into this research is to have a tool that can take an ontology under a specific domain and map it into language of encounter structure for that domain. In this regard, some messages structure are more sensitive than others. For example, *ContractQuery* and *Offer* are more sensitive than *Terminate* or *Accept* because they carry information on the agreement terms such as deadline, job type.

In order to commercialize this methodology, a designer can setup marketplace services for different domains. For example, for an airline tickets booking system, a designer can have a marketplace web service along with the language of encounter structure defined for that domain. Then, the designer can have another marketplace web service for printing photos for example, where the service users can upload their photos and print them and go pick them up. For this printing domain, the language of encounter will differ from the airline booking system.

Having a Web Services Description Language (WSDL) interface for the services that the marketplace provide us enable the development of this methodology as web services. For example, a service provider can have WSDL interface information about its name, location, IP address, port number, services that it provides. The marketplace as well needs to provide WSDL interface so that users know how to locate it and query it.

REFERENCES

[1] Bernard P. Zeigler, Herbert Praehofer and Tag Gon Kim, “Theory of Modeling and Simulation”, 2nd Ed, Academic Press, 2000.

[2] Bailin, S. and Truszkowski, W. “Ontology negotiation between scientific archives”, *Proceedings of the Thirteenth International Conference on Scientific and Statistical Database Management* (SSDBM 2001), IEEE Press, July 2001.

[3] M.H. Hwang and B.P. Zeigler, ``Reachability Graph of Finite & Deterministic DEVS``, IEEE Transactions on Automation Science and Engineering.

[4] Bernard P. Zeigler, "DEVS Today: Recent Advances in Discrete Event-Based Information Technology", 11th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp.148-161, 2003.

[5] Bernard P. Zeigler and Phillip E. Hammonds, “Modeling and Simulation-Based Data Engineering. Introducing Pragmatics into Ontologies for Net-Centric Information Exchange”, Academic Press, 2007.

[6] J. Kim and A. Segev, “A web services-enabled marketplace architecture for negotiation process management”. *Decision Support Systems*, Vol. 40, pp.71-87, July 2005.

[7] Y. Feng, Y. Lei, Y. Li and R. Cao, “Research on Collaborative Negotiation for E-Commerce"”. *Proceeding of the 2nd international conference on Machine Learning and Cybernetics*, Nov. 2003.

[8] Greg O’Hare and Nick Jennings, “Foundations of Distributed Artificial Intelligence”, *Sixth-Generation Computer Technology Series, John Wiley & Sons, Inc*. 1996.

[9] Inaba and T. Okamoto, “Negotiation Process Model to Support Collaborative Learning”. *Systems and Computers in Japan*, Vol.28, No. 14, 1997.

[10] Krishna, V.; Ramesh, V.C., “Intelligent agents for negotiations in market games. I. Model”, *IEEE Transactions on Power Systems*, Vol. 13, Issue 3, Aug 1998.

[11] Murugesan, S., “Negotiation by software agents in electronic marketplace”, *TENCON Proceedings*, Vol.2, 2000.

[12] Masvoula, M.; Kontolemakis, G.; Kanellis, P.; Martakos, D., “Design and development of an anthropocentric negotiation model”, *Seventh IEEE International Conference on E-Commerce Technology*, 2005.

[13] Park, H.C. and Kim, T.G., “Relational algebraic system entity structure for models management”, *IEE Preceedings on Computers and Digital Techniques*, Vol.143, Iss.1, pp.49-54, 1996.

[14] M. Contreras and J. Hernández, “Ontology Solution for Communicating Heterogeneous Negotiation Agents in a Web-based Environment”. *Proceedings of the Fourth Latin American Web Congress (LA-WEB’06) IEEE*, pp.59-66, 2006.

[15] D. Bell, S. A. Ludwig, M. Lycett, “Enterprise Application Reuse: Semantic Discovery of Business Grid Services”, *Journal of Information Technology and Management*, vol. 8, no. 3, pp. 223-239, 2007.

[16] Choi, B. Park, and J. Park, “A formal model conversion approach to developing a DEVS-based factory simulator,” *Simulation*, vol. 79, no. 8,pp. 440–461, Feb 2003.

[17] L. Ntaimo, B. Zeigler, M. Vasconcelos, and B. Khargharia, “Forest Fire Spread and Suppression in DEVS,” *Simulation*, vol. 80, no.10, pp. 479–500, Oct 2004.

[18] Concepcion and B. Zeigler, “DEVS Formalism: A Framework for Hierarchical Model Development,” *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 228–241, Feb 1988.

[19] J. Lee, Y. Lim, and S. Chi, “Hierarchical Modeling and Simulation Environment for Intelligent Transportation Systems,” *Simulation*, vol. 80, no. 2, pp. 61–76, Feb 2004.

[20] M. Hwang and S. Cho, “Timed Analysis of Schedule Preserved DEVS,” in *2004 Summer Computer Simulation Conference*, A. Bruzzone and E. Williams, Eds. San Jose, CA: SCS, pp. 173–178, 2004.

[21] Addis, M. J., Allen, P. J. and Surridge, M., “Negotiating for Software Services”. *Eleventh International Workshop on Database and Expert Systems Applications (DEXA2000)*, September 2000.

[22] Taekyu Kim, “Ontology/Data Engineering Based Distributed Simulation over Service Oriented Architecture for Network Behavior Analysis”, *Ph. D. Dissertation, Electrical and Computer Engineering Dept., University of Arizona*, Spring 2008.

[23] M.H. Hwang, ``Generating Finite-State Global Behavior of Reconfigurable Automation Systems: DEVS Approach``, *Proceedings of 2005 IEEE-CASE*, Edmonton, Canada, Aug. 1-2, 2005.

[24] V. Tamma, S. Phelps, I. Dickinson, and M. Wooldridge, “Ontologies for supporting negotiation in e-commerce”. *Engineering Applications of Artificial Intelligence*, 18:223–236, 2005.

[25] L. Yilmaz and S. Paspuleti, “Toward a Meta-Level Framework for Agent-Supported Interoperation of Defense Simulations”. *The Society for Modeling and Simulation International, JDMS*, vol.2, pp.161-175, July 2005.

[26] SESBuilder, “An Integrated tool to utilize System Entity Structure”. 2007, http://www.sesbuilder.com/

[27] W3C XML Schema for Finite Deterministic (FD) DEVS Models, 2007. http://saurabh-mittal.com/fddevs/

[28] Saehoon Cheon, Doohwan Kim, Bernard P Zeigler, “System Entity Structure For XML Meta Data Modeling; Application to the US Climate Normals”, *IEEE International Conference on Information Reuse and Integration*, Las Vegas, NV, July 2008.

[29] S. Decker, F. van Harmelen, J. Broekstra, M. Erdmann, D. Fensel, I. Horrocks, M. Klein, and S. Melnik. “The Semantic Web – on the respective roles of XML and RDF”. *IEEE Internet Computing*, September-October 2000.

[30] Krishna V. and Ramesh VC., “Intelligent Agents for Negotiations and Market Games, Part 1: Model”. *IEEE transaction on Power Systems*, Vol.13, pp.1103-1108, 1998.

[31] Archibald J. K., Hill J. C., Johnson F. R. and Stirling W. C., "Satisfying Negotiations". *IEEE Transaction on Systems, Man and Cybernetics, Part C*, Vol. 36, Issue 1, pp.4-18, Jan. 2006.

[32] Oracle Technology Network, Tutorial on JAXB “Unmarshaling and Marshaling Data: JAXB Insurance Profile System”, http://www.oracle.com/technology/sample_code/tutorials/index.html

[33] Ed Ort and Bhakti Mehta, Java Architecture for XML Binding (JAXB), March 2003. http://java.sun.com/developer/technicalArticles/WebServices/jaxb/

[34] Osborne M. J., and Rubinstein A., "Bargaining and Markets". Academic Press, San Diego, 1990.

[35] Mahajan R., Rodrig M., Wetherall D. and Zahorjan J., "Experiences Applying Game Theory to System Design", *proc. SIGCOMM PINS Workshop*, 2004.

[36] Persons S. and Wooldridge M., "Game Theory and Decisions Theory in Multi-Agent Systems". *Autonomous Agents and Multi-agent Systems* Vol.5, No.3, pp.243-254, 2002.

[37] Binmore K., and Vulkan N., "Applying Game Theory to Automated Negotiation", *Netnomics*, Vol.1, No.1, pp.1-10, 1999.

[38] Krishna V. and Ramesh VC., "Intelligent Agents for Negotiations and Market Games, Part 2: Application". *IEEE transaction on Power Systems*, Vol.13, pp.1109-1113, 1998.

[39] eBay. http://www.ebay.com

[40] Von Neumann J. and Morgenstern O., "The Theory of Games and Economic Behavior", Princeton Univ. Press, Princeton, NJ, 1944.

[41] Morris, J. and P. Maes. "Negotiating Beyond the Bid Price.", *Workshop Proceedings of the Conference on Human Factors in Computing Systems (CHI 2000),* April, 2000.

[42] "Modeling and Simulation-Based Data Engineering" Online Site. http://www.devsworld.org/

[43] Priceline. http://www.priceline.com/

[44] Mahajan R., Rodrig M., Wetherall D. and Zahorjan J., "Experiences Applying Game Theory to System Design", *proc. SIGCOMM PINS Workshop*, 2004.

[45] RTSync Tutorials. http://www.sesbuilder.com/ses_tutorial.html

[46] Amazon Auctions. http://www.amazon.com/auctions

[47] Susan E. Lander, "Issues in Multi agent Design Systems", *IEEE Expert: Intelligent Systems and Their Applications*, v.12 n.2, p.18-26, March 1997.

[48] Mittal, S., Risco-Martin, J.L., Zeigler, B.P., "DEVS-Based Simulation Web Services for Net-centric T&E", *Summer Computer Simulation Conference SCSC'07*, July 2007.

[49] Cheon, S., and B.P. Zeigler., "Web Service Oriented Architecture for DEVS Model Retrieval by System Entity Structure and Segment Decomposition." *Paper presented at the DEVS Integrative M&S Symposium*, Huntsville, AL 2006.

[50] L. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, and C. Wroe. "OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns", *IEEE 14th International Conference on Knowledge Engineering and Knowledge Management (EKAW)*, pp. 63-81, 2004.

[51] Arizona Center for Integrative Modeling and Simulation (ACIS). http://www.acims.arizona.edu/

[52] P.F. Patel-Schneider., "Building the Semantic Web Tower from RDF Straw.", *Proc. 19th Int'l Joint Conf. Artificial Intelligence (IJCAI),* pp.546-551-2005.

[53] P.F. Patel-Schneider., "what is OWL (and why should I care)?.", *Principles of Knowledge Representation and Reasoning*, 2004.

[54] H. Peter Alesso and Craig F. Smith, "Developing Semantic Web Services.", A K Peters, Ltd. 2005.

[55] S. Rodriguez, S. Le Mouëlic, J. P. Combe, C. Sotin, "Complementarity of Radar and Infrared Remote Sensing for the Study of Titan Surface", *Workshop on Radar Investigations of Planetary and Terrestrial Environments*, 2005.

[56] Fensel et al., "Enabling Semantic Web Services: The Web Service Modeling Ontology", Springer, 2007.

[57] Asuncion Gomez-Perez, Oscar Corcho, "Ontology Specification Languages for the Semantic Web," *IEEE Intelligent Systems*, vol. 17, no. 1, pp. 54-60, Jan/Feb, 2002.

[58] Shadbolt, N. Hall, W. Berners-Lee, T., "*The semantic Web revisited*.", *IEEE Intelligent Systems*, Vol. 21, Issue.3, 2006.

[59] Nancy Gordon, Cliff Ogleby, Remote Sensing Centre for Environmental Applied Hydrology, Department of Civil and Agriculture Engineering, University of Melbourne.

[60] Tim Berners-Lee, Senior Researcher at MIT's CSAIL, **http://www.w3.org/People/Berners-Lee/**

[61] Horrocks and P. F. Patel-Schneider., "A proposal for an owl rules language." *In Proc. of the Thirteenth International World Wide Web Conference (WWW 2004).* ACM, 2004.

[62] Henderson F. M. and Lewis A. J. *Manual of Remote Sensing*, *vol. 2*, 1999

[63] The European Southern Observatory (EOS) - http://www.eso.org/public/

[64] Hoh In, Olson, D. and Rodgers, T., "A Requirements Negotiation Model Based on Multi-Criteria Analysis Source", *Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, 2001.

[65] N. Lung, N. cheng, L. Lian-chen and WU Cheng, "An Auction-based Negotiation Procedure to resolve Price related Conflicts for Online Marketplaces", *Proceedings of the Third International Conference on Semantics, Knowledge and Grid*, pp. 86-91, 2007.

[66] H.M. Kim and A.Sengupta, "Extracting knowledge from XML document repository: a semantic Web-based approach Source", *Information Technology and Management*, Vol. 8, Iss. 3, September 2007, pp. 205 – 221, 2007.

[67] WordNet A lexical database for the English Language. http://wordnet.princeton.edu/

[68] John McCarthy, "Human Level AI Is Harder Than It Seemed", 1955.

[69] P. Zeigler, S. Mittal and X. Hu, "Towards a Formal Standard for Interoperability in M&S/System of Systems Integration", *GMU-AFCEA Symposium on Critical Issues in C4I*, May 2008.

[70] Bravo, M.C. Perez, J. Sosa, V.J. Montes, A. Reyes, G., "Ontology support for communicating agents in negotiation processes", *Proceedings of the Fifth International Conference on Hybrid Intelligent Systems*, Nov. 2005.

[71] Farquhar, A.; Fikes, R.; & Rice, J., "The Ontolingua Server: A Tool for Collaborative Ontology Construction." *Knowledge Systems Laboratory*, September, 1996.

[72] Dung, Tran Quoc; Kameyama, Wataru, “A Proposal of Ontology-based Health Care Information Extraction System: VnHIES”, *IEEE International Conference on Innovation and Vision for the Future*, March 2007.

[73] Tomai & M. Spanaki, "From ontology design to ontology implementation: A web tool for building geographic ontologies", *In Proceedings of the 8th 8th AGILE Conference on Geographic Information Science*, Estoril, Portugal, May 2005.

[74] Mathieu, P. and Verrons, M.H., “A generic model for contract negotiation.”, *In AISB'02 Symposium on Intelligent Agents in Virtual Markets,* April 2002.

[75] RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/rdf-schema/

[76] DISTAL – Distributed Software On-Demand For Large Scale Engineering Applications. EC project EP26386.

[77] Cooper, T.P., “Case studies of four industrial meta-applications”. *High Performance Computing and Networking*, Springer Lecture Notes in Computer Science, 1999.

[78] Zuo Z. and Zhou M., “Web Ontology Language OWL and its description logic foundation”, *Proceedings of the fourth International Conference on Parallel and Distributed Computing, Application and Technologies*, Aug. 2003.