Automated Certification of Online Auction Services

Thesis submitted in accordance with the requirements of
the University of Liverpool for the degree of Doctor in Philosophy
by
Wei Bai

October 2016

To my mum

Abstract

Auction mechanisms are viewed as an efficient approach for resource allocation and different types of auctions have been designed to allocate spectrum, determine positions for advertisements on web pages, and sell products on the Internet, among others. Online auctions can be implemented as an intermediary for both sellers and buyers in agent-mediated e-commerce systems. This raises two concerns. Firstly, the automation of online auction trading requires buyer agents to understand the auction protocol and have the ability to communicate with the seller agents (i.e., the auctioneer). Secondly, buyer agents need to automatically check desirable properties that are central to their decision making.

To address both concerns, we have proposed a certification framework to enable software agents automatically verify some desirable properties of a specific auction through a formally designed communication protocol, and then make decisions according to the result of the communication. Furthermore, we have extended the communication mechanism to the area of Semantic Web Service composition and have explored the verification of combinatorial auction mechanisms.

To demonstrate our approach, we have modelled online auctions as web services and have applied the technique of Semantic Web Service to represent auction protocols. Then we rely on computer-aided verification techniques to construct and check formal proofs of desirable properties for specific auctions. Finally, dialogue games are proposed to enable decision making and service compositions for software agents.

Contents

List of Figures

Acknowledgement

First, I would like to thank my supervisor Emmanuel Tadjouddine. I have benefited much from his research experience and wisdom. He has given me strong emotional support and taught me to do research "incrementally". I really appreciate his patience in our weekly meetings. Also, I would like to thank my secondary supervisors Terry Payne and Steven Guan for their invaluable support and guidance throughout my study.

I would also like to thank Yu Guo and Hui Zhang from USTC. Their suggestions on my research inspire me to explore various ideas. Many thanks to the staffs in CSSE of XJTLU: Kaiyu Wan, Gangmin Li, Ka Lok Man, Wenjin Lv, and Yong Yue. Also, I would like to thank my colleagues: Yuji Dong, David Afolabi, Dacheng Jiang, Meihua Li, Jie Ren, Ting Wang, Jieming Ma, Shi Cheng, Yungang Zhang, Chao Yan, Rongqiang Qian, Chun Guan, and Vijayakumar Nanjappan. Wish you every success in your further life. I also would like to thank the staffs in the library of XJTLU : Liping Yang, Jun Wang, and Qin Chen.

Furthermore, I would like to acknowledge the financial support from XJTLU and UoL.

Glossary

ADF Agent Dialogue Framework.

AMS Agent Management System.

CAP Combinatorial Auction Problem.

CDSA Composite Dialogue Service Automata.

DF Directory Facilitator.

DSA Dialogue Service Automata.

FIPA Foundation for Intelligent Physical Agents.

FPCC Foundational Proof-Carrying Code.

ITP Interactive Theorem Prover.

JADE Java Agent DEvelopment Framework.

OWL Web Ontology Language.

RDF Resource Description Framework.

RDF-S RDF Schema.

SOAP Simple Object Access Protocol.

SWRL Semantic Web Rule Language.

SWS Semantic Web Service.

UDDI Universal Description, Discovery and Integration.

VCG Vickrey-Clark-Groves Mechanism.

WSDL Web Service Description Language.

XML eXtensible Markup Language.

Chapter 1

Introduction

1.1 Motivation

In the last few decades, electronic commerce (e-commerce) has rapidly developed and widely spread which leads to faster and more efficient business process. E-commerce systems are viewed as tools for efficient allocation of resources and automated electronic transactions. Multi-agent systems consist of autonomous software components which have been introduced into e-commerce systems for the automation of electronic transactions. With the growth of online transactions, there has been an increased demand to develop advanced techniques to enhance agent-mediated e-commerce systems. In these systems, software agents (both seller and buyer agents) are designed and developed by different organizations to automatically accomplish their tasks on behalf of their owners. The automation of online transaction causes a variety of challenges. For example, one challenge is that agents automatically make their decisions without centralized control and they are constrained with limited computational resources. This implies that the trading mechanism of each e-commerce system should be represented in a machine-understandable way, and those agents should make decisions on the basis of the trading mechanism to maximize their payoffs. As a consequence, software agents should be self-interested which means that they attempt to gain the maximum profit for their owners. However, the behaviour of an agent may cause detrimental outcomes for its owner. To avoid losing profit, buyer agents should ensure that some desirable properties are held in the system. For instance, software agents may want to guarantee that they will not lose profit caused by fictitious bids in an auction. Another challenge is the lack of efficient communication protocol between agents. On one hand, redundant or unnecessary communication can increase instability and chaos into the system. On the other hand, well designed communication protocol can enhance coordination among agents. This thesis illustrates the research that has been conducted to address some of the challenges within agent-mediated e-commerce systems. We propose a certification framework that relies on techniques of communication and verification. In this thesis, *certification* is the procedure by which an agency assesses and verifies characteristics

of a system according to its requirements specification. *Verification* is the process of checking whether a specification satisfies certain properties. We are concerned with the formal verification that relies on formal methods which model systems as mathematical entities, so that software agents do not need to manually check the paper proofs. This research will benefit small and medium enterprises that develop software agents to carry out online trading. It also can inspire the research of designing and verifying new auction mechanisms in online settings.

1.2 Problem Statement

To enable buyer agents[1] to automatically verify desirable properties of a trading protocol and communicate with the service provider[2], we need to determine the following requirements.

1. *Representation of trading mechanisms.* In an agent mediated e-commerce system, the trading protocol should be represented in a machine understandable way, so that agents can read and interpret the meaning of the protocols. For example, the inputs of a system indicate the data should be provided to start a trading.

2. *Proof of desirable properties.* We need to use a language that is expressive enough to describe desirable properties of a trading mechanism and provide formal proofs of these properties, so that buyer agents can ensure these desirable properties of the trading mechanism.

3. *Automated verification of desirable properties.* Given the proofs of desirable properties, we need to enable buyers to verify the correctness of the proof. Thus the trustworthiness between buyer agents and seller agents can be promoted.

4. *Automated communication between agents.* The exchange of information between buyers and sellers requires the design of automated interaction protocol. The interaction between agents should not be limited to a client-server communication mode. The interaction protocol should support different types of communication. For instance, one participant can influence another to accept a proposition.

5. *Automated service composition.* As online transaction may require the combination of several services, automated service composition should be implemented to fulfill the requirements of agents.

[1] In this thesis, buyer agents are the same as the service consumers.
[2] In this thesis, service providers are the same as the seller agents.

1.3 Solution Outline

In this thesis, we focus on auctions which are important e-commerce protocols that have been used to buy and sell goods. Auctions are designed to have certain desirable properties, such as *incentive compatibility* which encourages agents to bid truthfully, or *efficient* [31] which maximizes the surplus. We focus on models of rationality by equilibrium concepts are well studied in game theory mechanism design. For example, dominant strategy equilibrium can be used to achieve incentive compatibility.

Online trading mechanisms can be represented as Web services. Semantic Web Services (SWS) [76] combine the technique of Web Services and Semantic Web [15]. Web Services enable automated discovery, selection, composition and execution of services while Semantic Web allows machine to interpret the meaning of ontologies. Therefore, we rely on SWS to describe the trading protocols to fulfill the first requirement of "Representation of trading mechanisms". By using ontology, SWS provide a shared knowledge base which defines the same interpretation of terms and operations for all participants. Thus SWS can be treated as a solution to represent trading mechanisms in heterogeneous environments where self-interested agents are designed and implemented.

Although SWS has the ability to represent the description of trading mechanisms, the expressiveness and reasoning abilities of SWS languages are limited. Inspired by the work of Tadjouddine et al. [97, 98] where model checking has been used to check the strategy-proofness property of an auction. We have applied logic-based languages to model and reason about systems to satisfy the second requirement of "Proof of desirable properties". Logic-based languages allow us to specify requirements of a system and formulate desirable properties for the system. Besides, we have adopted the Foundational Proof-Carrying Code (FPCC) [2] paradigm to enable software agents to automatically check the properties (such as safety) of the system to accomplish the third requirement of "Automated verification of desirable properties". As we have defined the specification of a trading mechanism using SWS, we need to translate the original specification to the logical based program. In the work of Caminati et al. [23, 24], they have explored the feasibility of applying formal proofs to verify properties of mechanisms. For example, they prove that allocations of a mechanism are pairwise disjoint, and prices are non-negative. Lapets et al. [60] have presented a typed language to define allocation algorithms for auctions. The property of truthfulness of the algorithm is automatically verified by analyzing the definition and keeping track of desirable characteristics. In our study, we have verified mechanisms that are written in ontology languages. This requires us to translate the ontology of a mechanism to a program that is expressed within a theorem prover.

In addition, we use dialogue games to support the communication between buyer agents and seller agents to satisfy the fourth requirement of "Automated communication between agents". In a dialogue game the seller provides Web services, and the buyer

agent allowed asking desirable questions to the seller. In our work, an inquiry dialogue game is designed as a bipartite communication protocol. In this protocol, buyer agents can query questions to the seller and then make judgments according to the result of a dialogue. Furthermore, we integrated the FPCC paradigm within the dialogue game so that buyers can check whether some desirable properties are held in a system. This design has been implemented in the agent development platform JADE [11]. Seller agents (i.e., auctioneers) and buyer agents are different components in the platform and they are integrated in a loosely coupled manner. To fulfill the fifth requirement of "Automated service composition", we propose dialogue service automata to integrate dialogue games to composite Semantic Web Services. The advantage of using automata based model is that the behaviors of services can be described as a sequence of states and the transitions of states can be associated with messages or activities [95].

1.4 Contributions

The main focus of this thesis is to enable software agents capable of automatically verify desirable properties of auction mechanisms. Furthermore, to make agents automatically carry out service inquiry and service composition in the setting of agent-mediated e-commerce systems. More specifically, this thesis addresses the research problems in Section 1.2 and provides the following contributions:

- The development of ontologies of Web services for online auctions. We have used OWL Web Ontology Language for Services (OWL-S) to describe single item English auction service in Section 4.1.2 and other related services such as payment and delivery in Section 6.2.2.

- We have translated the OWL-S specification into a program written using an imperative language within the interactive theorem prover Coq. The proofs of desirable properties such as dominant strategy have been developed based on Hoare Logic is covered in Chapter 4.

- We have implemented the FPCC paradigm to enable auctioneers (service providers) to publish the above proofs, then buyers (service consumers) to download these proofs combine with the specification. Therefore, buyers can automatically check the correctness of the proofs using the Coq proof checker [9].

- We have proposed an inquiry dialogue model [6] to enable buyer agents to automatically ask desirable questions to the auctioneer. These questions could relate to information such as QoS or properties which have formal proofs. Buyers can make decisions according to the result of the answer.

- We have developed information-seeking dialogue and persuasion dialogue to pass information during the process of a service and enable service composition in an online trading scenario which explained in Chapter 6.

- We have explored the verification of desirable properties for combinatorial auctions at the end of the thesis. As the expressiveness limitation of SWS languages, we have used Coq to formalize and prove desirable properties of a VCG auction which detailed in Chapter 7.

These contributions have led to a number of peer-reviewed publications:

1. Bai W, Tadjouddine E, Payne T. A Dialectical Approach to Enable Decision Making in Online Trading[M]//Multi-Agent Systems and Agreement Technologies. Springer International Publishing, 2015: 203-218.

2. Bai, W., Tadjouddine, E.M.: Automated program translation in certifying online auctions. In: ETAPS/VPT 2015, 11-18 April, London, UK (2015).

3. Bai W, Tadjouddine E, Payne T. Dialogue Driven Semantic Web Services. Yue, Y., Ariwa, E., IEEE International Conference on Computing and Technology Innovation (CTI 2015), 27-28 May, Luton, United Kingdom, 2015.

4. Bai, W., E. M. Tadjouddine, and Y. Guo (2014). Enabling Automatic Certification of Online Auctions. In: Proceedings 11th International Workshop on Formal Engineering Approaches to Software Components and Architectures.(EPTCS 147, Apr. 2, 2014). Ed. by J. K. B. Buhnova L. Happe, pp. 123-132. doi: 10.4204/EPTCS.147.9.

5. Bai W, Tadjouddine E M, Payne T R, et al. A proof-carrying code approach to certificate auction mechanisms[M]//Formal Aspects of Component Software. Springer International Publishing, 2013: 23-40.

1.5 Thesis Outline

Chapter 2 provides the literature that is related to agent-mediated e-commerce systems, including Multi-agent systems, Semantic Web, program verification, and agent communication techniques.

Chapter 3 studies a pilot example that verifies some desirable properties of a single item Vickrey auction, and then proposes a certification framework based on Semantic Web Service, FPCC and dialogue games.

Chapter 4 presents a specification of an online auction in a Semantic Web Service

language OWL-S, and then describes the translation of this specification from OWL-S to an imperative program that is defined within Coq. Some desirable properties of this auction have been stated and proved at the end of this chapter.

Chapter 5 proposes a dialogue game that integrates with FPCC paradigm to enable information inquiry and decision making in online auction services.

Chapter 6 introduces a dialogue driven approach to composite Semantic Web Services.

Chapter 7 provides the formalization and verification of desirable properties of combinatorial VCG auctions.

Chapter 8 summarizes the thesis and discusses the future research directions of this work.

Chapter 2

Background

This chapter provides an overview of the techniques and methodology that are relevant to the research topic of this thesis.

The organization of this chapter is as follows: Section 2.1 introduces the basic model of multi-agent systems, the knowledge of games and auctions which can be treated as a type of games. Section 2.2 shows the concept of Semantic Web and its application. Section 2.3 presents the introduction to the technique of program verification. Section 2.4 describes background knowledge of agent communication.

2.1 Distributed Computing/Artificial Intelligence

The network has linked billions of computers worldwide, which enables computers to interact and collaborate with each other and to share resources among the system. This leads to the development of *distributed system* that integrates multiple autonomous components to communicate with each other by message passing. Besides, a branch of artificial intelligence (AI) is centered on the concept of a rational agent. An agent is a computer program that is situated in some environment and is capable of autonomous actions to meet its design objectives [114]. An agent is called *rational* when it is consistently tries to optimize its performance. As the development of AI, more and more tasks are delegated to computers, such as unmanned aerial vehicles, cleaning robots, and self-driving cars. These trends have led to the development of *multi-agent systems*.

2.1.1 Multi-agent Systems

An agent is a computer system that can automatically carry out actions on behalf of its owner to achieve design objectives. A single intelligent agent has four types of properties [115]:

- *Autonomy*: this indicates how independently agents operate and control over their initial state and actions.

- *Reactivity*: agents should be able to perceive their environment, and respond to changes in a short time.
- *Pro-activeness*: agents should be able to generate and attempt to achieve goals, rather than simply act in response of external events.
- *Social ability*: agents have the ability to interact with each other to meet the social characteristics of cooperation, coordination, and negotiation.

In a multi-agent system, multiple intelligent agents cooperate, coordinate, and negotiate with each other to achieve their objectives in a common environment. Cooperation means agents work with others when a goal cannot be achieved by a single agent or it will gain better results. Coordination represents the problem of managing inter-dependencies between activities of agents. Negotiation is used when agents need to reach agreements on matters of mutual interest.

The realization of multi-agent systems requires the development of programming languages and tools. JADE [12] is one platform that has been designed to develop multi-agent applications in compliance with the FIPA specifications [38]. Our framework is implemented using JADE. JADE is a distributed middleware system that is written in Java, and it allows easily extended by users. JADE includes three core components [12]:

- a **runtime environment** where agents can be activated and executed;
- a **library** that can be used by programmer to develop agents;
- **graphical tools** that can be used to administrate and monitor agents' activities.

One running instance of the JADE runtime environment is called a **Container**, and each container can contain several agents. Agents are distinct from one another with unique names. A collection of active containers is called a **Platform**. A very first container is a single **Main Container** which starts a platform. Other containers in the same platform follow the main container and they must register to the main container as soon as they start. The difference between a main container and other normal containers is the main container includes two special components which are not in the normal containers. They are **AMS** (Agent Management System) which has the authority to perform management actions and **DF** (Directory Facilitator) which allows agents to publish and find services.

JADE agents carry out actual jobs by implementing **Behaviour** objects which are used to specify the actions of agents. JADE provides two main classes to represent primitive and composite behaviours respectively. The components of these two behaviours are as follows:

- Primitive Behaviours

- **OneShotBehaviour** models atomic behaviours only be executed once.
- **CyclicBehaviour** models atomic behaviours that are executed as long as an agent is alive.
- **TickerBehaviour** models atomic behaviours that are executed periodically.
- **WakerBehaviour** models atomic behaviours only execute after a particular time elapse.

- Composite Behaviours
  - **SequentialBehaviour** models behaviours that execute its children behaviours one by one until the final behaviour has ended.
  - **ParallelBehaviour** models behaviours that execute its children behaviours concurrently until reach the termination condition.
  - **FSMBehaviour** models behaviours that execute its children behaviours on the basis of a Finite State Machine.

2.1.2 Game Theory and Auctions

Game theory is the formal study of decision-making among intelligent rational agents applied in economics, political science and computer science, etc. In game theory, there are two premises: one is that all participating agents are rational; the other is that agents take into account other agents' decisions in their decision making. In this thesis, we concentrate on *noncooperative* game theory where the basic modeling unit is the individual [90]. Games can be represented in normal form, or strategic form, where all players choose their strategies simultaneously. The normal form uses a triple to define a game: a set of players $N = \{1, ..., n\}$, a set of strategies S_i that are available to player i, where $i \in N$, and a utility function $u_i(S_i, S_{-i})$, which describes the utility of player i if it uses strategy S_i and other players use strategies S_{-i}. Extensive form games model the sequential decision making of agents, and they can be represented by game trees.

A noncooperative game can be illustrated by the prisoner's dilemma example. Assume that there are two prisoners R and C. They are interviewed separately so that they cannot communicate with each other. Each of them has two strategies, called "Confess" and "Deny". They aim to minimize the year of imprisonment. Table 2.1 shows the payoff of this game. If both of them choose "Deny", they will both serve 2 years in prison. If one of them chooses "Confess", the confessor will serve 1 year and the other will serve 5 years. If both of them choose "Confess", both of them will serve 3 years. Under the setting of noncooperation, each prisoner is interested in their own payoff. If prisoner R chooses "Confess", the better response of prisoner C is "Confess".

R \ C	Confess	Deny
Confess	3, 3	1, 5
Deny	5, 1	2, 2

Table 2.1: Prisoner's dilemma

If prisoner R changed to choose "Deny", the better choice for prisoner C is still "Confess". The case of prisoner R is symmetric. According to this reasoning, both of them will choose "Confess". In the game of prisoner's dilemma, "Confess" is a dominant strategy for each player to yield the best payoff.

An auction can be modeled as a noncooperative game where bidders can have conflict of interests. Auctions have been exploited as a testing-ground for game theory with incomplete information. Besides, a huge volume of goods has been sold through auctions, and new auction mechanisms have been designed to sell different kinds of products, such as the "Generalized Second Price Auction" which is used by Google to sell Google Ads advertisement slots.

Auctions are composed of rules that describe the mechanisms of winner determinations and payments. Four basic types of auctions for single item are widely used.

- *English auctions*, also called *Ascending-bid auctions*. In this auction format, a single item is offered for sale interactively in real time. One auction starts from a lowest bidding amount, and the seller gradually raises the price until only one bidder remains. The last bidder wins and pays the last bidding amount. The seller can set a reserve price for the item. If all the bids are less than the reserve price, then this item is not sold. In other forms of English auctions, bidders can shout out bids or submit them through the network.

- *Dutch auctions*, also called *Descending-bid auctions*. These auctions are also interactive auctions in which the seller gradually decreases the price of the item from a highest bidding amount. A Dutch auction ends when the first moment a bidder accepts the offered bidding price. The payment of the winner is the accepted price. These types of auctions are exhaustively used by flower merchant in the Netherlands.

- *First-price sealed-bid auctions*. These are auctions in which bidders simultaneously submit their sealed bids to the seller, so that a bidder only knows his own information. The bidder with the highest bid wins and the payment is the highest bid.

- *Vickrey auctions*, also called *Second-price sealed-bid auctions*. In this kind of auction, all bidders simultaneously submit their sealed bids to the seller. The

winner is the bidder who has quoted the highest amount, but the winner only pays the second highest quoted amount.

In an English auction, the dominant strategy for a bidder is rise from the reserve price, and then dropout when the actual bidding amount is equal to its valuation. For a Vickrey auction, the dominant strategy for each bidder is bid its valuation, which is also called *incentive compatible*. An extended Vickrey auction that also satisfied the property of incentive compatibility is called *Vickrey-Clarke-Groves* or *VCG*. A VCG auction is a combinatorial auction in which bidders can place bids on combinations of items. It is performed by finding the allocation that maximizes the social welfare and the payment rule gives discount to each winner on the basis of his contribution to the overall value for the auction.

2.2 Semantic Web

Normal Web is designed for application to human interactions. Semantic Web [15] extends the normal Web by providing machine-readable data for computers. Semantic Web techniques enable people to create and publish data on the Web in a machine interpretable format so that computers can automatically query and reason on these data. To achieve the goal of Semantic Web, meta-data are used to express the meaning of data and can be exchanged among computers. The World Wide Web Consortium (W3C) promotes the so-called "Semantic Layer Cake" framework for the development of meta-data in Semantic Web.

2.2.1 Semantic Web Layer Cake

Semantic Web techniques are composed of markup languages, which are designed to annotate documents or web pages, with a formal syntax and semantics. These languages are used to describe data in a standard concept and form the Semantic Web hierarchy [93]. Figure 2.1 shows the hierarchy of the Semantic Web.

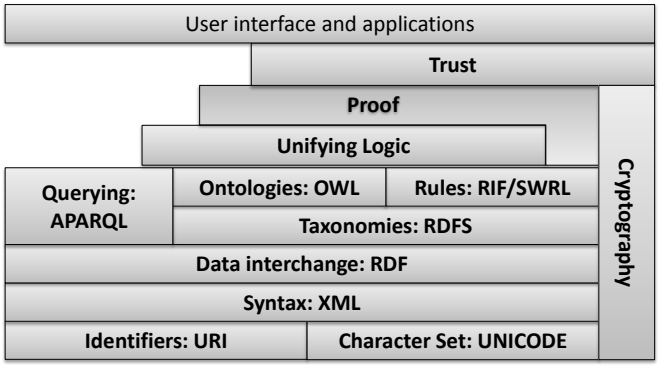


Figure 2.1: Semantic Web Stack

The bottom layer contains Unicode and Uniform Resource Identifiers (URI). Unicode is a character encoding system that provides a unique number for every character. URIs are short strings that identify resources in the web that provide unique and unambiguous names for resources. The second layer is XML (eXtensible Markup Language) which can be used to define the structure of data by adding tags. XML provides a way to define the syntax of data, while the semantics of data cannot be expressed. RDF (Resource Description Framework) provides an approach to describe resources via meta-data. All the statements in RDF are expressed as triples (subject, predicate, and object). RDF-S (RDF Schema) is an extension of RDF which allows describing the taxonomies of classes and properties.

The OWL Web Ontology Language is a standard W3C Recommendation ontology language. The term *ontology* represents the kinds of entities and their relations in the world. OWL provides more vocabulary to describe classes and properties than RDF-S. OWL is a Description Logic based ontology language and the details of Description Logic can be found in [56]. OWL provides set operations (e.g. union, intersection, complement), universal and existential quantifications, cardinality constraints, etc. Given an OWL ontology, its logical consequences can be entailed on the basis of the OWL formal semantics. W3C provides standardize OWL [75] in 2004 and its successor OWL 2 [78] in 2009. OWL provides three sub-languages based on their expressiveness and computational complexity:

- **OWL DL** is a Description Logic based language with RDF syntax. It is computational complete which means that all conclusions are guaranteed to be computable and decidable which means that all computations will finish in finite time.
- **OWL Lite** is a subset of OWL DL, a good choice for users to describe classification hierarchy and simple constraint features.
- **OWL Full** combines OWL DL and RDF. Compared to OWL DL, OWL Full is undecidable.

OWL 2 extends OWL by adding more functionality, such as property chains, qualified cardinality restrictions, and enhanced annotation capabilities, etc. OWL 2 has different variants.

- **OWL 2 Full** is an extension of RDFS, a very expressive language and it is fully upward-compatible with RDF. However, OWL 2 Full is not decidable.
- **OWL 2 Lite** is a syntactically restriction version of OWL 2 Full and it offers reasonably efficient reasoning support.
- **OWL 2 QL, OWL 2 EL, OWL 2 RL** are three subsets of OWL 2. They are designed to guarantee scalable reasoning for different application scenarios.

SWRL (Semantic Web Rule Language) [50] is a rule language that extends OWL to provide more powerful deductive reasoning capabilities. SWRL is represented as an implication between an antecedent and consequent. One of the powerful functionalities of SWRL is that it provides the built-ins for comparisons and arithmetic operations. Both SWRL and OWL obey the assumption of Open World Assumption where absence of information is interpreted as unknown information. RIF (Rule Interchange Format) [53] is another W3C Recommendation rule language. SPARQL [45] is a query language for RDF and it provides a standard format and a set of rules for writing and processing queries. The above three layers which concern complex logics and proofs to establish trustworthiness is still an ongoing research topic in the area of Semantic Web.

2.2.2 Semantic Web Services

Web services provide support for inter-operable machine-to-machine interaction over networks. There are three major roles in Web services: the first one is the service provider that provides the implementation of a particular Web service, the second is the service requester that wishes to make use of a provider's Web service, and the third is the service registry that registers Web services. Web Services are built by using different standard technologies. The exchanging messages are encoded in XML format so that information can be interpreted by computers. SOAP (Simple Object Access Protocol) is a standard messaging protocol for packaging and exchanging XML messages and SOAP messages can be carried by variety of network protocols. The Web Service Description Language (WSDL) is a language that used to define the public interface of Web services. WSDL describes the operations of a specific Web service, the data format to access the Web service, and the location of the Web service. Another protocol UDDI (Universal Description, Discovery and Integration) is used to publish and locate services in a repository.

Web Services provide standards to ensure interoperability across diverse platforms. However, it is not sufficient for software agents to automatically use Web services. To integrate Web Services with multi-agent systems, agents need to contain and reason about the semantics of the Web Services. The technique of Semantic Web inspires the development of Semantic Web Services (SWS). SWS address the challenge of automatically discovery, composition and execution of Web services that involved in intelligent software agents. OWL-S (Web Ontology Language for Services) is an ontology language that is proposed to represent an upper ontology for the description of Semantic Web Services expressed in OWL. OWL-S contains three interrelated parts: **ServiceProfile** describes what the service does; **ServiceModel** describes how the service is used; and **ServiceGrounding** describes how to interact with the service.

2.3 Program Verification

Program verification is a research area that studies formal methods for checking a software program conforms to its specification. It concerns properties of codes, which can be studied in many different methods such as *logic*, *model checking* and *abstract interpretation*. In our work, we use the logic-based methods to verify the properties of a specification.

2.3.1 Program Specification

A specification describes the desired behaviors of a program. There are three branches in formal methods to describe a specification of a system. The first one is the model-based specification which describes the internal states and the transition of states of a system. The second is the algebraic approach that specifies a system in terms of its operations and their relationships. The last is the declarative specifications which describe systems using logic-based languages, functional languages and rewriting languages, etc. In our work, we use the declarative approach to describe the specification of programs.

2.3.2 Hoare Logic

Hoare Logic [48] is introduced by C.A.R. Hoare in 1969 to reason about the correctness of imperative programs. In our work, we used Hoare Logic to verify the desirable properties of *While* programs which are written in a simple imperative language. The specification of a *While* program consists of a *precondition* and a *postcondition*. The inference system of Hoare Logic is used to construct a derivation to determine the correctness of the specification. The imperative language includes a skip command, assignment, sequential composition, if-then-else branches and a while loop.

In this work, we consider a WHILE-language composed of assignments, `if`, and `while` statements, and in which expressions are formed using basic arithmetic or logical operations. We denote $\mathbb{V}$, a set of program variables that are integer or Boolean, $\mathbb{E}$ the set of arithmetic expressions, $\mathbb{B}$ the set of Boolean expressions, and $\mathbb{C}$ the set of commands or statements. This language can be described as:

$$
\begin{array}{lcl}
x & \in & \mathbb{V} \\
aop & \in & \{+, -, \times, /\} \\
rop & \in & \{<, >, ==, \leq, \ldots\} \\
lop & \in & \{\wedge, \vee, \neg, \ldots\} \\
\mathbb{E} \ni e & ::= & const \mid x \mid e \text{ aop } e \\
\mathbb{B} \ni b & ::= & true \mid false \mid e \text{ rop } e \mid b \text{ lop } b \\
\mathbb{C} \ni c & ::= & \text{skip} \mid x := e \mid c; c \mid \text{if } b \text{ then } c \text{ else } c \mid \text{while } b \text{ do } c
\end{array}
$$

The states $\sigma \in \mathbb{S} = \mathbb{V} \to \mathbb{Z}$ are defined as associations of values to variables, and the evaluation of expressions remains standard in the natural (or big-step) semantics,

see for example [61]. We denote $\sigma \rightarrowtail C \rightarrowtail \sigma'$ to mean a command c evaluated in a pre-state σ leads to a post-state σ'. This allows us to reason on the program by using Hoare Logic.

$$\frac{}{(|\psi[E/x]|)x = E(|\psi|)} \textsc{Assignment}$$

$$\frac{(|\phi|)C_1(|\eta|) \qquad (|\eta|)C_2(|\psi|)}{(|\phi|)C_1; C_2(|\psi|)} \textsc{Composition}$$

$$\frac{(|\phi \wedge B|)C_1(|\psi|) \qquad (|\phi \wedge \neg B|)C_2(|\psi|)}{(|\phi|)if\ B\ \{C_1\}\ else\ \{C_2\}(|\psi|)} \textsc{If-statement}$$

$$\frac{(|\psi \wedge B|)C(|\psi|)}{(|\psi|)\ while\ B\ \{C\}(|\psi \wedge \neg B|)} \textsc{Partial-while}$$

$$\frac{\vdash \phi' \rightarrow \phi \qquad (|\phi|)C(|\psi|) \qquad \vdash \psi \rightarrow \psi'}{(|\phi'|)C(|\psi'|)} \textsc{Implied}$$

Figure 2.2: Proof rules for partial correctness of Hoare triples.

Hoare logic is a sound and complete formal system providing logical rules for reasoning about the correctness of computer programs. For a given statement S, the Hoare triple $\{\phi\}S\{\psi\}$ means the execution of S in a state satisfying the pre-condition ϕ will be in a state satisfying the post-condition ψ when it terminates. The conditions ϕ and ψ are first order logical formulae called *assertions*. Hoare proofs are *compositional* in the structure of the language in which the program is written. A judgment $\vdash \{\phi\}S\{\psi\}$ is *valid* if the triple $\{\phi\}S\{\psi\}$ can be proven in the Hoare calculus. The proof rules for partial correctness of Hoare triples are given in Figure 2.2. Partial correctness does not require the program to terminate, whereas total correctness requires the program terminates [51].

2.3.3 Proof-Carrying Code

Proof-Carrying Code (PCC) [79] is a technique used by a host computer system (the code consumer) to automatically verify the safety properties of code that are provided by untrusted agents (the code producer). PCC can be used in the environment of distributed computing where mobile code is allowed. In this environment the code producer on one side of the network produces a software program that is transmitted to the code consumer on another side for execution. The code consumer can apply PCC to verify that the code behaves correctly and satisfied a set of safety rules. The first step of PCC is that the code consumer specified the *safety policy* which describes the conditions for safe behavior of foreign program. Then the code producer generates a

proof that the program adheres to the safety policy. This process is called certification which is a form of verification with respect to the program specification. At last, the code consumer uses a proof checker to check the validity of the program. If the proof is valid, the code consumer will trust that the program is safe to execute. In the original PCC framework, proofs are written in a logic extended with language-specific typing rules which assumes that there is no bug in the typing rules. In the work of Appel [2], *Foundational Proof-Carrying Code* (FPCC) has been proposed to use a foundational mathematical logic to define the semantics of instructions and proof rules. In our work, we use the Coq proof assistant which is a tool for the calculus of inductive construction to implement the FPCC framework.

2.3.4 An Interactive Theorem Prover Coq

We have applied an interactive theorem prover Coq[1], which is based on a logical framework known as the Calculus of Inductive Construction, to formalize auction mechanisms and prove interested properties. Coq allows users to define functions or predicates, to state mathematical theorems and develop interactively formal proofs of theorems, to describe software specifications and develop proofs for desirable properties of the specification. Besides, Coq contains a small certification "kernel" to check the correctness of formal proofs.

Coq uses a specification language which is named *Gallina* to describe declarations and definitions. *Gallina* can be used to represent programs as well as properties of these programs. Coq also provides a proof engine to build proofs using *tactics*, which are commands that are used to build proofs. In Coq, logical propositions are called *Prop*, mathematical collections are called *Set*, and abstract types are named with *Type*.

Coq is also a functional programming language which offers a powerful mechanism for defining new data types and writing programs. Inductive definitions can be used to define data types and their members. For example, the basic data type *boolean* can be defined as the following declaration `bool`. This definition shows that the type `bool` of boolean has members `true` and `false`.

```
Inductive bool : Type :=
    | true : bool
    | false : bool.
```

Definitions in Coq can be used to define non-recursive functions. The function `and` represents the $\wedge$ operation of two boolean variables. In this definition, *pattern-matching* is used over the term `b1` to compare it with different branches.

```
Definition and (b1:bool)(b2:bool) : bool :=
```

[1] http://coq.inria.fr

```
match b1 with
| true => b2
| false => false
end.
```

Recursive functions are defined using the command `Fixpoint`. For example, the function `beq_nat` is defined to determine the equivalence of two natural numbers.

```
Fixpoint beq_nat (n m : nat) : bool :=
    match n with
    | O => match m with
            | O => true
            | S m' => false
            end
    | S n' => match m with
              | O => false
              | S m' => beq_nat n' m'
              end
    end.
```

In Coq, an assertion states a proposition of which the proof is interactively built using tactics. The basic assertion commands for a theorem and a lemma are:

Lemma ident : type

and

Theorem ident : type.

Table 2.2: Most Common Tactics

Tactic	**Effect**
intro.	Introduce one assumption
intros.	Introduce as many assumptions as possible
apply H.	Applies assumption H
induction t.	Perform induction proof over term t
rewrite H.	Rewrite assumption H
destruct t.	Perform case analysis without recursion
omega.	An automatic decision procedure for Presburger arithmetic

After a statement has asserted, Coq needs a proof to validate the statement. A tactic language which is called Ltac is used to code procedures of proofs in Coq. Ltac offers a wide variety of tactics to build proofs of statements, including unfold definitions, check equality, etc. The proof development in Coq is based on a goal-oriented approach. The system keeps track of the current goal and the set of premises. Then, users insert

tactics to adjust the proof state. The proof finishes when we match a state with the conclusion. Table 2.2 illustrates some common used tactics in a proof.

In our work, we have used Coq as a proof assistant for higher order logic to formalize and validate the properties of Vickrey and Combinatorial auctions, as a platform to model imperative languages, and as a logical framework to develop the reasoning system of Hoare Logic.

2.4 Agent Communication

One of the key properties of multi-agent systems is interoperability which requires agents to communicate with each other to fulfill their goals. The communication mechanism among agents contains three aspects: the *Syntax* which shows how the symbols of communication are combined to form grammatical sentences, the *Semantics* which describes what the symbols denote, and the *pragmatics* which represents how the symbols are interpreted. Most of the work of communication in multi-agent systems borrows their inspiration from *speech act theory*, which describes how utterances are used to achieve intensions. In the original work of speech act theory [4], communications among participants are not only information transition, but also actions that change the state of the world. The study of speech act focuses on the pragmatic aspect of languages that is how language is used by people to achieve their goals and intentions by communication. Austin [4] lists three aspects of speech acts: *Locution* which represents the act of making an utterance, *Illocution* which describes the intension of an utterance, and *Perlocution* which represents the action that occurs as a result of an illocution. In the work of Searle [89], five different types of speech acts are identified:

- *Representatives*: is an act commits the speaker to the truth of an expression.
- *Directives*: is an attempt to get the hearer to do something.
- *Commissives*: is an act commits the speaker to some course of action.
- *Expressives*: is an act expresses the mental state of a speaker.
- *Declaratives*: is an act effects the change of state.

In general, there are two components in a speech act: one is a performative verb (e.g. "request"), while the other is the propositional content (e.g. "the desk is clean"). The semantics of speech acts are defined using the *precondition-delete-add list* formalism of planning research in [28]. Using this model, the mental states of agents are defined in terms of beliefs and desires. Consider the semantics of the *Request* act where SPEAKER requests HEARER to do action ACT in Table 2.3.

The first precondition states that a speaker doesn't ask somebody to do something unless he thinks they can do it. The second condition indicates that a speaker doesn't

Table 2.3: The Semantics of Request

Request(SPEAKER,HEARER,ACT)	
precondition:	(1) SPEAKER believes HEARER can do ACT;
	(2) SPEAKER believes
	HEARER believes HEARER can do ACT;
	(3) SPEAKER believes SPEAKER wants ACT.
postcondition:	(1) HEARER believes
	SPEAKER believes SPEAKER wants ACT.

ask somebody unless he believes they can do it. The third condition shows that a speaker doesn't ask somebody unless the speaker wants it. The postcondition of this act states that hearers are aware of the speaker's desire.

Speech act theory inspires the development of Agent Communication Language (ACL). An ACL contains three components: an 'outer' language (pragmatics) to express the primitives, an 'inner' language (syntax) to write the message, and vocabulary (semantics) to describe the meaning of messages. A well-known ACL is "KQML" [25]. The *Knowledge Query and Manipulation Language* (KQML) is a high-level, message-based communication language, which is developed in the project of Knowledge Sharing Effort (KSE). There are three layers in a KQML message: the content layer, message layer, and communication layer. The content layer contains the actual expression in some languages. Sender and recipient can choose any languages they would like to share knowledge. The message layer is used to encode a message which could contain the type of content message or declaration message. A content message is used to describe a speech act (e.g. query or assertion) that involves some sentences in a given content language. A declaration message is used to provide information about the content messages that an agent will generate and would like to receive. The communication layer is used by agents to exchange packages which are wrappers around messages that specify communication attributes, such as the information of the sender and recipient.

KQML provides different kinds of performatives. For example, the performative *ask-if* represents that a sender wants to know if the content is in the receiver's knowledge base. Those performatives in KQML are identified by reserved parameters. Table 2.4 summaries the reserved parameters and their meanings [57]. The semantics of each performative can be defined in terms of *preconditions* which indicate the states for a sender to send a performative and a receiver to receive and process it, *postconditions* describe the states after the success of sending and processing messages, and *completion conditions* indicate the final state of a conversation and the fulfillment of the original intention of the performative [58].

Based on speech acts theory, the Foundation for Intelligent Physical Agents (FIPA) has developed another standard language *FIPA ACL*. The message structure of *FIPA*

Table 2.4: Reserved Parameters and Their Meanings in KQML

Keyword	*Meaning*
:sender	the actual sender of the performative
:receiveer	the actual receiver of the performative
:from	the origin of the performative in :content when forward is used.
:to	the final destination of the performative in :content when forward is used.
:in-reply-to	the expected label in a response to a previous message (same as the value of the previous message).
:reply-with	the expected label in a response to the current message.
:language	the name of the representation language of the :content.
:ontology	the name of the ontology (e.g., set of term definitions) assumed in the :content parameter.
:content	the information about which the performative expresses an attitude.

ACL is composed of a set of parameters, such as *performative* (Type of communicative acts), *sender* (Initiator of the message), and *content* (Content of the message). *FIPA ACL* has 22 performatives and the syntax of these performatives is similar to KQML. The semantics of *FIPA ACL* are defined based on the mental attitudes of agents, such as belief, desires and intensions, via the SL (Semantic Language). The meaning of those performatives is given in terms of Feasibility Conditions (*FPs*) and Rational Effects (*REs*). *FPs* represent the conditions that should be true when a sender can send a message. *REs* describe the expected outcome of a performative. However, the *REs* cannot be guaranteed from sending the message. The semantics of performative *request* is defined as follows.

```
<i,request(j,A)>
FP: (and(B i (capable_of j A)))
        (not (B i (I j (done A))))
RE: (done A)
```

The `FP` says `i` believes that `j` is capable of action `A` and does not believe that `j` intends to do `A`. The `RE` describes that this performative is used to get `A` done. The details of *FIPA ACL* can be found in [38]. The languages of KQML and FIPA ACL have been widely applicable in multi-agent systems. However, agents participant in conversations have too many choices to make an utterance by using these languages, and thus cause the problem of state-space explosion for dialogues [73]. To avoid state-space explosion, rule-governed interactions between two or more players has been studied and this led

to the development of formal dialogue games.

2.4.1 Dialogue Games

Dialogue games are made up of a set of communicative acts and set of rules to determine the sequence of different acts. In the well-known work of Walton and Krabbe [111], dialogues are categories into different types according to the initial situation of dialogues, the personal aims of participants and the main goal of the dialogue. The six dialogue types in this typology are: *persuasion dialogues* (where one participant persuades another to resolve conflicts of opinion); *inquiry dialogues* (where participants collaborate to find evidence or proof for specific knowledge); *negotiation dialogues* (where participants need to reach a deal that resolves their conflicting interests); *information seeking dialogues* (where participants aim to acquire or provide knowledge); *deliberation dialogues* (where participants aim to make decisions about what actions to adopt); *eristic dialogues* (where participants quarrel verbally to vent perceived grievances). The mixtures of different types of dialogues are called embedded dialogues. For example, in an online trading scenario, the buyer needs to seek information from sellers, and then the seller may need to persuade the buyer to buy a specific item by introducing some features of this product.

The syntax of a dialogue games contains the utterances which agents can make and the rules to determine the order of making utterances. The rules regarding of assertions are also included in the syntax of a dialogue game, because these rules possibly influence the order of utterances. These assertions given by participants are stored as a public readable database which forms the *commitment store* of a dialogue game. An utterance can be divided into two layers: the outer layer is made up based on the locutions and the inner layer contains the topics of discussion. In the work of McBurrney and Parsons [71], a framework has been proposed to represent the specification of a dialogue game.

Chapter 3

Motivation Example and Proposed Certification Framework

Software agents are involved in buying and selling items or services in agent mediated e-commerce systems. The success of an agent mediated e-commerce system relies on the trustworthiness between agents. To meet the requirements of trustworthiness, game theory mechanisms which guarantee desirable properties have been carried out. For example, these mechanisms can guarantee that the e-commerce system is strategy-proofness (which means that there is a dominant strategy for agents) or false-name-proofness (which means that agents cannot benefit from submitting multiple bids under false names). The next major challenge in agent mediated e-commerce systems is to enable software agents to comprehend the rules and social norms which govern the behavior of new institutions in open, heterogeneous environments. These facilitate agents to be capable of making rational decisions in the marketplace. Existing work has addressed interoperability at the communication level (with agent communication languages such as FIPA-ACL [38], and RDF [55] to underpin recent developments within the Semantic Web [15]) thus *allowing agents to communicate*, the decision of whether or not *the communication is meaningful* is still an open challenge. Agents may understand how to conduct their behavior in certain familiar scenarios, and strategically bid in marketplaces that adhere to certain rules (e.g., an English or Dutch auction). However, such strategies may not be applicable to other markets, such as those that based on Vickrey auctions. Within an open and dynamic environment (such as e-commerce), agents might encounter a variety of auction houses, which forms part of an agent mediated e-commerce scenario. It is therefore necessary for the agent to be able to acquire a deeper model of the marketplaces which agents engaged in (other than simply relying in simple classifications). This enables agents rationally determine whether or not they should engage in the marketplace.

Agents should be able to query and comprehend the rules that govern an auction

house, and verify desirable properties such as privacy, security, and economics. This research focuses on the economic properties, by looking at specifying and verifying game-theoretic properties for the single item and multiple items online auctions. An important game-theoretic property is strategy-proofness, which means the existence of a dominant strategy for the players indicates a strategy that is optimal regardless of the game configuration. For example, truthful bidding can be the dominant strategy in certain auction settings. Section 3.1 introduces a verification example of Vickrey auctions, and Section 3.2 describes a certification framework.

3.1 A Pilot Example: Verification in a Vickrey Auction

In a single item Vickrey auction, bidders simultaneously submit their bids to an auctioneer. The winner of a Vickrey auction is the highest bidder and the payment is the second-highest bid. In this section, the focus is expressing a mechanism (a Vickrey auction) and game-theoretic proofs in a machine checkable formalism. We have used Coq [105], an interactive theorem prover, to carry out the formalizations of a Vickrey auction in Section 3.1.1. Then, different bidding strategies have been specified followed by the proofs of a dominant strategy for each bidder in Section 3.1.2.

3.1.1 Formalization of a Vickrey Auction within Coq

Consider n bidders with values $v_i \geq 0, i = 1, ..., n$ who are competing for one item. If bidder i wins the item and pays p, then bidder i's utility is $v_i - p$. A formal definition of a Vickrey auction is given here.

Definition 1. *Given one item and n bidders, the rule of a* ***Vickrey auction*** *is as:*

- *Bidders make bids* $b = (b_1, ..., b_n)$ *simultaneously and all these bids are stored in a list.*
- *The highest bidder wins the item and pays the second highest bid.*
- *If there are more than one highest bidders, then the one that close to the head of the list is selected as the winner and pays own bid.*

To specify a single item auction, a type of `Item` is defined, with one member `item` to indicate the item that was sold.

```
Inductive Item : Type := item.
```

The following objects are also defined as types: `bidder_ID`, `bid`, `valuation` to represent respectively the sets of agents, their bids, and their valuations. Note that both `bid` and `valuation` are declared as natural number to restrict that each bidder should give nonnegative values to bids and valuations in the auction.

```
Definition bidder_ID := nat.
Definition bid := nat.
Definition valuation := nat.
```

We then describe an inductive relation `HasBid` binding agents with their bids and provides one function `getBid` to return the bid for a given relation.

```
Inductive HasBid : Type :=
    hasBid : bidder_ID -> bid -> HasBid.
Definition getBid (p : HasBid) : nat :=
    match p with
    | hasBid bidder bid => bid
    end.
```

Another relation `HasVal` binding a valuation with an agent, and the function `getVal` returns the value of a valuation for a given `HasVal` relation.

```
Inductive HasVal : Type :=
    hasVal : bidder_ID -> valuation -> HasVal.
Definition getVal (p : HasVal) : nat :=
    match p with
    | hasVal bidder val => val
    end.
```

In a Vickrey auction, each agent has three different strategies: bidding truthfully (the bid given by an agent equals to its valuation), bidding beyond the valuation, and bidding below the valuation. We define a strategy as a mapping from `valuation` to `bid`.

```
Definition strategy := valuation -> bid.
```

The three bidding strategies are defined as propositions. The definition of truthful bidding is `strategy_is_truthful`. In this definition, the input is a variable `s` with type of `strategy`, while the output is a proposition that the bid of the strategy equals to `v`.

```
Definition strategy_is_truthful (s: strategy) : Prop :=
    forall v,
        v = s v.
```

The definitions `strategy_is_up_bidding` and `strategy_is_below_bidding` defines the strategies that bid beyond and below valuation, respectively.

```
Definition strategy_is_up_bidding (s: strategy) : Prop :=
    forall v,
        v < s v.
Definition strategy_is_below_bidding (s: strategy) : Prop :=
    forall v,
        v > s v.
```

Besides these three definitions for the bidding strategy in a Vickrey auction, we also define a proposition `strategy_is_not_truthful` represents non-truthful bidding.

```
Definition strategy_is_not_truthful (s: strategy) : Prop :=
    forall v,
        v <> s v.
```

In the following definition `hasStrategy`, the constructor `has_strategy` has been applied to arguments `bidder_ID` and `strategy`. This definition binds a bidder with a specific strategy.

```
Inductive hasStrategy :Type :=
    has_strategy : bidder_ID -> strategy -> hasStrategy.
```

To evaluate the payoffs of each agent in a Vickrey auction, we define the function `utility`. This function has two inputs that represents the valuation and bid of an agent and returns an integer. The approach to calculate the value of a utility will be defined in another function `eval_utility`.

```
Definition utility := valuation -> bid -> Z.
```

To compare the utility of each agent, we define a proposition `utility_better` that shows one utility `u1` is better than the other utility `u2`.

```
Definition utility_better (u1 u2: utility) : Prop :=
    forall v b,
        u1 v b >= u2 v b.
```

One property that will be proved is that the utility of each agent is greater or equal to zero in a Vickrey auction. The definition of `utility_nonnegative` states that the return value of this utility is nonnegative.

```
Definition utility_nonnegative (u1 : utility) : Prop :=
    forall v b,
        u1 v b >= 0.
```

The winner of a Vickrey auction is the one with the highest bid, or if there is more than one highest bidder, the one near the head of the bidding list will be the winner. We define the recursive function `get_maximal_bid` to calculate the highest bid. The input of this function is a list of strategy `sl`, which represents a list of bids that submitted by every agents, and a variable `v` with type of `valuation`.

```
Fixpoint get_maximal_bid (sl: list strategy) (v: valuation): bid :=
    match sl with
    | nil => 0
    | cons s sl' => let cb := s v in
          let max_among_remains := (get_maximal_bid sl' v) in
          if (bge_nat cb max_among_remains)
          then cb
          else max_among_remains
    end.
```

The function `eval_utility` is defined to calculate the utility of an agent. In this definition, `s` represents the strategy of an agent, while `sl_others` contains all the bidding strategies of other agents. If the bid given by this agent is greater than any other bid, it will get the utility of $v_i - p_i$, where v_i is the valuation of this agent, and p_i is the second highest bid. Otherwise, this agent will get utility of zero.

```
Definition eval_utility (s: strategy) (sl_others: list strategy) :
    utility :=
        fun (v: valuation) (b: bid)
           => let cb := s v in
              let max_among_others := get_maximal_bid sl_others v in
              if (bgt_nat cb max_among_others)
              then ((Z.of_nat v) - (Z.of_nat max_among_others))
              else 0.
```

3.1.2 Proof of Incentive Properties of a Vickrey Auction within Coq

The first property established in Coq is *non-negative utility* for all bidders.

Theorem 1. *In a Vickrey auction, every truthtelling bidder is guaranteed non-negative utility.*

The related Coq formalization is as follows.

```
Theorem nonnegative_utility : forall (i : bidder_ID)
       (s : strategy)(sl_others : list strategy),
           s = truthful_strategy
               ->  utility_nonnegative (eval_utility s sl_others).
```

Proof. In this theorem, `s` represents the strategy of a bidder, `sl_others` is the strategy of other bidders. Given `s` as the truthful bidding strategy in the premise, a bidder will get non-negative utility. Function `eval_utility` is used to calculate the utility of a bidder. In this proof, each loser gets a utility of 0. If bidder i is the winner, then its utility is $v_i - p_i$, where v_i is the highest bid and p_i is the second highest bid, thus $p_i \leq v_i$. Hence, $v_i - p_i \geq 0$. □

The second property established is that truthful bidding is a weakly dominant strategy in a Vickrey auction.

Definition 2. *In an auction, a strategy profile s_i^* is **weakly dominant strategy** for player i if*

$$u_i(s_i^*, s_{-i}) \geq u_i(s_i, s_{-i}) \forall s_i \in S_i, s_{-i} \in S_{-i}$$

for some $s_{-i} \in S_{-i}$ for any $s_i \neq s_i^*$.

Definition 3. *In an auction, $s^* \in S$ is a **dominant strategy equilibrium** if for every player $i \in N$,*

$$u_i(s_i^*, s_{-i}) \geq u_i(s_i, s_{-i}) \forall s_i \in S_i, s_{-i} \in S_{-i}$$

.

Theorem 2. *In a Vickrey auction, truthful bidding ($b_i = v_i$) is a weakly dominant strategy.*

The Coq formalization of this theorem is:

```
Theorem dominant_strategy:
  forall (s1 s2: strategy) (bi vi : nat)
       (sl_others : list strategy) (u1 u2: utility),
            s1 = truthful_strategy
                -> strategy_is_not_truthful s2
                -> u1 = eval_utility s1 sl_others
                -> u2 = eval_utility s2 sl_others
                -> utility_better u1 u2.
```

In this theorem, we get two variables `s1` and `s2` with the type of `strategy`, two natural number `bi` and `vi` represent the bidding and valuation of a bidder, the variable `sl_others` denotes a list of strategies while `u1` and `u2` are in the type of `utility`. We have four premises in this theorem. First, `s1` is a truthful bidding strategy. Second, `s2` is not a truthful bidding strategy. Third, `u1` is the utility with truthful bidding strategy `s1`. The fourth premise means that `u2` is the utility of untruthful bidding. Under these premises, the theorem states that `u1` is greater or equal to `u2`, i.e., the utility of truthful bidding is better than untruthful bidding.

Proof. The proof is implemented by case analysis. Suppose participant i bids in `s1`, i.e. truthful bidding. There are two cases:
(1) i wins. This means that `vi` is the highest bid in the auction. As the definition of `eval_utility`, the value of `u1` equals to `vi` subtracts the second highest bid. Suppose i uses `s2` to bid and gets utility of `u2`. There are two cases:

(a) i wins. Then `u1` and `u2` are equal.

(b) i loses. Then `u2` becomes 0, which is less or equal to `u1`.

(2) i loses. This implies `u1` equals to 0, and `vi` less than the highest bid. There are also two cases for i to bid untruthfully:

(a) i loses. The value of `u2` is 0, i.e. the same as `u1`.

(b) i wins. Bidder i submits a higher value and `u2` becomes a non-positive value, i.e. `u2` less or equal to `u1`. □

By applying to all bidders, truthful bidding constructs an equilibrium in weakly dominant strategies.

Theorem 3. *In a Vickrey auction, truthful bidding ($b_i = v_i$) is a weakly dominant strategy. Hence $b^* = v$ is a dominant strategy equilibrium.*

The Coq formalization of this theorem is:

```
Theorem dominant_strategy_equilibrium:
    forall (i : bidder_ID)(v : nat)
        (s1 s2: strategy)(sl_others : list strategy)
        (u1 u2: utility) (h1 : hasStrategy)
        (h2 : hasStrategy),
            h1 = has_strategy i s1
            ->  h2 = has_strategy i s2
            -> s1 = truthful_strategy
            -> strategy_is_not_truthful s2
            -> u1 = eval_utility s1 sl_others
            -> u2 = eval_utility s2 sl_others
            -> utility_better u1 u2.
```

The proof of this theorem is similar to the above theorem.

3.2 Certification Framework

In the above example, some desirable properties of a Vickrey auction within the proof assistant Coq have been formalized and proved. This example illustrates the feasibility of the idea that the economic properties of an auction can be constructed for future certification by buyer agents. In order to effectively enable automatic checking of

desirable properties, we need to take into account the fact that software agents have limited computer resources and constrained in their reasoning. The main difficulty for a software agent is to find the best possible or optimal bidding strategy on its own or to optimize its utility out of various strategies in the same way as humans perform. To address this difficulty, the specification of auction protocols and proofs are published in a machine-readable formalism, and then automatic checking is facilitated by the software agents which reduces the computational complexity. To achieve this, we rely on the Proof-Carrying Code (PCC) idea as it allows us to shift the burden of proof from the buyer agent to the auctioneer who can spend time to prove a claimed property once for all. Thus it can be checked by any agent who is willing to join the auction house.

The PCC is a paradigm that enables a computer system to automatically ensure that a computer code provided by a foreign agent is safe for installation and execution. A weakness of the original PCC was that the soundness of the verification condition generator is not proved. To overcome this weakness, Foundational PCC (FPCC) [2] provides us with stronger semantic foundations to PCC by generating verification conditions directly from the operational semantics. Figure 3.1 illustrates our framework that adapts FPCC to certify auction properties. At the producer or auctioneers side, we have the specifications of the auction mechanism along with the proofs of desirable properties in a machine-checkable formalism in the form of a Coq file. The certification procedure works as follows. The buyer agent arriving at the auction house can download its specification and the claimed proof of a desirable property. Then, the buyer requests the proof checker *coqhk*, which is a standalone verifier for Coq proofs, to the auctioneer. The auctioneer provides the address of the proof checker which is stored by a trusted third party (e.g., the home page of Coq) to the buyer. After the proof checker is installed to the consumer side, the buyer can perform all verifications of claimed properties of the auction before deciding to join and with which bidding strategy.

The objective of this work is to enable software agents in an open e-commerce system to understand a published specification of a trading protocol, to request for a formal or informal evidence of desirable properties, and automatically check those evidences. Finally, agents decide whether or not to participate in the trading. Furthermore, agents need to automatically composite web services to fulfill their requirements. To achieve our research objective, the following issues should be addressed.

1. Machine understandable specifications of a set of trading protocols that related to e-commerce systems. These trading protocols can be expressed as a format of Web services.

2. Using a logical framework to formally specify trading protocols and provide proofs of desirable properties of the trading protocols.

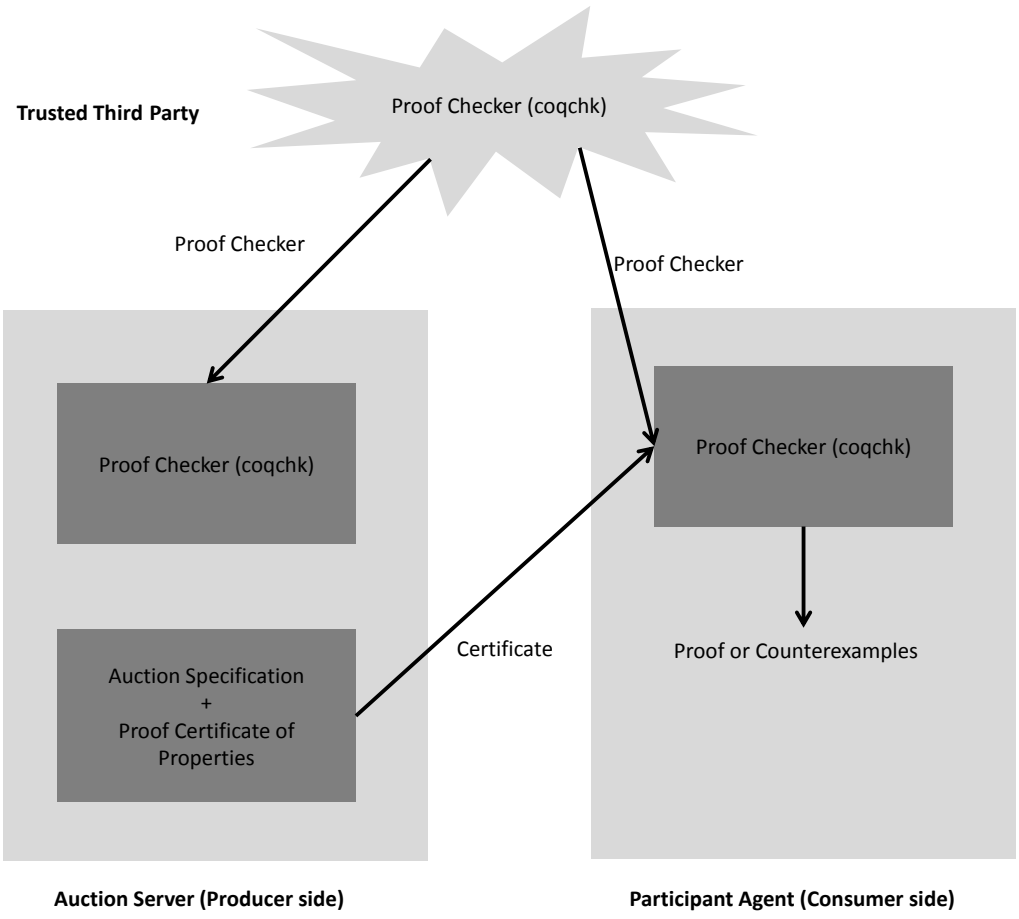


Figure 3.1: The Adapted FPCC Certification Paradigm

3. Mapping the original specification to the specification that expressed in the logical framework.

4. An extension of the PCC paradigm that enables buyer agents to automatically check the correctness of proofs that is constructed by the seller agents.

5. An interaction mechanism that supports automatically communication between buyer agents and seller agents.

6. A mechanism that supports buyer agents to automatically composite Web services that fulfill the requirements of a trading.

A certification framework which combines a series of techniques is proposed to meet the above mentioned issues. The first issue is to use a machine-understandable language, such as OWL-S, to describe the specification of auction mechanisms which have been widely used in off-line and online trading. Secondly, the above specification will be translated to another specification, which is formalized within Coq that provides a meta-language to specify programs. The translation process preserves the semantics of the given specification. Thirdly, we will construct the proofs of desirable properties of the specification within the interactive theorem prover Coq. Fourthly, we will build dialogue games using the agent development platform JADE to enable automatic communication between buyer agents and auctioneer agents. The FPCC paradigm will be integrated into the dialogue games to make buyer agents check the correctness of proofs. At last, dialogue games will be applied to realize automatic services composition in e-commerce systems.

3.3 Summary

In this chapter, some desirable properties of a Vickrey auction have been formalized and proved within an interactive theorem prover Coq. This pivot example shows the feasibility of verifying desirable properties of online trading mechanisms. Then the certification framework has been proposed that combines a series of techniques such as Semantic Web Services, Foundational Proof-Carrying Code (FPCC), interactive theorem proving and dialogue games. It allows an auctioneer to publish the specification of auction mechanisms using a machine-understandable language, and then the auctioneer to translate the above specification to another specification which is formalized within Coq; this translation preserves the semantics of the given specification. It allows the auctioneer to construct proofs of desirable properties of a specification within Coq, and to publish the auction mechanism along with the proofs of desirable properties. It allows the potential buyer agent to read the published protocol, to make sense of it, and at will, to check the proof of a given property by using a simple trusted checker, which makes the automatic checking procedure computationally reasonable. The potential

buyer makes its decision about whether or not to participate in the auction according to the results of communication with the auctioneer. It allows the composition of Semantic Web Services on the basis of communication between buyer agents and auctioneers.

In the next chapter, the Semantic Web Service language OWL-S will be used to describe a specification of an English auction, and then an imperative language that preserve the semantics of OWL-S within Coq will be defined, followed by the translation of the specification from OWL-S to the imperative language, and the proof of desirable properties of the specification.

Chapter 4

Formal Specification and Verification

A *specification* provides a high-level description of behaviors and properties of a system. Compared to an informal specification which is represented in a non-mathematical form, formal specifications provide a precise description of software systems by using a language whose syntax and semantics are formally defined. The need of formal definitions requires specification languages to express abstract and precise mathematical concepts which can be drawn from set theory, logic and algebra. Formal specifications provide an abstract view of systems which describes *what* a system should accomplish and enable inferring interesting properties from the specification. The formal software verification is to ensure a system behave according to its specification. One verification approach is to use formal proofs to guarantee the properties of a specification. A sound proof system can imply that a program meets its specification for all inputs. Another advantage of formal verification is that the correctness of those proofs is machine-checked.

We consider electronic markets based on single item auctions whose mechanisms are described in machine readable formalism, e.g. OWL-S, so that software agents can understand their rules. However, we would like to enable potential participants to check that the auction house is trustworthy before entering it and bid for items on sale by verifying desirable properties. To construct proofs, we have used the proof assistant Coq. This implies that the auction mechanism needs to be transformed into Coq descriptions in an automated fashion so that proofs can be developed from within Coq. However, bugs in the transformation process may potentially invalidate the assurances almost not gained by certifying certain desirable auction properties. Therefore it has to be guaranteed that the transformed mechanism is semantically equivalent to the original one.

There is already a large body of literature on certifying program transformations [61]. Leroy's compositional approach in certifying a compiler is most relevant to this work: certifying that all phases of the transformation are correct. In our case, we have de-

fined the target language (a WHILE language) within Coq to mirror the input language (OWL-S) and then have used a one to one mapping to establish the correctness of the transformations. This is carried out through using natural semantics of computer programs to establish that the semantics of the target is equivalent to that of the source program. Besides the requirement to have a semantics-preserving transformation, we would like to show how to use the Hoare Logic to prove desirable auction properties from within Coq. This approach is different from the pivot example in Chapter 3, wherein we have relied upon Coq's constructive approach to carry out the verification of some game-theoretic properties of a Vickrey auction mechanism.

In terms of prototype implementation, we have used Antlr [84] to automatically transform an OWL-S auction description into a specified Coq imperative language. Antlr is a widely used tool to read, process, or translate structured data or texts such as source computer programs. To fully certify the program translation step in the verification approach, we can proceed as in Leroy's paper [61] by implementing all transformation phases within Coq and the Ocaml programming language. However, we would like to point out at least an example of such a certifying compiler and focus more on verifying auction properties. Structurally, our transformation is a one-to-one mapping between two kinds of WHILE languages and the background information we have described in Section 2.3.2 is enough to understand the semantics-preserving nature of our transformation framework. Besides, note that Leroy's approach [61] separates the algorithmic and implementation issues in the certification framework.

In this chapter, the first to be introduced is an auction model and the related machine readable description of this model using OWL-S ("OWL for Services") in Section 4.1. The translation of an OWL-S specification into a Coq specification will be explained in Section 4.2. In Section 4.3, the certification of desirable properties will be described. The related work is presented in Section 4.4 and the summary of this chapter is in Section 4.5.

4.1 Auction Model and Description

This section will introduce the background information of game theoretic properties in auctions, and provide the description of an English auction using a Semantic Web Service description language OWL-S.

4.1.1 Auction Model and Incentive Properties

We consider single item online auctions that run over a fixed time period in which the seller may have reserve revenue under which items cannot be sold. This reserve revenue can typically be the reserve price for the item on sale. After a bid, the seller waits for some time before accepting the bid if it yields a revenue that is greater or

equal to the reserve one or waits for the expiry time before deciding whether to reject or accept the bid. In this work, we aim at describing online auctions by using the OWL-S specification language and then translate the resulting description into Coq specifications in order to verify some desirable auction properties. These properties are used to ensure trust in the auction house and therefore make it more attractive to potential buyers or profitable for the seller. For example, an online buyer may want to have a guarantee that the auction is always carried out as specified or that the auction is free from collusion. Some of those properties can be formalized and proved using theorem provers. Therefore, the focus will be on the following incentive properties.

- For a buyer participant, bidding up to its valuation is the optimal strategy.
- The highest bidder wins the auction.
- The payment is indeed the highest bid.

4.1.2 Machine Readable Description

Online auctions for software agents are a good application wherein data can be processed by automated reasoning tools. Logic-based languages are useful tools to model and reason about systems. They allow researchers to specify behavioral requirements of components of a system and to formulate desirable properties for an individual component or the entire system. The Semantic Web [15] enables us to describe and reason about Web services by using *ontologies*. Ontologies are used to formally describe the semantics of terms representing an area of knowledge and give explicit meaning to the information, thus allowing for automated reasoning, semantic search and knowledge management in a specific area of knowledge. OWL [78], a W3C standard, is a description logic-based language that enables researcher to describe ontologies by using basic constructs such as concept definitions and relations between them. It has been used in a wide range of areas including biology, medicine, or aerospace [10, 30]. In this work, it is advocated to use Semantic Web for at least the following reasons.

- It is expressive enough so that a range of auction mechanisms can be described in it.
- It provides us with a machine readable formalism enabling software agents to understand auction mechanisms.
- There is a scope for semantics interoperability for heterogeneous software agents engaging in an auction.

An auction is viewed as a Web service with enough logical attachments, enabling a software agent to understand the auction rules and to carry out verifications of claimed properties.

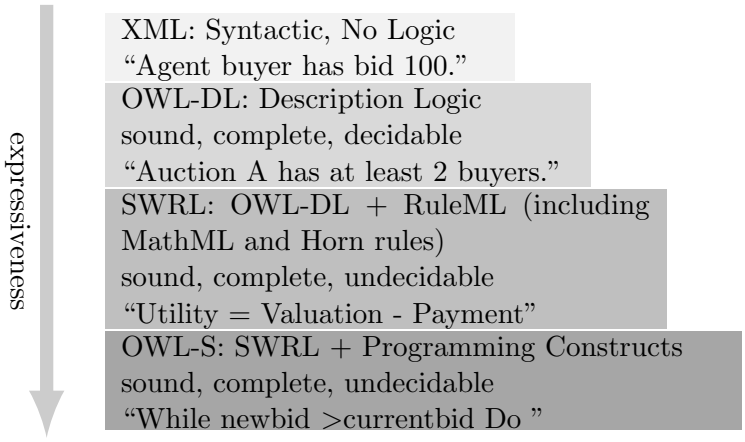
Logic-based languages are usually chosen for their *expressivity* or on the fact that their underlying logic is *sound*, *complete* or *decidable*. Expressivity provides powerful constructs to describe things that may not be otherwise expressed. Soundness ensures that if a property ϕ can be deduced from a system (a set of statements) Γ ($\Gamma \vdash \phi$), then ϕ is true as long as Γ is satisfied ($\Gamma \models \phi$). Completeness states that any true statement can be established by proof steps in the logic's calculus. Formally $\Gamma \models \phi$ implies $\Gamma \vdash \phi$. A logic is decidable if we can construct a terminating algorithm that decides for any well-formed formula if it is true or false.

To enable automated reasoning in the auction system, we have used the ontology language OWL-S, which is corresponding to OWL, to build up the auction ontology. See [8] for further details.

The OWL-S Description Language

In this section, we argue that we can describe online auction houses as Web services by using the OWL-S language, which provides us with machine readable formalism and logical reasoning capabilities for software agents. We will start by showing that OWL-S is expressive enough to enable us to describe online auction mechanisms.

To describe online auctions, we may ask why not use XML (Extensible Markup Language) as a description language for this task. XML provides a syntactic approach but no logical basis for reasoning. The meaning of the relationships between XML elements cannot be encoded. A language that builds upon XML and allows for reasoning is the OWL-DL (Web Ontology Language), which is based on DL (Description Logic). Description logics are a family of logics that are decidable fragments of first-order logic. OWL-DL is sound, complete, and decidable but with limited expressivity. For example, we cannot express arithmetic statements that 'The winner's utility is the value of valuation minus payment'.



To extend OWL-DL, the SWRL (Semantic Web Rule Language) combines OWL-DL with RuleML that includes among others, MathML and Horn rules. As a result, SWRL is more expressive than OWL-DL but SWRL is not decidable [86]. However,

in SWRL, we cannot express the statement that 'while the newbid is greater than the currentbid do'. Furthermore, auction mechanisms can be viewed as functions with inputs, outputs, preconditions or post-conditions. They may contain complex programming constructs such as branching or iterations. OWL-S enables us to describe online auctions as Web Services.

An OWL-S description is mainly composed of a service *profile* for advertising and discovering services; a *process* model, which describes the operation of a service; and the *grounding*, which specifies how to access a service. In our case, the process contains information about inputs, outputs, and a natural language description of the auction, e.g., this is an English auction. The grounding contains information on the service location so that an agent can run the service by using the OWL-S API. The process model is described as follows.

```
define composite process Auction
(inputs: (...)
outputs: (...)
preconditions: (...)
results: (...)
)
{ // Process's Body
WinDetermAlgo(...);
PayeAndUtil(...) }
```

```
<process:CompositeProcess rdf:ID="EnglishAuction">
<process:hasInput> ...
<process:hasOutput>
<process:hasPrecondition> ...
<process:hasResult> ...
...
<process:CompositeProcess rdf:ID="WinDetermAlgo"> ...
<process:CompositeProcess rdf:ID="PayeAndUtil"> ...
</process:CompositeProcess>
```

The auction process model is basically composed of inputs, outputs, preconditions, results and a composition of two processes, which are the winner determination algorithm `WinDetermAlgo` and `PayeAndUtil` that calculates the payments as well as the utilities for the buyer agents. The following sample description provides a more detailed example from `WinDetermAlgo`. It shows that is if `CurrentBid` is less than `NewBid`, then the value of `CurrentBid` and `CurrentWinner` will be updated.

```
<process:If-Then-Else rdf:ID="If-Then-Else">
        <process:ifCondition>
            <expr:SWRL-Condition>
                    <expr:expressionObject>
                    <swrl:AtomList>
                        <rdf:first>
                            <swrl:BuiltinAtom>
                                <swrl:builtin rdf:resource="&swrlb;#lessThan"/>
                                <swrl:arguments>
                                    <rdf:List>
                                        <rdf:first rdf:resource="#CurrentBid"/>
                                        <rdf:rest>
                                            <rdf:List>
                                                <rdf:first rdf:resource="#NewBid"/>
                                                <rdf:rest rdf:resource="&rdf;#nil"/>
                                            </rdf:List>
                                        </rdf:rest>
                                    </rdf:List>
                                </swrl:arguments>
                            </swrl:BuiltinAtom>
                        </rdf:first>
                        <rdf:rest rdf:resource="&rdf;#nil"/>
                    </swrl:AtomList>
                    </expr:expressionObject>
            </expr:SWRL-Condition>
        </process:ifCondition>
        <process:then>
```

```
            <process:Sequence>
                <process:components>
                    <process:ControlConstructList>
                        <list:first rdf:resource="#UpdateCurrentBid"/>
                        <rdf:rest>
                            <rdf:List>
                                <rdf:first rdf:resource="#UpdateCurrentWinner"/>
                                <rdf:rest rdf:resource="&rdf;#nil"/>
                            </rdf:List>
                        </rdf:rest>
                    </process:ControlConstructList>
                </process:components>
            </process:Sequence>
        </process:then>
        <process:else>
            <process:Sequence>
                <process:components>
                    <process:ControlConstructList>
                        <list:first rdf:resource="#Skip"/>
                        <list:rest rdf:resource="&list;#nil"/>
                    </process:ControlConstructList>
                </process:components>
            </process:Sequence>
        </process:else>
</process:If-Then-Else>
```

4.2 Automated Program Translation

In Section 4.1.2, we have sketched how online auction mechanisms can be described using the OWL-S description language in order to have machine-understandable protocols by software agents. Furthermore, we have illustrated [9] that auction mechanisms can be specified within Coq so as to develop machine-checkable proofs of desirable mechanism properties that can be automatically verified by software agents [9]. To bridge the gap between these two processes, we have used Antlr to systematically transform OWL-S code into Coq specifications by

1. Identifying a subset of the OWL-S language formed by key constructs used in our description of auction mechanisms; this basically mirrors a WHILE-language for imperative programming.

2. Translating these OWL-S constructs into a tailored subset of Coq specifications so that an OWL-S program or logical formula can be transformed into a Coq one in an automated fashion.

This translation is semantics-preserving since it is a one-to-one mapping from the source language (a subset of OWL-S) into the target language (a specialized Coq WHILE-language).

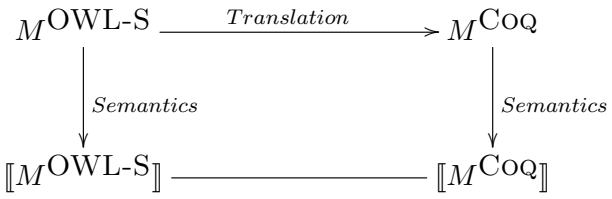
4.2.1 Translation Architecture

The structure of the translator is an Antlr grammar file (`owlsmall.g`), a Java class (`Olws2Coq.java`) that transforms an OWL-S parse tree into a valid Coq parse tree

and a main program that launches the generated parser and the parse tree transformer for an input OWL-S code.

ANTLR [84] is a powerful tool that takes as input a formal language description or EBNF grammar to generate a parser for that language along with tree-walkers that can be used to visit the nodes of those trees and run application-specific code. For an input code in the input language, it generates appropriate representations as parse trees that can be transformed into parse trees for a target language and be unparsed or pretty-printed. ANTLR parsers use the alleged Adaptive LL(*) that performs a just-in-time analysis of the input grammar in lieu of analyzing it statically before execution. It is a widely used tool to read, process, or translate structured data or texts such as source computer programs.

We have proceeded by translating an OWL-S process model, which includes variable declarations, inputs, outputs, preconditions, results, and IF and WHILE constructs, into the target language. In the following diagram, we illustrate how the transformation can preserve the semantics of the original mechanism: a mechanism M, written in OWL-S, is translated into a COQ imperative language description and we would like to ensure that the semantics of the target is equivalent to that of the source code. In this way, true or false properties that are established from within COQ can be inferred back into the OWL-S description.



Syntax and Semantics of the Coq Imperative language

We have used COQ as a meta-language to define the syntax and semantics for the simple imperative language described in Section 2.3.2. Let us start by defining few basic variables to be used in future definitions. A `bidder` is defined as a type and the relation `bidderpair` binds a `bidder` with a `bid`, which is a natural number.

```
Variable bidder : Type.
Inductive bidderpair : Type :=
            bpair : bidder -> nat -> bidderpair.
```

Variables are represented by identifiers `Id` that are mapped into natural numbers in an obvious and incremental way. For simplicity, we assume that all variables are global and this does not conflict with the notion of variable scope in OWL-S.

```
Inductive id : Type :=
             Id : nat -> id.
```

The `state` of all variables at some point in the execution of a program is represented as a mapping from identifiers to natural numbers.

```
Definition state := id -> nat.
```

After a variable and a state are defined, the arithmetic and Boolean expressions are defined. Each definition is composed of two functions. One function is to define the syntax and the other is to define the semantics. The syntax of the arithmetic expressions contains basic data types (e.g. bidderpair (`ABidderp`), numbers (`ANum`) and identifier (`AId`)), arithmetic expressions (e.g., addition (`APlus`), subtraction (`AMinus`) and multiplication (`AMult`)) and operations on a List data structure (e.g., `AHeadb`, `ATail`, `ACons` and `ANil`). The semantics of these mathematical operations are defined as a relational function. For example, the operation `AHeadb` returns the bid of the first bidder in a list.

The syntax of the Boolean expressions defines two basic types: True (`BTrue`) and False (`BFalse`). It also contains the syntax of comparison operators, such as equality (`BEq`), less than (`BLt`) and negation (`BNot`). Another term in this syntax is `BIsCons`, which is used to check whether a list is empty or not. The semantics for the Boolean expressions are also defined as a relational function. For readability, we do not show the Coq code for these two expressions. We finally define the syntax of our mini-language as in the Coq definition of `com` below for commands such as skip, assignment, conditional statements, sequences, and loops. To ease up the presentation, we have introduced the `Notation` declarations to abbreviate these commands.

```
Inductive com : Type :=
  | CSkip : com
  | CAsgn : id -> aexp -> com
  | CSeq : com -> com -> com
  | CIf : bexp -> com -> com -> com
  | CWhile : bexp -> com -> com
  | CRepeat : com -> bexp -> com.
```

```
Notation "'SKIP'" :=
  CSkip.
Notation "c1 ; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "X '::=' a" :=
  (CAsgn X a) (at level 60).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
Notation "'REPEAT' e1 'UNTIL' b2 'END'" :=
  (CRepeat e1 b2) (at level 80, right associativity).
```

The following definition `ceval` defines the semantics of commands in our Coq imperative language. For example, `ESeq` means that the state will change from `st` to `st'` after executing command `c1`, and after running command `c2`, the state moves to `st''`.

```
Inductive ceval : state -> com -> state -> Prop :=
      | ESkip : forall st,
          ceval st SKIP st
      | EAss  : forall st a1 n X,
          aeval st a1 = n ->
          ceval st (X ::= a1) (update st X n)
      | ESeq : forall c1 c2 st st' st'',
          ceval st c1 st' ->
          ceval st' c2 st'' ->
          ceval st (c1 ; c2) st''
      ...
```

Since we have defined the syntax and semantics of all the components of our Coq imperative language, we can proceed by showing how we have translated the OWL-S into the Coq descriptions.

Translating Pre- and Post- conditions and Effect

Hoare Logic is used as a paradigm to implement program verification. On the one hand, it provides a way to describe the pre/post conditions of programs by defining Hoare triples. On the other hand, it provides compositional proof rules to prove the validity of Hoare triples. To build up the Hoare triples, we need to make *assertions* about properties that hold during the execution of a program. An `Assertion` is defined as a proposition indexed by a state.

```
Definition Assertion := state -> Prop.
```

A Hoare triple is composed of three components: the precondition `P`, command `c` and post-condition `Q`. As recalled in Section 2.3.2, a Hoare triple `{P} c {Q}` is valid iff the claimed relation among `P`, `c` and `Q` is true. This means that if a command `c` starts in a state where `P` is true, it will move to a state wherein `Q` is true when `c` terminates.

```
Definition hoare_triple (P:Assertion)
                (c:com) (Q:Assertion) : Prop :=
        forall st st',
            c / st || st'  ->
            P st  ->
            Q st'.

Notation "{{ P }}  c  {{ Q }}" := (hoare_triple P c Q) (at level 90, c at next level).
```

The proof rules of Hoare logic provide a compositional way to prove the validity of Hoare triples. For each command that was defined in `com`, we have constructed its inference rule in the usual way. These inference rules are used to prove the desirable property in Section 4.3. The precondition and effect in OWL-S is mapped to the precondition and postcondition of Hoare triples, respectively.

Translating SWRL into our Coq Imperative Language

The Semantic Web Rule Language (SWRL) is a proposed language to express rules in the Semantic Web. SWRL is used to represent the pre/post-conditions in OWL-S, and it is also used to express comparison and arithmetic operations. In our project, we have just selected a subset of the SWRL, which are used to build up the auction mechanism. The translation of these operations into our Coq imperative language is shown in Table 4.1.

Table 4.1: The Mapping of SWRL and Coq definitions

	SWRL	Coq definitions
Comparison	`swrlb:equal` `swrlb:lessThan`	`BEq` `BLt`

Arithmetic Operations	`swrlb:add`
	`swrlb:subtract`
	`swrlb:multiply`
		`APlus`
		`AMinus`
		`AMult`

List Operations	`swrlb:listConcat`
	`swrlb:rest`
	`swrlb:empty`
		`Acons`
		`ATail`
		`BIsCons`

To illustrate this translation, let us consider the addition of 5 and 3. This operation can be expressed as `swrlb:add(?x,5,3)` in SWRL, while the related Coq imperative language formula should be `X ::= (APlus (ANum 5) (ANum 3))`.

Translating Variable Declarations

In OWL-S, variable declarations are part of *Input*, *Output* and *Local*. As mentioned in the OWL-S standards document, the variables of input, output and local have the same scope as the entire process they occur in. The grammar of these three elements is defined as follows.

```
            <!-- inputs -->
inputs ::= '<process:hasInput>'
                input '</process:hasInput>' inputs;
input ::= '<process:Input
                rdf:ID=' quotedString '>'
                    parameterType '</process:Input>';

            <!-- outputs -->
outputs ::= '<process:hasOutput>'
                 output '</process:hasOutput>' outputs;
output ::= '<process:Output
                 rdf:ID=' quotedString '>'
                     parameterType '</process:Output>';

            <!-- locals -->
locals ::= '<process:hasOutput>'
                local '</process:hasOutput>' locals;
local ::= '<process:Loc
                rdf:ID=' quotedString '>'
                    parameterType '</process:Loc>';

parameterType ::= '<process:parameterType
                       rdf:datatype=' xsdURI '>'
                           type '</process:parameterType>';
```

The keyword `type` in the line of `parameterType`, could be any XML data type or the type of objects that are defined as an OWL Class. The corresponding variable declaration in our Coq imperative language is defined as:

```
Definition VARIABLE : id := Id NUM.
```

In this declaration, `VARIABLE` is the variable name and `NUM` is a unique identifier for this variable.

Translating an Assignment

The OWL-S assignment operation is defined as an element of data flow, output binding, Set or Produce operations. Here, we give one example of how we map an output binding to an assignment operation in our Coq imperative language. The grammar of an output binding can be described as below.

```
        <!-- OutputBinding -->
OutputBinding ::= '<process:OutputBinding>'
                        toVar valueSource
                            '<process:OutputBinding>' OutputBinding;

toVar ::= '<process:toVar
                rdf:resource=' quotedString '/>';

valueSource ::= '<process:ValueOf>'
                    fromProcess theVar'</process:ValueOf>';

fromProcess ::= '<process:fromProcess
                    rdf:resource="&process;#ThisPerform"/>';

theVar ::= '<process:theVar
                rdf:resource=' quotedString '/>';
```

The keyword `quotedString` in the line of `theVar` is defined as the combination of characters. The related assignment operation in our Coq imperative language is in the format of `''X ':=' a''`. The pre- and post- conditions of an OWL-S description are translated to comments in our Coq imperative language. Since we have defined the Hoare Logic rules in Coq, when we develop a proof, we can introduce the related pre- or post- conditions to this proof and verify their correctness for example.

Translating Control Constructs

Composite processes in OWL-S can be decomposed into atomic processes by using control constructs. OWL-S contains ten control constructs including Sequence, Split, Choice, If-Then-Else and Repeat-While. To give an idea of the translation of these complex structures, we describe how we have carried out the translation of the `If-Then-Else` and `Repeat-While` control constructs. The grammar of the control constructs `If-Then-Else` and `Repeat-While` is defined as follows.

```
        <!-- If-Then-Else -->
ifthenelse ::= '<process:composedOf>'
```

```
                    '<process:If-Then-Else
                            rdf:ID=' quotedString '>'
                                  if then else
                    '</process:If-Then-Else>'
                '</process:composedOf>';

if ::= '<process:ifCondition>'
            SWRL-Condition '</process:ifCondition>';
then ::= '<process:then>'
            ControlConstruct '</process:then>';
else ::= '<process:else>'
            ControlConstruct '</process:else>';

        <!-- Repeat-While -->
repeatWhile ::= '<process:Repeat-While
                     rdf:ID=' quotedString '>'
                            whileCondition whileProcess
                '</process:Repeat-While>' ;

whileCondition ::= '<process:whileCondition>'
                       SWRL-Condition '</process:whileCondition>';

whileProcess ::=  '<process:whileProcess
                       rdf:resource=' quotedString '/>';
```

The keyword `SWRL-Condition` is the condition that is written in SWRL, and `ControlConstruct` represents the combination of control constructs. The related syntax of the branching and while commands in our Coq imperative language are defined respectively as follows.

```
'IFB' e1 'THEN' e2 'ELSE' e3 'FI'

'WHILE' b 'DO' c 'END'
```

4.3 Certifying Desirable Properties

In this section, we will present a sample code which is written using previously defined imperative language for an English auction, and then provide the proofs of some incentive properties.

4.3.1 Proof Development

A well-defined auction mechanism can be viewed as a function that maps a set of typed agents into outcomes characterized by utilities usually defined as linear functions. Not only, we need to specify rules and properties but we also need to carry out some calculations. The constructive approach provided by Coq offers possibilities to describe auctions along with desirable properties and prove them. To illustrate how we can specify an auction within Coq, let us consider the English auction example. In the sample code of Table 4.2, `ListBP` is a list of buyers with their own bid variable. `CurrentWinner` and `CurrentBid` are two variables to store the information of buyers and their bids. `NewBuyer` is the buyer and `NewBid` is the bid of this buyer in one round of the auction. Function `AHeadBuyer` is used to find the first buyer in the bidding list,

while function `AHeadBid` is used to find the bid of the first buyer. The third function `ATail` returns the tail of a list. The auction runs from the first bidder in the bidders' list to the last bidder of the list. In each round, a bidder, who does not want to submit a bid is supposed to have a lower bid for the round. This way, we can simulate the English auction.

Table 4.2: The Sample Code of an English Auction

```
(1)  Definition ListBP : id := Id 0.
(2)  Definition CurrentBid : id := Id 1.
(3)  Definition CurrentWinner : id := Id 2.
(4)  Definition NewBuyer : id := Id 3.
(5)  Definition NewBid : id := Id 4.

(6)  Definition sample_englishAuction :=
(7)     WHILE BIsCons (AId ListBP) DO
(8)          NewBuyer ::= AHeadBuyer (AId ListBP);
(9)          NewBid ::= AHeadBid (AId ListBP);
(10)       IFB (BLt (AId CurrentBid )(AId NewBid)) THEN
(11)         CurrentBid ::= (AId NewBid);
(12)         CurrentWinner ::= (AId NewBuyer)
(13)       ELSE
(14)         SKIP
(15)       FI;
(16)       ListBP ::= ATail (AId ListBP)
(17)    END.
```

Using this program, we have proved the following desirable properties: the payment is equal to the highest bid, and the winner has the highest bid. To construct these proofs, we need to define few functions and prove some lemmas. The function `appe` is used to add an element to the end of a list:

```
Fixpoint appe {X:Type} (l:list X) (v:X) : (list X) :=
   match l with
   | nil => [v]
   | cons h t => h :: (appe t v)
   end.
```

We have proved the following lemma `appe_equation`, which states that given a list `x`, an element `h` and another list `y`; the result of the operation `appe x h ++ y` is equal to `x ++ h :: y`, wherein `++` represents the traditional 'append' operation and `::` the usual 'cons' operation.

Lemma 1 (appe_equation)**.** `forall (A:Type) (h:A) (x y : list A),`
`appe x h ++ y = x ++ h :: y.`

Proof. The proof is by induction on list x. Firstly, we show the lemma is true when x is nil, then we show it is true when x is not empty. □

The function `max` is defined to find the maximum bid in a bidding list. Two lemmas are proved for the function `max`. The lemma `max_one` states the following: given the highest bid `cb` in the current list `p` and a new bid whose value comes from a bidderpair

h, if the value of the new bid is greater than cb, then it becomes the highest bid in the resulting list (appe p h).

Lemma 2 (max_one). `forall (cb:nat) (p: list bidderpair) (h:bidderpair), cb = max p, cb < getbid h -> getbid h = max (appe p h).`

Proof. The proof uses the fact that cb is the maximum value in the list of p and h is a new bid which is greater than cb, then h is the maximum value in the new list. □

We also prove the following lemma max_two, which states that given the highest bid cb in the current list p and a new bid from h; if cb is not less than this bid, then cb is the highest bid in the resulting list (appe p h).

Lemma 3 (max_two). `forall (cb:nat) (p: list bidderpair) (h:bidderpair), cb = max p, ~cb < getbid h -> cb = max (appe p h).`

Proof. The proof uses the fact that cb is the maximum value in the list of p and h is a new bid which is less than cb, then cb is the maximum value in the new list. □

We now explain our need of the predicate appear_in, which is defined to state whether a given element appears in a list. Two lemmas on the appear_in were proved. Lemma 4 shows that an element e appears in the resulting list (appe l e). Lemma 5 shows that if an element e appears in list l, then we can deduce that e appears in the resulting list (appe l b).

Lemma 4 (appears_in_snoc1). `forall e l, appear_in e (appe l e).`

Proof. The proof follows from the definition of appear_in. □

Lemma 5 (appears_in_snoc2).
```
forall e b l,
  appear_in e l ->
  appear_in e (appe l b).
```

Proof. The proof follows from the definition of appear_in. □

In the following section, we will provide the proofs of several theorems.

Theorem 4. *In an English auction, the payment equals to the highest bid.*

Theorem 5. *In an English auction, the winner has the highest bid.*

The proofs of these two theorems are carried out using Hoare-style calculus and are shown in Table 4.3.

Table 4.3 illustrates a Coq proof that in the end of the English auction, the value of CurrentBid is the highest bid and CurrentWinner has CurrentBid. By assigning the value of CurrentBid to a payment variable and CurrentWinner to a variable, we can

Table 4.3: Proof Sketch of a Sample of an English Auction

(1)	{{[(0,0)]::ListBP = l ∧ CurrentBid = 0 ∧ CurrentWinner = 0 ∧ NewBid = 0 ∧ NewBuyer = 0}}
(2)	{{exists p, p++X=l ∧ CurrentBid = (max p) ∧ appear_in (CurrentWinner,CurrentBid) p}} WHILE BIsCons (AId ListBP) DO
(3)	{{exists p, p++X=l ∧ CurrentBid = (max p) ∧ appear_in (CurrentWinner,CurrentBid) p ∧ (BIsCons (AId ListBP))}}
(4)	{{exists p, p++X=l ∧ CurrentBid = (max p) ∧ appear_in (CurrentWinner,CurrentBid) p ∧ AHeadBuyer ListBP = AHeadBuyer ListBP ∧ AHeadBid ListBP = AHeadBid ListBP ∧ (BIsCons (AId ListBP))}} NewBuyer ::= AHeadBuyer (AId ListBP);
(5)	{{exists p, p++X=l ∧ CurrentBid = (max p) ∧ appear_in (CurrentWinner,CurrentBid) p ∧ NewBuyer = AHeadBuyer ListBP ∧ AHeadBid ListBP = AHeadBid ListBP ∧ (BIsCons (AId ListBP))}} NewBid ::= AHeadBid (AId ListBP);
(6)	{{exists p, p++X=l ∧ CurrentBid = (max p) ∧ appear_in (CurrentWinner,CurrentBid) p ∧ NewBuyer = AHeadBuyer ListBP ∧ NewBid = AHeadBid ListBP ∧ (BIsCons (AId ListBP))}} IFB (BLt (AId CurrentBid) (AId newBid)) THEN
(7)	{{exists p, p++X=l ∧ CurrentBid = (max p) ∧ appear_in (CurrentWinner,CurrentBid) p ∧ NewBuyer = AHeadBuyer ListBP ∧ NewBid = AHeadBid ListBP ∧ (BIsCons (AId ListBP)) ∧ (BLt (AId CurrentBid) (AId newBid))}}
(8)	{{exists p, p++tail X=l ∧ NewBid = (max p) ∧ appear_in (NewBuyer,NewBid) p}} CurrentBid ::= (AId NewBid)
(9)	{{exists p, p++tail X=l ∧ CurrentBid = (max p) ∧ appear_in (NewBuyer,CurrentBid) p}} CurrentWinner ::= (AId NewBuyer)
(10)	{{exists p, p++tail X=l ∧ CurrentBid = (max p) ∧ appear_in (CurrentWinner,CurrentBid) p}} ELSE
(11)	{{exists p, p++X=l ∧ CurrentBid = (max p) ∧ appear_in (CurrentWinner,CurrentBid) p ∧ NewBuyer = AHeadBuyer ListBP ∧ NewBid = AHeadBid ListBP ∧ (BIsCons (AId ListBP)) ∧ !(BLt (AId CurrentBid) (AId newBid))}}
(12)	{{exists p, p++tail X=l ∧ CurrentBid = (max p) ∧ appear_in (CurrentWinner,CurrentBid) p}} SKIP
(13)	{{exists p, p++tail X=l ∧ CurrentBid = (max p) ∧ appear_in (CurrentWinner,CurrentBid) p}} FI;
(14)	{{exists p, p++tail X=l ∧ CurrentBid = (max p) ∧ appear_in (CurrentWinner,CurrentBid) p}} ListBP ::= ATail (AId ListBP)
(15)	{{exists p, p++X=l ∧ CurrentBid = (max p) ∧ appear_in (CurrentWinner,CurrentBid) p}} END.
(16)	{{exists p, p++X=l ∧ CurrentBid = (max p) ∧ appear_in (CurrentWinner,CurrentBid) p ∧ !(BIsCons (AId ListBP))}}
(17)	{{CurrentBid = (max l) ∧ appear_in (CurrentWinner,CurrentBid) l}}

establish Theorem 4 and Theorem 5. In the following paragraph, we have explained the proof steps in Table 4.3.

The preconditions of this program, which is shown in line (1), are that `ListBP` is a bidding list, and that all the values in `ListBP` are stored to another list `l` by adding a default bidderpair `(0,0)` to the head of `ListBP`. All other variables in the precondition have default value 0. The post conditions of this program are 1) the final value of `CurrentBid` is the highest bid of the original list `l`, and 2) the `bidderpair (CurrentWinner, CurrentBid)` appears in the original list `l` as in line (17). The loop invariant in this proof is: `exists p, p++ListBP=l` $\wedge$ `CurrentBid = max p` $\wedge$ `appear_in (CurrentWinner,CurrentBid) p`, which states that at each iteration of the loop, the original list `l` is equal to the append of the current value of `ListBP` and some other list `p` that keeps track of information from the original state. To satisfy the loop invariant, a function named `appe` is defined to add the head of `ListBP` to the end of `p`. The preconditions of the inner if-sentence in line (6), are the combinations of loop invariant, the while guard and two equalities that are introduced from the two assignments that are in line (8) and (9) of Table 4.2. The proof in the if-sentence is branched as the guard holds in line (7) and not holds in line (11). Line (2) is implied from line (1), while line (17) is implied from line (16). To indicate that the payment is the highest bid, we just need to add an assignment `Payment ::= (AId CurrentBid)` to set the value of `Payment` as `CurrentBid`. By adding another assignment `Winner ::= (AId CurrentWinner)`, we can deduce that the winner has the highest bid. Note that if a property is proven from within Coq, then this property holds in our OWL-S specification since the translation from OWL-S to Coq is sound and complete.

Then, we have implemented the proof of a weakly dominant strategy in an English auction. The definition of weakly dominant strategy is refer to Definition 2 in Section 3.1.2. The utility of the winner in an English auction is defined as $u_i = v_i - p_i$, where p_i is the payment, which is equal to the final bid of i. Other participants' utility is 0. There exists a dominant strategy in the English auction. The proof of this theorem is constructed by different cases.

Theorem 6. *In an English auction, bidding up to its valuation ($b_i \leq v_i$) is a weakly dominant strategy for an agent.*

Proof. Suppose participant i's valuation is v_i and its bidding is b_i. The second highest bid in this auction is *shb*. There are two strategies for each participant: one is bidding up to its valuation, i.e. $b_i \leq v_i$; the other is bidding beyond its valuation, i.e. $b_i > v_i$. There are two cases in this proof:

1. i wins. There are two cases in this condition.

(a) Participant i wins by using either strategy. However, the utility of bid beyond its

valuation ($b_i > v_i$) is $u_i = v_i - b_i$, which is a negative value, while the utility of the other strategy ($b_i <= v_i$) is non-negative value.

(b) Participant i wins by using the strategy of bid beyond its valuation and its valuation is less than the second highest bid, i.e. $b_i > shb > v_i$. The utility of i is negative when it bids beyond its valuation, while the utility of bid up to its valuation is 0.

2. i loses. Participant i gets utility of 0 by using either strategy.

□

In the table of 4.4, we only display the proof sketch of case (a). It shows that if one agent wins, the utility of bid up to its valuation is better than bid beyond its valuation, i.e. `U2 >= U1`. The proof of other cases has been constructed in a similar way. In this program, `V` represents the valuation, `P` is the payment of the winner and `shb` is the second highest bid in an English auction. We start with the outer precondition in (1) and (8). Following `hoare_if` rule and adding the postcondition in (4) and (7), we conjoin the precondition (1) with the guard of the conditional in the `if` construct to obtain (2). Then, we conjoin (1) with the negated guard of the conditional to obtain (5). Substitutions of `U1` and `U2` are used to generate (3) and (6). Finally, we deduce (3) from (2), and deduce (6) from (5).

Table 4.4: Proof Sketch of dominant strategy in an English Auction

```
(1)   {{fun st => True ∧ (st U1) = 0 ∧ (st U2) = 0 ∧ st V > st shb}}
        IFB (BLt (AId V)(AId P)) THEN
(2)   {{fun st => True ∧ (st U1) = 0 ∧ (st U2) = 0 ∧ st V > st shb ∧ st V < st P}}
(3)   {{fun st => st U2 >= st (V-P)}}
              (U1 ::= AMinus (AId V) (AId P))
(4)   {{fun st => st U2 >= st U1}}
              ELSE
(5)   {{fun st => True ∧ (st U1) = 0 ∧ (st U2) = 0 ∧ st V > st shb ∧ !(st V < st P)}}
(6)   {{fun st => st (V-P) >= st U1}}
              (U2 ::= AMinus (AId V) (AId P))
(7)   {{fun st => st U2 >= st U1}}
        FI;
(8)   {{fun st => st U2 >= st U1}}
```

In the proof shown in Table 4.4, we assume that we have already got the winner, payment and second highest bid from an English auction. We can prove more game-theoretic properties from within Coq. In the work [9], we have developed a Coq proof of the well-known statement that bidding its true valuation is the dominant strategy for each agent in a Vickrey auction. An important property that can be checked is that the auction is a well-defined function and that it does implement its specifications through certified code generation [23]. More challenging properties to be checked might be that the auction mechanism is collusion-free or that it is free from fictitious bidding.

4.4 Related Work

There is by now a large body of literature on program transformations and their verification. For example, the researcher may be interested in translating a C code into a Java code or vice versa. More importantly, it may have to be guaranteed that the correctness of all compilation phases and any optimization technique used to improve code performance for safety-critical software. This implies the verification of program transformations from one source into a target in a possibly different programming language, see example [61] and the references therein. Usually, the verification of these kinds of program transformations relies upon the concept of refinement; that is the set of behaviors of the target program is a subset of the source program. In our work, we have translated OWL-S specifications into specifications in a Coq imperative language by using Antlr. Our transformation is merely a one-to-one mapping between two WHILE languages and the semantic equivalence between the source and target specifications can be established using the relational Hoare logic in [14].

In the context of specifications translation, the work reported in [77], OWL-S was mapped into Frame logic for using first order logic based model checking to verify certain properties of Semantic Web Service systems. In our work, we have automated a syntactic translation scheme that preserves the semantics of the source code so that properties that are shown to be true or false from within Coq will stay respectively true or false in the OWL-S paradigm.

The idea of software agents automatically checking desirable properties has been investigated in [100, 101] by using a model checking approach. The computational complexity of such costly verification procedures are investigated in [99]. A typed language which allows for automatic verification that an allocation algorithm is monotonic and therefore truthful was introduced in [60]. More recently, a proof-carrying code approach [79, 2] relying on proofs development tools to enable automatic certification of auction mechanisms have been investigated in [9, 23]. This work aimes at bridging the gap between the need for a machine-readable formalism to specify online auction mechanisms and the ability of software agents to formally verify possibly complex auction properties.

4.5 Summary

In this chapter, we have considered the problem of trust in a network of online auctions by software agents. This requires software agents to understand auction protocols and to prove some desirable properties whose correctness will give confidence in the system. We have first discussed how OWL-S is expressive enough to be used as a machine-readable formalism to describe these mechanisms. We have then shown how such an OWL-S specification can be automatically translated into a Coq imperative

program in order to enable us to develop proofs that are machine-checkable. Machine-checkable proofs are required as we have used the proof-carrying code paradigm wherein an auctioneer will have to provide proofs of claimed properties of its auction and buyer agents will need to check the correctness of those proofs.

The translation from an OWL-S to a Coq imperative language code is carried out using a one-to-one mapping between two WHILE languages and the correctness of the transformation is based on semantics equivalence between the source and the target codes. This semantics equivalence can be justified by using a relational Hoare logic [14] that is sound and complete. This automatic transformation is implemented using Antlr. Finally, we have illustrated the verification of auction properties by developing a Coq proof of the fact that in an English auction, the highest bidder wins the auction. In Chapter 7, we will work on the formalization and prove desirable properties of combinatorial auctions. We cannot express combinatorial auctions in OWL-S since those auctions involve quantification over valuation or bidding functions. Therefore, we will rely on Coq to implement this task.

In the following chapter, we will introduce an inquiry dialogue game that integrated FPCC to enable buyer agents to automatically communication with an auctioneer. Buyer agents can ask questions which are properties that have been proved in this chapter to the auctioneer, and then make their decisions whether or not to participate in an auction according to the result of the dialogue.

Chapter 5

A Dialogue Game Approach to Enable Decision Making

5.1 Introduction

We consider e-commerce scenarios wherein software agents can buy or sell goods on behalf of their owners. To enable software agents to participate in such online trading, the trading mechanism should be presented in a machine understandable way. The Semantic Web provides an approach to enable agents to read and interpret a trading mechanism as an online service for which static and dynamic information can be explicitly described using ontology languages. For example, the Web Ontology Language (OWL) [75], which is based on description logic, can be used to express classes and relationships among them. The Semantic Markup for Web Services (OWL-S) [69], which is focused on the process description of a service, can be used to describe the procedures of the trading mechanism. However, the message exchange in the architecture of Semantic Web Services is restricted as a client-server or request-response pattern. To extend the interactivity of the Semantic Web Services, dialogue games are introduced to support message exchanges. By using dialogue games, agents can assert, challenge and justify their arguments according to their knowledge [109].

Dialogue games are rule-governed interactions among software agents [73], wherein each agent presents its ideas by making "moves" based on a set of rules. Since the common agent communication languages, such as FIPA ACL [38], lack certain locutions to express justifications for statements, additional locutions are proposed to extend the FIPA ACL so that argumentation can be supported in a dialogue [72]. Dialogue games permit agents to carry out various types of interactions, such as information seeking, inquiry, persuasion or negotiation. Agents can construct a dialogue by dynamically adjusting the content and a sequence of utterances as the discussion ensues. In our framework, we have used an inquiry dialogue to make two agents take turns in asserting, questioning, accepting, or rejecting statements. The goal of an inquiry dialogue is to find out whether a statement is true or false or show that there is insufficient

evidence to accept a statement [112]. In our work, evidence can be formal proof or an informal statement (e.g., statistical evidence) for a desirable property of the trading mechanism. An example of a desirable property of an auction mechanism could be that the highest buyer wins or that bidding its true valuation is the optimal strategy for a buyer. Formal proofs are constructed using the Coq [34] theorem prover within the PCC (Proof-Carrying Code) paradigm [80]. In our online auction scenario, PCC enables the auctioneer to develop proofs for properties of interest and the buyer to check the correctness of a given proof [9]. By considering the set of evidences collected in a dialogue for a set of desirable properties, a buyer agent as a service consumer can evaluate the quality of a service and make decision as to whether to enter or leave a service.

On the one hand, in the language architecture of Semantic Web [49], each language in the above layer extends the capabilities of the layer below, as shown in Figure 2.1. For example, OWL extends RDFS by providing more semantic features like cardinality, union, intersection and more reasoning possibilities. The top two layers in this semantic web stack are the Proof and Trust layers. In essence, proofs can be constructed so as to increase trustworthiness in the system. However, the current system does not support yet a proof construction or procedure for building up trust. On the other hand, Interactive Theorem Proving provides an approach to develop formal proof by man-machine collaboration. It is an important approach which using formal methods to verify the correctness of a protocol or a mathematical statement. In an Interactive Theorem Prover (ITP), human can create definitions, theorems and generate proofs using an interactive proof editor. The ITP also supports automatic checking of proofs. In our work, we use Coq [34], which is an ITP, to generate and check the proofs of desirable properties of an online auction.

The contributions of this chapter are three-fold:

- We have integrated dialogue games within the PCC paradigm so as to increase trust in an agents-mediated online auction. As a consequence, we have extended the interactivity of agent communication wherein formal proofs can be used as arguments in a dialogue.

- Since, not all desirable properties of an online auction mechanism will have associated formal proofs, we have allowed for informal or empirical evidence related to the QoS (Quality of Service). For example, an auctioneer may claim that it does not have a proof that its mechanism is free from cheating, but 98% of its consumers never complained about being cheated. We have designed a dialogue game wherein formal and informal evidence can be used as arguments.

- We have constructed a decision model over the formal and empirical evidences allowing a buyer agent, with predefined expectations, to decide whether to join

or not an auction.

The proposed dialogue game framework is implemented in JADE [12], which is a widely used tool to implement multi-agent systems. It provides mechanisms to create agents, enable agents to execute tasks and make agents communicate with each other.

The remainder of this chapter is organized as follows: In section 5.2, we will introduce the technique of Semantic Web Service and give an example of an English auction which is written in the language of OWL-S. The proposed dialogue game framework is presented in Section 5.3. In Section 5.4, we present an example that implements our dialogue game framework, and is then evaluated in Section 5.5. Related work is presented in Section 5.6, before summary in Section 5.7.

5.2 Ontologies for Services

The Semantic Web [15] not only enables greater access to content but also to services on the Web. Semantic Web Service is a technology that combines Semantic Web and Web services to develop new Web applications. Web services technology is based on a set of standard protocols such as UDDI (Universal Description, Discovery and Integration), SOAP (Simple Object Access Protocol), and WSDL (Web Services Description Language). However, these standards do not support automated Web services. To improve the automatic properties of Web services, Semantic Web Services (SWS) are created [26]. OWL-S [69] and WSMO [32] are two standards for the SWS technology. OWL-S is composed of three main parts: the service profile for advertising and discovering services; the process model, which gives a detailed description of a service operation; and the grounding, which provides details on how to interoperate with a service via messages. WSMO (Web Service Modeling Ontology) provides a conceptual framework for semantically describing all relevant aspects of Web services in order to facilitate the automation of discovering, combining and invoking electronic services over the Web. WSMO comprises four main elements: ontologies, which provide the terminology used by other WSMO elements; Web service descriptions, which describe the functional and behavioral aspects of a Web service; goals that represent the user's desires; and mediators, which aim at automatically handling interoperability problems between different WSMO elements. Current SWS technique can help us build a system that enables agents to publish, discover and invoke services in an open environment (the Internet).

5.2.1 A Scenario: An English Auction

In our work, we use OWL-S to build up the Web service. OWL-S can be used together with other Semantic Web languages, such as OWL DL [75] and SWRL [50], to describe the properties and capabilities of a Web service in unambiguous, computer-interpretable

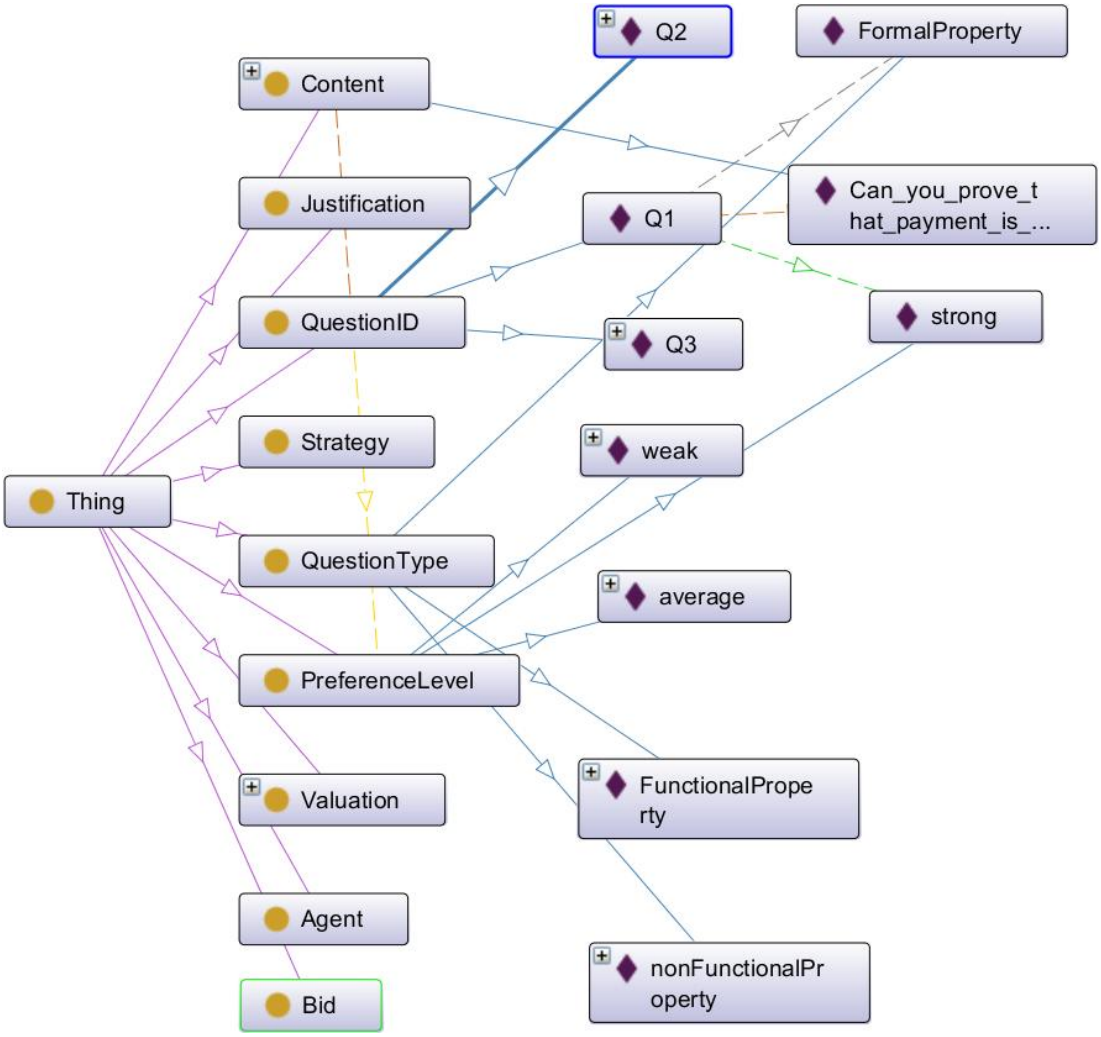


Figure 5.1: A Fragment of the Auction Ontology

form. To set up a Semantic Web Service, the first step is to build the ontology of the specific area. The ontology can be used to formally describe the semantics of terms representing an area of knowledge and give explicit meaning to the information. This enables automated reasoning, semantic search and knowledge management of the specific area. For example, the ontology of auction domain can contain the constructs of classes, relations, axioms, individuals and assertions. We give a fragment of the auction ontology in Figure 5.1. The auction domain ontology not only defines basic information of an agent (e.g., bid, valuation, strategy); but also the preference of an agent. These preferences can be used to help agents make decisions during a trading dialogue. The following code illustrates the ontology about a question named Q1. Q1 has three object properties which are *hasContent, hasPreference, hasType* and one data property with name *hasWeight*.

```
<ClassAssertion>
    <Class IRI="#QuestionID"/>
    <NamedIndividual IRI="#Q1"/>
</ClassAssertion>
<ObjectPropertyAssertion>
    <ObjectProperty IRI="#hasContent"/>
    <NamedIndividual IRI="#Q1"/>
    <NamedIndividual IRI="#Can_you_prove_that_payment_is_the_highest_bid"/>
</ObjectPropertyAssertion>
<ObjectPropertyAssertion>
    <ObjectProperty IRI="#hasPreference"/>
    <NamedIndividual IRI="#Q1"/>
    <NamedIndividual IRI="#strong"/>
</ObjectPropertyAssertion>
<ObjectPropertyAssertion>
    <ObjectProperty IRI="#hasType"/>
    <NamedIndividual IRI="#Q1"/>
    <NamedIndividual IRI="#FormalProperty"/>
</ObjectPropertyAssertion>
<DataPropertyAssertion>
    <DataProperty IRI="#hasWeight"/>
    <NamedIndividual IRI="#Q1"/>
    <Literal datatypeIRI="&xsd;double">0.5</Literal>
</DataPropertyAssertion>
```

After the construction of domain ontology, the auction mechanism should be described using the OWL-S ontology. In such a trading mechanism, functional description should be defined: inputs, outputs, preconditions and results (IOPEs). The *inputs* are the objects that should be provided to invoke the service; the *outputs* are the objects that the service produces; the *preconditions* are the propositions that should be true prior to service invocation; and the *results* consist of effects and outputs. The following code is an example showing that, the precondition for running an auction is that at least one agent must have some bid.

```
<process:hasPrecondition>
    <expr:SWRL-Condition>
        <expr:expressionObject>
        <swrl:AtomList>
        <rdf:first>
            <swrl:DatavaluedPropertyAtom>
```

```
                <swrl:propertyPredicate   rdf:resource="&mybid;#hasBid"/>
                <swrl:argument1 rdf:resource="&this;#agent"/>
                <swrl:argument2 rdf:resource="&this;#abid"/>
            </swrl:DatavaluedPropertyAtom>
        </rdf:first>
        <rdf:rest rdf:resource="&rdf;#nil"/>
        </swrl:AtomList>
        </expr:expressionObject>
    </expr:SWRL-Condition>
</process:hasPrecondition>
```

In OWL-S, a process represents a specification of the means a buyer uses to interact with a service. In our scenario, we use two kinds of processes: one is *atomic process*, which corresponds to a single interchange of a request message and a response message; the other is *composite process*, which consists of a series of processes linked together by control flows and data flows. The control flow describes the relations between the executions of different sub-processes. The control constructs include *sequence, split, if-then-else* and *iterate* etc. Data flow specifies how information is transferred from one process to another process. In our example of an English auction, we can define one *if-then-else* branch to describe that when a new bid is greater than current bid (a local variable), we update the value of current bid with the new bid. This process can be simply described as follows.

```
<process:CompositeProcess>
 <process:composedOf>
<process:If-Then-Else rdf:ID="CompareBid">
<process:ifCondition>
<expr:SWRL-Condition>
swrlb:lessThan(#currentBid,#newBid)
</expr:SWRL-Condition>
</process:ifCondition>
<process:then>
 <!-- Update the value of #currentBid -->
        ...
</process:then>
<process:else>
<!-- Keep the value of #currentBid as usual-->
        ...
</process:else>
   </process:If-Then-Else>
 </process:composedOf>
</process:CompositeProcess>
```

In our setting, we consider the scenario that buyers communicate with the seller to decide whether or not to join an online auction. Therefore, we do not need to define the grounding of the service. OWL-S can be used to build complex business solutions by describing the functional, non-functional properties of a service, so that agents can perform automatic reasoning on these descriptions. Dialogue games, which can help software agents interact rationally by providing support or counterexample for a conclusion, make this reasoning more flexible for greater interaction between an auctioneer and a buyer.

5.3 Our Dialogue Game Framework

Online auction web sites, such as eBay, have attracted millions of users around the world to sell, bid and buy goods. In our specific scenario, software agents are assumed to be capable of buying or selling goods through online auction houses. In this scenario, how can buyer agents choose the appropriate auction house? A buyer agent will have properties of interest. These properties need to be kept in the auction mechanism for the agent to join, bid, and buy items. Our framework is aimed at enabling a potential buyer (e.g., software agent) to interact with the auctioneer before deciding whether or not to join the auction. These interactions between the auctioneer and a buyer are carried out within a dialogue game wherein the buyer agent can query whether desirable properties hold and request associated evidences. To enable trust, the auctioneer uses the PCC (Proof-Carrying code) paradigm to convince buyers that a service has some desirable properties, as shown in our previous work [9, 8]. This enables the auctioneer to specify and to develop formal proofs for those properties. Then, the buyer uses a proof checker to check that a given proof of a well specified property of the auction is correct. Besides, a buyer can object to the auctioneer under the condition that a counterexample is found by the proof checker. Thus, trust can be established between an auctioneer and a buyer. In our PCC implementation [9], we have used the theorem prover Coq [34] to develop the proofs of desirable properties. This is achieved by translating OWL-S descriptions into Coq specifications [7], and then uses Coq's machinery to specify and prove a property of interest. On the buyer side, we have used the Coq proof checker so that a buyer can automatically check that a claimed property by the auctioneer holds in the system. Figure 3.1 illustrates our PCC implementation in this setting.

We have integrated the PCC paradigm within our dialogue model as illustrated by the framework in Figure 5.2. In this framework, both the auctioneer and the buyer share the same question ontology for general online auction services. The auctioneer holds an OWL-S ontology which provides a machine understandable description of the auction mechanism. Evidence for a claimed property may have an informal or formal justification. Informal justification relies upon data collected by the service. In the case of properties with formal proofs, the related specification of the OWL-S description can be translated into Coq specifications so that proofs of these properties can be developed from within Coq. This kind of sound language translation is described in our previous work [7]. The proof certificates for the established desirable properties are local to the auctioneer. This strengthens the interactivity between a buyer and an auctioneer and reduces knowledge disclosing. The proofs will be disclosed to a buyer when related questions are proposed in a dialogue game. The dialogue game is used to enable a buyer agent to find out about properties and certificates. This dialogue is a two-person game, which means that only one buyer can communicate with the auctioneer at a time.

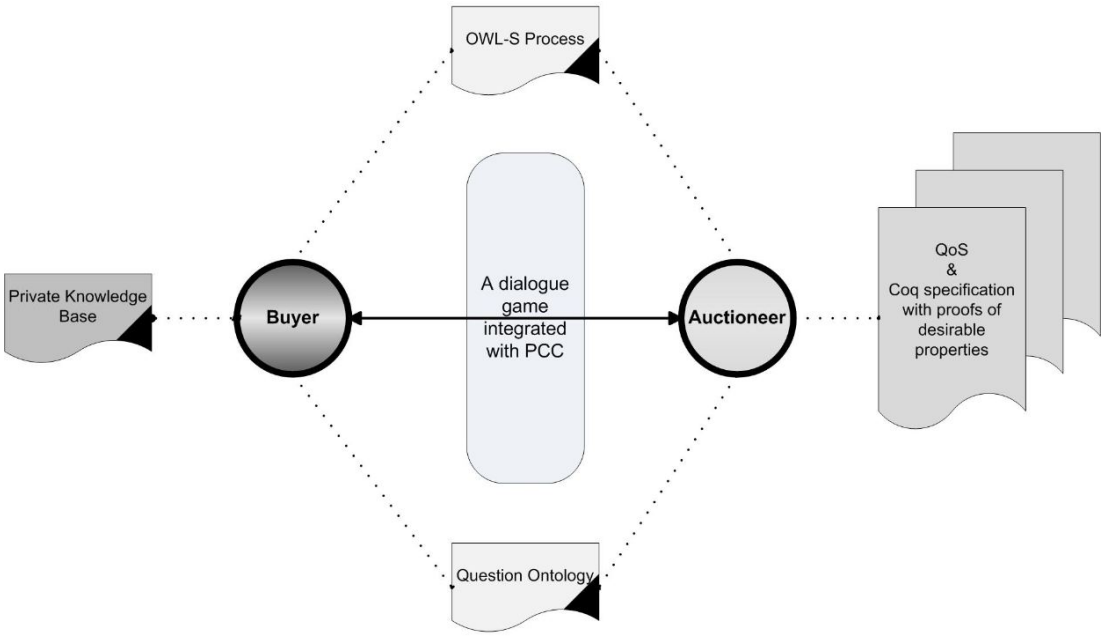


Figure 5.2: The dialectical approach framework. A buyer agent uses a dialogue to communicate with the auctioneer and the checking of proofs of desirable properties can be integrated within the dialogue using PCC paradigm.

5.3.1 The Formal Dialogue Model

The proposed inquiry dialogue consists of a number of *locutions* or *moves*, *pre-conditions* that indicate the rules that must be satisfied before a move, and the *post-conditions* that describe the actions that will occur after a move. We have restricted the number of participants in the inquiry dialogue to two. Let P be the participants in the dialogue. A participant is either a sender or a recipient.

A dialogue D is simply defined by a sequence of moves between the participants. One move represents a message exchange made from one participant to the other. As the dialogue progresses, each move is indexed by a timepoint, which is denoted by a natural number, and only one move can be made at each timepoint. In our inquiry dialogue model, seven types of moves are defined. They are *open, assert, question, justify, accept, reject* and *close*, and the type of each legal move should be one of them.

Definition 4. *A **dialogue**, denoted D, is a sequence of moves $[m_r, ..., m_t]$, where $r, t \in \mathrm{N}, r < t$, involving two participants $P_i \in P, i = \{1, 2\}$, such that:*

1. *the first move of the dialogue, m_r is of type open,*
2. *the last move of the dialogue, m_t is of type close,*
3. $Sender(m_s) \in P(r \leq s \leq t)$
4. $Sender(m_s) \neq Sender(m_{s+1})(r \leq s < t)$

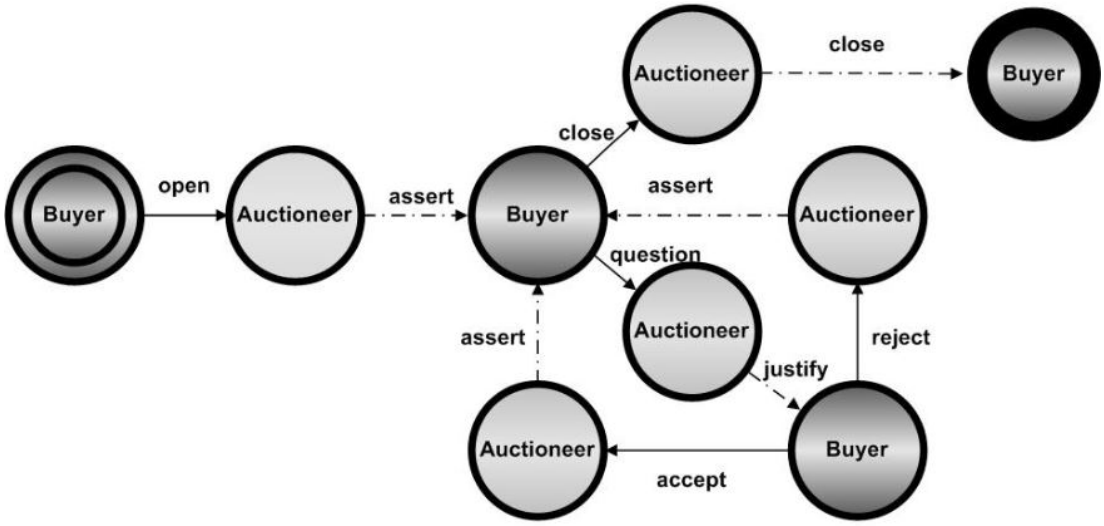


Figure 5.3: The state diagram of the dialogue. Nodes indicate the agent whose turn is to utter a move. Moves uttered by the auctioneer are labeled with a dashed line, while those uttered by the buyer are labelled with a solid line.

The first move of a dialogue D must always be an *open* move (condition 1), while the last move should be a *close* move (condition 2). Each move of the dialogue must be performed by a participant of the dialogue (condition 3). Finally, participants take turns to make moves (condition 4).

The dialogue assumes that each participant holds a commitment store that records its statements in a dialogue. The commitment store of participant P_i is defined as a private-write, public-read record containing all the commitments incurred by P_i. Both of the participants can read the commitment store of P_i, but the content of this commitment store can only be written using the moves made by P_i.

Definition 5. *A **commitment store** is a set of beliefs denoted as CS_x^t, where $x \in P$ is an agent and $t \in \mathrm{N}$ is a timepoint.*

The commitment store of P_i is created when the agent enters into a dialogue and persists until the dialogue terminates.

Definition 6. *A **proof** is an argument from hypotheses to a conclusion and each step of the argument follows the laws of logic.*

The state diagram of the dialogue is given in Figure 5.3. A buyer can open a dialogue by using the *open* move. Then, the auctioneer can give an assertion to the buyer. The buyer can choose to close the dialogue on the condition that he does not have any issues to raise with the auctioneer. Otherwise, the buyer can give a move of type *question* to query on the issues that he is concerned about. The auctioneer must give a justification to the buyer for each specific question. There are three cases in the justification process: 1) the auctioneer has a formal proof for the answer to

the question at hand; 2) the auctioneer has an informal evidence for the answer; or 3) the auctioneer does not know the answer for the question. In the first case, the buyer will use the PCC paradigm to check the correctness of the formal proof, if this proof is certified, then the buyer gives an *accept* move, otherwise, the proof checker will generate a counterexample and deny the property that related to the question, then the buyer make the move of *reject*. In the second case, the buyer will compare the informal evidence with a reasonable expected value for the issue of interest and will accept it with some score. In the last case, the buyer will give a *reject* move to the auctioneer. The auctioneer can give assertions to the buyer, so that the dialogue can carry on until both participants agreed to close the dialogue.

The components of a dialogue include:

- $\sum$: The knowledge base, or beliefs of each agent.
- $CS \in \sum$: an agent's commitment store that refers to the statements that have been made in the dialogue.
- a: auctioneer.
- b: buyer.
- x: either auctioneer or buyer.
- ρ: the last sender in the current dialogue.

The locutions used in the dialogue model are defined as follows.

Definition 7. *open*(b)*: buyer* b *opens a dialogue.*

- *Pre-conditions:*
  1. $b \notin P$, *where* P *is the set of dialogue participants.*
- *Post-conditions:*
  1. $P' = P \cup \{b\}$
  2. $CS^b = \emptyset$
  3. $\rho = b$

A buyer b can open a dialogue. The precondition of this locution is that the buyer is not a participant of this dialogue. The postcondition is that buyer b becomes a participant of the dialogue (post-condition 1) and his commitment store is created (post-condition 2).

Definition 8. *assert*(a, b, ϕ)*: auctioneer* a *gives an assertion to the buyer.*

- *Pre-conditions:*

1. $a \neq \rho$
2. $locutiontype(l_{s-1}) \in \{open, accept, reject\}$
3. $\phi \in \Sigma^a$

- *Post-conditions:*

1. $CS^{a'} = \{\phi\} \cup CS^a$
2. $\rho = a$

The *auctioneer* should not have uttered the previous move (pre-condition 1). The *assert* move should be given under the conditions that a buyer has open a dialogue, the justification is accepted or rejected by the buyer (pre-condition 2). A belief ϕ from the beliefs store of the auctioneer will be asserted by the auctioneer to ask the buyer to propose a question (pre-condition 3). Once the assert has been uttered, the belief ϕ will be added to the commitment store of the auctioneer (post-condition 1).

Definition 9. *$question(b, a, \phi)$: buyer b asks a question about property ϕ to the auctioneer a.*

- *Pre-conditions:*

1. $b \neq \rho$
2. $locutiontype(l_{s-1}) \in \{assert\}$
3. $\phi \notin \Sigma^b$
4. $\phi \notin CS^b$

- *Post-conditions:*

1. $\phi \notin \Sigma^b$
2. $\phi \in CS^b$
3. $\rho = b$

A buyer can ask questions to the auctioneer after the auctioneer has uttered an assertion (pre-conditions 1 and 2). The property ϕ does not contain the knowledge base of b (pre-condition 3) and the buyer has not proposed questions about this property (pre-condition 4). Once b has uttered the question, the knowledge base of b does not change (post-condition 1), and the commitment store of b has been updated (post-condition 2).

Definition 10. *$justify(a, b, \phi)$: auctioneer a justifies the property ϕ for buyer b.*

- *Pre-conditions:*

1. $a \neq \rho$

2. $locutiontype(l_{s-1}) \in \{question\}$
3. $\phi \in CS^b$
4. $\phi \notin \Sigma^b$

- *Post-conditions:*

1. $\phi \notin \Sigma^b$
2. $\phi \in CS^a$
3. $\rho = a$

After receiving a question about property ϕ (pre-condition 2 & 3), the auctioneer a should provide a justification to this property. The knowledge base of the buyer and commitment store remains the same in this move (post-condition 1). The justification of this property is added to the commitment store of the auctioneer (post-condition 2).

Definition 11. *$accept(b, a, \phi)$: buyer b accepts the justification of property ϕ .*

- *Pre-conditions:*

1. $b \neq \rho$
2. $locutiontype(l_{s-1}) \in \{justify\}$
3. $\phi \notin \Sigma^b$
4. $\phi \in CS^a$

- *Post-conditions:*

1. $\phi \in \Sigma^b$
2. $\phi \in CS^a$
3. $\rho = b$

The *accept* move is used to accept the justification for property ϕ in the preceding *justify* move (pre-condition 2). The property ϕ is not contained in the knowledge base of the buyer before the move of *accept* (pre-condition 3), and the auctioneer has provided the justification for this property (pre-condition 4). Property ϕ becomes the element of the knowledge base of b after this move (post-condition 1). The commitment store of a does not change in the move of *accept* (post-condition 2).

Definition 12. *$reject(b, a, \phi^{att}, \phi)$: buyer b rejects the property ϕ using ϕ^{att} .*

- *Pre-conditions:*

1. $b \neq \rho$
2. $locutiontype(l_{s-1}) \in \{justify\}$

3. $attack(\phi^{att}, \phi)$
4. $\phi \notin \Sigma^b$
5. $\phi \in CS^a$

- *Post-conditions:*

1. $\phi \notin \Sigma^b$
2. $\phi \in CS^a$
3. $\rho = b$

Buyer can raise an rejection by giving an evidence ϕ^{att} to previously declared property ϕ in response to the move *justify* (pre-condition 2). There is an evidence ϕ^{att} which attacks ϕ (pre-condition 3). The knowledge base of b and commitment store of a does not change (pre-condition 4 & 5 and post-condition 1 & 2).

Definition 13. *close(p): participant p closes a dialogue.*

- *Pre-conditions:*

1. $p \neq \rho$
2. $locutiontype(l_{s-1}) \in \{assert, close\}$
3. $\forall \phi \in \Sigma^b, \phi \in CS^b$

- *Post-conditions:*

1. *if matched-close* $P = \emptyset$

The participant can only close a dialogue after an *assert* or *close* (pre-condition 2). The buyer chooses to close the dialogue when all the questions in his knowledge base have been proposed (pre-condition 3). When both participants have agreed to close the dialogue, they will be removed from the dialogue (post-condition 1).

5.3.2 Decision Model and Processes of the Dialogue

In our setting, both the auctioneer and the buyer agents share the same knowledge base, which includes the ontologies of the online auction and related specifications. Both agents can understand each other's messages but have a private knowledge base. The set of questions are private to the buyer and the set of answers associated to the questions are private to the auctioneer. However, when a question or answer is proposed, it becomes public to both participants. Each question is associated with a *preference level*, which determines the order in which the questions are proposed by the buyer. The preference level $E = \{strong, average, weak\}$ is then used to drive the dialogue forward.

A *Weighted Sum* model is used in the dialogue game. The output of the *Weighted Sum* model is a binary set of O, whose elements are $\{Yes, No\}$. Decision is made by evaluating a bunch of properties, which are represented as the set of H. We assume that every property is bind with a score s_i and a relative weight w_i, which indicates the importance of a property. The value of s_i is determined by a function which depends on the type of evidence provided. If we have formal evidence, then the scoring function will return either negative value in the case of the proof cannot be accepted by the proof-checker or a full score otherwise. If the evidence is empirical, then the scoring function is in the form of intervals, see Section 5.4 for more details. It is the usual practice to set the summation of weights to 1 in the weighted sum model, such that $\sum_{i=1}^{n} w_i = 1$ wherein n is the number of elements in H [33]. The final score fs is calculated as follows:

$$fs = \sum_{i=1}^{n} w_i s_i$$

, where n is also the number of elements in H.

The buyer has a reasonable expectation in the form of an *admissibility threshold* ε beyond which the final score will lead the buyer to join the auction. In other words, if $fs \geq \varepsilon$, then the buyer will choose Yes to join the auction at hand. The threshold ε may come from experience or be derived from historical data.

Algorithm 1 The processes of an inquiry dialogue

```
1: buyer opens an inquiry dialogue uses an open move
2: auctioneer gives an assert to ask buyer to propose a question
3: while buyer has question(s) that has(have) not been proposed do
4:    buyer selects a question which with the highest preference level
5:    buyer asks this question using the move whose type is question
6:    auctioneer gives a justification using the move whose type is of justify
7:    if the justification is unknown or has failed to be checked then
8:       buyer gives a reject move
9:    else
10:      buyer gives an accept move
11:   end if
12:   auctioneer gives an assert to ask buyer to propose a question
13: end while
14: buyer gives a close move
15: auctioneer gives a close move to terminate the dialogue
```

The inquiry dialogue is described by Algorithm 1. A dialogue starts with a move opening an inquiry dialogue, that has a matched-close to terminate the dialogue and whose moves conform to the rules for each locution described in Section 5.3.1. The *buyer* starts an inquiry dialogue by giving an *open* move, then the auctioneer propose an *assert* move to ask *buyer* to propose questions. The *buyer* chooses an unproposed question which with the highest preference level from his knowledge base and then uses

the move of *question* to propose it. After receiving a question, the auctioneer should respond by a *justify* move. If the content of a justification is unknown, or a formal proof that is failed to be checked. The *buyer* will give a *reject* move, otherwise he should accepts the justification use an *accept* move. The *auctioneer* should give an assertion to ask *buyer* to propose a new question in the end of the loop. A dialogue can be closed when both of the participants agree to terminate it. All of the proposed questions and related justifications are stored in the commitment store of the dialogue. Besides, each question can only be asked once.

5.4 Inquiry Dialogue Example

We have illustrated the proposed inquiry dialogue by means of an example where a *buyer* talks to an *auctioneer* to decide whether or not to join an online auction service.

Table 5.1: Questions in the shared ontology

QuestionID	hasType	hasContent
Q1	FormalProperty	Can you prove that the payment is the highest bid?
Q2	FunctionalProperty	What's the payment method?
Q3	FunctionalProperty	What's the delivery company for your product?
Q4	FunctionalProperty	What's the Input of the service?
Q5	nonFunctionalProperty	What's the Reputation of your service?
Q6	nonFunctionalProperty	What's the ResponseTime of your service?
Q7	FormalProperty	Can you prove that the winner has the highest bid?

The shared question ontology contains three types of questions: 1) questions about those properties whose answers could be formal proofs of a service; 2) questions about the functional properties (e.g. inner operation of a service) of a service; and 3) questions about the non-functional properties (e.g. QoS) of a service. In this example, seven questions are proposed as listed in Table 5.1.

In Table 5.1, each question is marked by an ID, and the related type and content of each question are defined use the OWL *objectProperty* relationship. Users can extend this ontology by adding questions as the classification of types.

The *buyer* holds a private knowledge base that contains the questions he wants to inquire the dialogue. The preference level and weight of each question are shown in Table 5.2. As mentioned above the *buyer* will choose the questions in order on the basis of the preference level and calculate the scores of the justifications using the variable of weight, w. The summation of all the weights in this table equals to one.

The *auctioneer* uses Table 5.3 to search for the answers for each query, a *buyer* can not directly visit this table unless he queries the *auctioneer*. All the questions in the shared ontology should be contained in this table, though there may exist questions

Table 5.2: Questions in the knowledge base of the buyer

QuestionID	Preference Level	Weight (w)	hasContent
Q1	strong	0.3	Can you prove that the payment is the highest bid?
Q2	average	0.15	What's the payment method?
Q3	weak	0.1	What's the delivery company for your product?
Q5	strong	0.2	What's the Reputation of your service?
Q6	average	0.15	What's the ResponseTime of your service?
Q7	weak	0.1	Can you prove that the winner has the highest bid?

Table 5.3: Justifications of questions hold by the auctioneer

QuestionID	hasContent	Justification
Q1	Can you prove that the payment is the highest bid?	FormalProof_PEquHB
Q2	What's the payment method?	Visa DEBIT
Q3	What's the delivery company for your product?	DHL
Q4	What's the Input of the service?	BuyerID & Bid
Q5	What's the Reputation of your service?	0.85
Q6	What's the ResponseTime of your service?	*unknown*
Q7	Can you prove that the winner has the highest bid?	FormalProof_WhasHB

that do not have related answers. Table 5.4 shows the grading standards of the *buyer* for the justifications. The highest score for each question is 100, the lowest score for the formal question is -50 when the proof provided by the auctioneer is failed to check, and the lowest score other questions is 0. As this grading table is subjective, a buyer can grade the justification based on his own preference. However, for the question whose justification is a formal proof, the buyer should give full marks to it when the proof is successfully checked.

An example dialogue between *buyer* and *auctioneer* is presented as in Figure 5.4. The turn order in the above dialogue is deterministic, the buyer should open a dialogue in **Move 1**. Then the *auctioneer* and *buyer* give assertions one by one.

Moves 2-5: The *auctioneer* asks *buyer* to ask a question. The *buyer* searches in his knowledge base of questions and find out a question that with the highest preference, which is *Q1*. Then the auctioneer searches the justification for the question from Table 5.3. This justification is a formal proof, so the *auctioneer* should send the address of a proof checker (in the case that the *buyer* does not have the proof checker) and related proof files to the *buyer*. The *buyer* then downloads and uses the proof checker to check the correctness of the proof, and gets positive feedback. Finally, the *buyer* accepts the justification.

Moves 6-9: The *auctioneer* asks *buyer* to ask another question. The *buyer* finds question *Q5*, which has the highest preference level within the remaining questions.

Table 5.4: Grading Table of the buyer

QuestionID	Justification	Score
Q1	Formal Proof Successes	100
	Formal Proof Fails	-50
	Do Not Have Formal Proof	0
Q2	Alipay	100
	Visa DEBIT	80
	Bank Cards	50
	Others	0
Q3	SF EXPRESS	100
	DHL	80
	EMS	70
	Others	0
Q5	[0.90,1.00]	100
	[0.70,0.90)	80
	[0.50,0.70)	50
	[0.00,0.50)	0
	unknown	0
Q6	(0.00ns,0.03ns)	100
	[0.03ns,0.05ns)	80
	[0.05ns,0.10ns)	50
	[0.05ns,$+\infty$)	0
	unknown	0
Q7	Formal Proof Successes	100
	Formal Proof Fails	-50
	Do Not Have Formal Proof	0

(1) *buyer* → *auctioneer*: *open(buyer)*
(2) *auctioneer* → *buyer*: *assert(auctioneer, buyer, You can ask a question.)*
(3) *buyer* → *auctioneer*: *question(buyer,auctioneer,Q1)*
(4) *auctioneer* → *buyer*: *justify(auctioneer,buyer,FormalProof_PEquHB)*
(In the justification process, the *auctioneer* sends the proof to the *buyer* and the proof is successfully checked by using the COQ proof checker.)
(5) *buyer* → *auctioneer*: *accept(buyer,auctioneer,FormalProof_PEquHB)*
(6) *auctioneer* → *buyer*: *assert(auctioneer, buyer, You can ask a question.)*
(7) *buyer* → *auctioneer*: *question(buyer,auctioneer,Q5)*
(8) *auctioneer* → *buyer*: *justify(auctioneer,buyer,0.85)*
(9) *buyer* → *auctioneer*: *accept(buyer,auctioneer,0.85)*
(10) *auctioneer* → *buyer*: *assert(auctioneer, buyer, You can ask a question.)*
(11) *buyer* → *auctioneer*: *question(buyer,auctioneer,Q2)*
(12) *auctioneer* → *buyer*: *justify(auctioneer,buyer,Visa DEBIT)*
(13) *buyer* → *auctioneer*: *accept(buyer,auctioneer, Visa DEBIT)*
(14) *auctioneer* → *buyer*: *assert(auctioneer, buyer, You can ask a question.)*
(15) *buyer* → *auctioneer*: *question(buyer,auctioneer,Q6)*
(16) *auctioneer* → *buyer*: *justify(auctioneer,buyer, unknown)*
(17) *buyer* → *auctioneer*: *reject(buyer,auctioneer, unknown)*
(18) *auctioneer* → *buyer*: *assert(auctioneer, buyer, You can ask a question.)*
(19) *buyer* → *auctioneer*: *question(buyer,auctioneer,Q3)*
(20) *auctioneer* → *buyer*: *justify(auctioneer,buyer, DHL)*
(21) *buyer* → *auctioneer*: *accept(buyer,auctioneer,DHL)*
(22) *auctioneer* → *buyer*: *assert(auctioneer, buyer, You can ask a question.)*
(23) *buyer* → *auctioneer*: *question(buyer,auctioneer,Q7)*
(24) *auctioneer* → *buyer*: *justify(auctioneer,buyer,FormalProof_WhasHB)*
(In the justification process, the *auctioneer* sends the proof to the buyer. The *buyer* failed to checked the proof by using the COQ proof checker, which means the proof provided by the *auctioneer* is wrong. The *buyer* finds an attack to this property.)
(25) *buyer* → *auctioneer*: *reject(buyer,auctioneer,FormalProof_WhasHB)*
(26) *auctioneer* → *buyer*: *assert(auctioneer, buyer, You can ask a question.)*
(27) *buyer* → *auctioneer*: *close(buyer)*
(28) *auctioneer* → *buyer*: *close(auctioneer)*

Figure 5.4: An example of the inquiry dialogue

The *auctioneer* gives the justification with value *0.85* and *buyer* accepts it.

Moves 10-13: After the *auctioneer* requests the *buyer* to ask a question. The *buyer* proposes question **Q2** and receives the justification with value of *Visa DEBIT*. The *buyer* accepts the justification in this round.

Moves 14-17: In this round, the *buyer* raises question *Q6*, the *auctioneer* does not have a justification for this question, so the content of this justification becomes *unknown*. Then the *buyer* rejects this justification.

Moves 18-21: The *buyer* proposes another question in his knowledge base and accepts the justification that is given by the *auctioneer*.

Moves 22-25: The *auctioneer* asks *buyer* to ask a question. The *buyer* proposes the last question from his knowledge base, which is *Q7*. Then the auctioneer sends the proof to the *buyer*. The *buyer* uses the proof checker to check the correctness of the proof and gets a counterexample, which means the proof provided by the auctioneer is wrong. In this case, the *buyer* finds an attack of this property. Finally, the *buyer* rejects the justification.

Moves 26-28: The *auctioneer* asks the *buyer* to propose a new question. However, all the questions in the *buyer*'s knowledge base have been raised. The *buyer* chooses to close the dialogue and the *auctioneer* agrees to close the dialogue.

After the termination of the dialogue, the *buyer* calculates the scores for each justification. The final score *fs* is calculated as: $fs = 0.3 * 100 + 0.15 * 80 + 0.1 * 80 + 0.2 * 80 + 0.15 * 0 + 0.1 * (-50) = 61$. We assume that the *admissibility threshold*, ε, of the *buyer* is 70. As $fs < \varepsilon$, the *buyer* decides to not join the auction house.

5.5 Empirical Evaluation

We have implemented the proposed dialogue model in the JADE platform. We have added the proposed locutions to the original FIPA ACL messages, so that the original methods (e.g. *addReceiver*) can be called directly. The pre/postconditions of each locution are hard-coded in the program to detect the state of a dialogue. The combination rules which are illustrated in Figure 5.3, are implemented using the behaviour control mechanisms of JADE. When a buyer needs to check a proof, the program will use a command to call the external software Coq and check the correctness of a proof. In our dialogue model, only two participants are allowed.

Table 5.5: Decision making for different auction services

AuctioneerID	Q1 ($w = 0.35$)	Q2 ($w = 0.15$)	Q3 ($w = 0.1$)	Q5 ($w = 0.25$)	Q6 ($w = 0.15$)	Total Score	Decision ($\varepsilon = 70$)
A1	100	100	80	100	80	95	Y
A2	0	50	70	80	0	34.5	N
A3	100	50	80	80	100	85.5	Y
A4	-50	80	100	50	50	24.5	N
A5	100	50	70	50	50	69	N

In the experiment, there were five different auctioneers, from A_1 to A_5, which is shown in the first column of Table 5.5. Each of the auctioneers holds an auction service. A buyer dialogues with each of them based on the questions $\{Q_1, Q_2, Q_3, Q_5, Q_6\}$. This table shows the grades given for each questions as well as the total scores, the grading is based on Table 5.4. For the auction hold by A_1, because the value of the total score is greater than the *admissibility threshold*, ε, whose value is 70, the buyer decides to join this auction after the dialogue. For A_4, the score for Q_1 is -50 which means that the result of the proof check failed. This table shows that a buyer can make decisions for different auction services on the basis of the same questions and grading system through dialogues.

Table 5.6 shows the result of our second experiment. We kept the value of final scores (the second column) of each service the same as in Table 5.5. Then, by changing the *admissibility threshold* from 60 to 100, the number of services that are rejected has increased. The value of ε reflects the subjective belief of a buyer. If the buyer has

Table 5.6: Decision making for different auction services with different admissibility threshold

AuctioneerID	Total Score	Decision ($\varepsilon = 60$)	Decision ($\varepsilon = 70$)	Decision ($\varepsilon = 80$)	Decision ($\varepsilon = 90$)	Decision ($\varepsilon = 100$)
A1	95	Y	Y	Y	Y	N
A2	34.5	N	N	N	N	N
A3	85.5	Y	Y	Y	N	N
A4	24.5	N	N	N	N	N
A5	69	Y	N	N	N	N

higher expectations from the service, then the value of ε will be higher.

In the third experiment, we assume that there is only one auctioneer and three questions (Q_1, Q_2 and Q_5). There are three buyers (B_1, B_2 and B_3) whose weights for each question are different. We use the same grading Table 5.4 to score each question as in the previous experiments. The *admissibility threshold* of all buyers is set to 70. The last column shows that B_2 and B_3 choose to join the auction while B_1 refused to. Table 5.7 shows that, even if each buyer gets the same justification from the same auctioneer and they have the same *admissibility threshold*, the weights of questions can influence the result of the decision making.

Table 5.7: Decision making for the same auction service with different weight

	Q1($w = 0.20$)	Q2($w = 0.50$)	Q5($w = 0.30$)	Total Score	Decision ($\varepsilon = 70$)
B1	100	50	80	69	N
	Q1($w = 0.30$)	Q2($w = 0.50$)	Q5($w = 0.20$)	Total Score	Decision ($\varepsilon = 70$)
B2	100	50	80	71	Y
	Q1($w = 0.40$)	Q2($w = 0.35$)	Q5($w = 0.25$)	Total Score	Decision ($\varepsilon = 70$)
B3	100	50	80	77.5	Y

The protocol proposed in this chapter satisfied a set of desiderata [74].

- *Stated dialogue purpose*: The purpose of the dialogue is to enable buyer agents to make decision by querying the auctioneer. All participants are aware of this purpose before they enter in the dialogue.
- *Diversity of individual purposes*: The dialogue model lets agents achieve their own purposes in terms of inquiry with peers.
- *Inclusiveness*: There is no elimination of agents.
- *Transparency*: The protocol syntax and semantics are public and available to all participants, along with the combination rules of locutions.
- *Fairness*: The locutions and protocol are the same for all participants, which means that no agents have any privileges in the society.
- *Rule-consistency*: All protocol rules are consistent with the syntax and semantics.

- *Separation of Syntax and semantics*: The syntax and semantic of the proposed locutions are separately defined.
- *Encouragement of resolution*: The dialogue will terminate based on the dialogue driven algorithm and termination rules.
- *Discouragement of disruption*: The commitment rules preclude disruptive behaviours. For example, the buyer agent cannot ask the same question more than once.
- *System simplicity*: There are seven locutions in this model. Only a set of locutions are permitted to utter in each stage of the dialogue. Agents take turns to make locutions in bipartite dialogues.
- *Computational Simplicity*: In our dialogue system, we use the Coq proof checker to check a proof, which reduces the complexity compared to generate a proof. The length of the dialogue resulting from the protocol is: $1+4|H|+3$, where $|H|$ is the number of questions in the knowledge base of a *buyer*.

5.6 Related Work

The formal study of human argument and dialogue has been proposed for modeling agent interactions for more than a decade [73]. In the seminal work of [112], a typology of primary dialogue types has been provided to model human dialogues. Dialogues are categorized as six primary types: *Information-seeking Dialogues* (where participants aim to exchange knowledge); *Inquiry Dialogues* (where participants aim to find or destroy a proof of hypothesis); *Persuasion Dialogues* (where participants aim to resolve conflicts of opinions); *Negotiation Dialogues* (where participants aim to make a deal on the conflicts of interests); *Deliberation Dialogues* (where participants aim to decide best available course of action) and *Eristic Dialogues* (where participants quarrel verbally that aim to vent grievances). Most of the dialogues among humans or agents may involve mixtures of these dialogue types, which are called embedded.

In formal dialogue games, players interact with each other by making utterances according to previously defined rules. In the work of McBurrney and Parsons [71], five dialogue game rules have been identified. These rules include: *Commencement Rules* which define the circumstance under which the dialogue commences; *Locutions*, which define the rules that indicate what are permitted; *Combination Rules*, which define the contexts under which particular locutions are permitted or not; *Commitments*, which define the rules that indicate the circumstances under which participants express commitment to a proposition; and *Termination Rules*, which define the circumstances under which the dialogue ends. FIPA ACL [38] is a standard language for agents to communicate in. FIPA ACL contains 22 locutions (e.g. inform, request, refuse

and agree) which make agents share knowledge and negotiate contracts. However, FIPA ACL does not support argumentation statements. A protocol named *Fatio* with five locutions (*assert, question, challenge, justify* and *retract*) has been proposed for argumentation by [72]. In our work, we have presented a dialogue model, which is in accordance with the rules for a dialogue game. The difference between our work and the protocol of *Fatio* is that we focus on the type of inquiry dialogue. In [18], a formal inquiry dialogue system based on Defeasible Logic Programming [40] has been presented. In this inquiry dialogue system, three locutions *open*, *assert* and *close* are defined for generating dialogues. Besides, they also define an exhaustive strategy to decide which move to make. We defined our inquiry dialogue by adding more locutions to the model and we also introduced the idea of attack to enable agents to reject unsatisfied justifications. In the work of [85], an inquiry dialogue has been proposed to enable agents to negotiate over ontological correspondences. In this dialogue, agents can not only make *assert* moves to assert beliefs, they also can *object* to a belief by providing an attack and *accept* or *reject* beliefs. In the ArguGRID [108] project, Web service, agents and argumentation technique have been combined to support decision making and negotiations inside Virtual Organizations. ArgSCIFF [109] is a project that aims to make Web service reasoning more visible to potential users by using dialogues for service interaction. It extends the kind of request-response interaction among Web services and makes agents justify the interaction outputs. The difference between their work and our approach is that we utilize the PCC paradigm to the dialogue to enable agents to automatically check formal proofs that provided by the service provider. An automate negotiation approach has been presented among agents in an open environment in [102]. In this work, protocols are expressed in terms of a shared ontology. Based on the shared ontology, agents can learn to tune strategies based on existing algorithms. We extend their work by introducing dialogue games into the scenario.

In our previous work, we have represented some properties (e.g. truthful bidding is a dominant strategy in a Vickrey auction) in the theorem prover Coq [9]. We have implemented the Proof-Carrying Code (PCC) [80] paradigm to certify the desirable properties of online auction services, which are written in OWL-S. PCC is a paradigm that enables a computer system to automatically ensure that a computer code provided by a foreign agent is safe for installation and execution. We use the interactive theorem prover Coq because it has been developed for more than twenty years [34] and is widely used for formal proof development in a variety of contexts related to software and hardware correctness and safety. Coq has been used to develop and certify a compiler [61], and a fully computer-checked proof of the Four Colour Theorem has been created by [41]. In the work of [8], we have formalized an auction service using OWL-S, then we translate this representation into a program that is written in Coq.

The semantics of these expressions is preserved in this transaction. Then, we use proof tactics to prove desirable properties of these expressions within the theorem prover Coq.

5.7 Summary

In this chapter, we have proposed a dialogue game to enable buyer agents to automatically query an auctioneer before deciding whether or not to join an online auction. We have formally described this inquiry dialogue model by defining the rules of locutions and commitments. The auctioneer and the buyer share a common knowledge enabling them to communicate and make sense of each other's arguments. But they also have private knowledge. For example, the questions related to properties of interest to the buyer are not known to the auctioneer until they have been revealed through the dialogue. The buyer has a ranking function over the questions in the form of a preference level, which is used to drive the dialogue forward. A scoring function over possible answers in line with predefined expectations is used to decide whether to join or not an auction.

A noticeable feature of our dialogue game is that it uses formal proofs as arguments. Thus, it combines formal evidence with informal evidence in an interaction. We have implemented our dialogue framework from within JADE wherein we have integrated the Coq theorem prover in the Proof-Carrying Code paradigm enabling an agent to check whether a proof of a given auction desirable property is correct or not. Experimental results have demonstrated the feasibility as well as the validity of our approach.

In this chapter, we have presented a dialogue game to enable buyer agents to check some desirable properties of a service specification. To extends the model of dialogue games, we will consider the communication during a service and the composition of different services in a transaction. We will simulate virtual markets that are composed of various services using dialogue games in the next chapter.

Chapter 6

Dialogue Driven Semantic Web Service Composition

6.1 Introduction

Future agent-mediated e-commerce systems will deal with software agents involved in financial transactions and in legally-binding contracts on behalf of humans in the Internet. To provide a good foundation for automating the use of web services, Semantic Web Services are proposed to automate service discovery, interoperation, and composition. OWL-S [69] is an ontology language to describe the semantics of Web services, which can be used to define the capabilities, requirements and internal structures of Web services. The OWL-S provides a good approach to describe the process of an online auction service. Nevertheless, using OWL-S alone to implement online auction services is not sufficient for us, although we can provide a specification for auction services. In a sequential auction system, such as the English auction, the auctioneer should report current highest bid to every bidder at the beginning of each round and then bidders can make bids based on the current highest bid and their own strategy. The bidding process of an English auction requires frequent information exchange. The OWL-S model provides a way to describe what is carried out in a service and OWL-S WSDL grounding provides methods to implement an atomic process that is described in the service's process model. This grounding is not sufficient to implement the request-response communication approach in an English auction for example. Thus, we have introduced the idea of information-seeking dialogues to our implementation of online auction systems.

After introducing a dialogue into an inner service, we take another step forward to expand the dialogue to dynamically compose Semantic Web Services. In the example of the Semantic Web online book store service Congo.com[1], a fictitious B2C site is written in OWL-S. In this example, the process of selecting books, collecting buyer information, collecting payment and delivering books are clearly described. Although this service

[1] http://www.ai.sri.com/daml/services/owl-s/1.2/examples.html

can be successfully implemented as a good example of a semantic web service, it can be improved to achieve more complex functionality. For example, we can enable the book seller to persuade a buyer to enter another book selling service or to buy additional books related to an already purchased book. These kinds of communications need some form of dialogues, which are beyond the exchange of static data.

The specification of e-commerce systems is a crucial research issue in the development of distributed applications on the Internet [39]. In this chapter, we present a machine understandable description of an online trading system which includes the services of auction, payment and delivery. Then, we introduce dialogues to Semantic Web Services so that the interactivity among participants of an agent mediated e-commerce system can be improved and the dynamic composition of Semantic Web Services can happen. To achieve this aim, we formalize Semantic Web Services and dialogues as automata that describing the interactions among participant agents. To realize the goal of service composition, we have defined Composite Dialogue Service Automata (CDSA) to synthesize services into a global automaton. Finally, we compose an auction service with online payment and delivery services to validate our approach.

This chapter is structured as follows. In Section 6.2, we set up an online trading scenario and give a general discussion on how Semantic Web Services and dialogue games can support it. Then, we briefly describe the Semantic Web Service and agent dialogue framework. Formal dialogue model is introduced in Section 6.3. In Section 6.4, we will describe the proposed automata approach, which combines the techniques described in Section 6.2. Examples that implement our automata are described in Section 6.5. Related work is presented in Section 6.6 and summary is given in Section 6.7.

6.2 Background

6.2.1 The scenario: Auction, Payment, and Delivery

In our scenario, Brad is a student who wants to buy some books which are related to his major through online auction site. One traditional solution for this situation is that he can visit the online auction web site *eBay* and search for the books that he preferred. After one book is found, he joins in an auction to bid for it. If Brad succeeds in one auction, he can discuss with the seller about the delivery issue and decide the terms of payment. Brad should repeat all of the above steps for each of the remaining books.

Another solution is that Brad delegates this task to his agent, and then this agent completes this task automatically. To realize the second solution, the online auction system should be published in a way to be understood by the agent. This requires a semantically rich language to support the understanding. The buyer agent should be able to reason on the rules of an auction and make sure that he will not lose unnecessarily money by participating into the auction. Another issue to this solution

is that the system should enable communication among buyer agents and seller agents. For example, the buyer agent can tell the seller agent which delivery company he wants to choose or the seller agent can recommend and persuade the buyer agent to buy related books, and then invoke another auction process.

The second solution combines the advantages of Semantic Web Services and dialogue games. A Semantic Web service provides a way to express machine understandable services and dialogue games enable service consumer to communicate with service provider so that the buyer agents can make decisions based on the result of communication. The dialogue communication is a collaborative business activity rather than a formatted client-server interaction.

6.2.2 Service Description

OWL-S has been designed as an ontology language to make Web service descriptions computer-interpretable. Thus, we rely on OWL-S to implement the Web service and service automata in this chapter. To describe a Semantic Web Service, the domain ontology of a service should be developed in the first stage. The domain ontology describes the semantics of terms representing an area of knowledge and give explicit meaning to the information. For example, we have defined classes `Agent`, `PaymentMethod`, `DeliveryCompany` as the following declaration.

```
<owl:Class rdf:ID="Agent"/>
<owl:Class rdf:ID="DeliveryCompany">
    <owl:oneOf rdf:parseType="Collection">
        <DeliveryCompany rdf:ID="EMS"/>
        <DeliveryCompany rdf:ID="FedEx"/>
    </owl:oneOf>
</owl:Class>
<owl:Class rdf:ID="PaymentMethod">
    <owl:oneOf rdf:parseType="Collection">
        <PaymentMethod rdf:ID="Alipay">
        <PaymentMethod rdf:ID="CreditCard">
    </owl:oneOf>
</owl:Class>
```

Then OWL-S ontology will be used to describe the trading mechanism. As the trading mechanism has been treated as Web service, the IOPEs (i.e., inputs, outputs, preconditions and results) should be defined explicitly. In the service of *Payment*, the inputs are `Account` and `PaymentMethod` while the output is `Gift`. We define `Gift` as benefits for a specific payment type. In the service of *Delivery*, the inputs are an account of an agent, products, delivery company and address, and the output is an announcement. The following declarations display a segment of the inputs and outputs of the service *Payment*.

```
<process:hasInput>
    <process:Input rdf:ID="PayMethod">
        <process:parameterType rdf:datatype="&xsd;anyURI"
            >http://www.owl-ontologies.com/PaymentOntology.owl#PaymentMethod</process:parameterType>
    </process:Input>
```

```
</process:hasInput>

<process:hasOutput>
    <process:Output rdf:ID="PaymentGift">
        <process:parameterType rdf:datatype="&xsd;anyURI"
            >http://www.owl-ontologies.com/PaymentOntology.owl#Gift</process:parameterType>
    </process:Output>
</process:hasOutput>
```

In OWL-S, the term `inCondition` is used to express the binding between inputs variables and the particular result variables. In the service of *Payment*, we would like to express that if an agent chooses to use `Alipay` as the payment method, then this agent will get a gift. This result can be expressed by SWRL as following.

```
<process:inCondition>
    <expr:SWRL-Condition rdf:ID="PaymentAli">
        <expr:expressionObject>
            <rdf:List>
                <rdf:first>
                    <rdf:Description>
                        <rdf:type rdf:resource="&swrl;ClassAtom"/>
                        <swrl:argument1 rdf:resource="#Alipay"/>
                        <swrl:classPredicate rdf:resource="#PaymentMethod"/>
                    </rdf:Description>
                </rdf:first>
                <rdf:rest rdf:resource="&rdf;nil"/>
            </rdf:List>
        </expr:expressionObject>
    </expr:SWRL-Condition>
</process:inCondition>
<process:hasEffect>
    <expr:SWRL-Expression rdf:ID="effect">
        <expr:expressionObject>
            <rdf:List>
                <rdf:first>
                    <rdf:Description>
                        <rdf:type rdf:resource="&swrl;SameIndividualAtom"/>
                        <swrl:argument2 rdf:resource="#gift"/>
                        <swrl:argument1 rdf:resource="#PaymentGift"/>
                    </rdf:Description>
                </rdf:first>
                <rdf:rest rdf:resource="&rdf;nil"/>
            </rdf:List>
        </expr:expressionObject>
    </expr:SWRL-Expression>
<process:hasEffect>
```

Two sorts of process can be invoked in an OWL-S process model. One is *atomic process*, which corresponds to a one-step request-response message exchange. Both of the services *Payment* and *Delivery* are described as atomic process. The other is *composite process*, which corresponds to multi-step actions that linked together by control flows and data flows. The control flows are described using constructs such as *sequence, split, if-then-else* and *iterate* etc. Data flow describes how information is transferred from one step of a process to another. The example of a composite process can be found in Section 5.2.1.

6.2.3 Agent Dialogue Framework

Formal dialogue games have been applied in multi-agent systems to enable software agents to interact with each other. Walton and Krabbe [112] propose a typology of primary dialogue types to model dialogues. In their typology, six primary dialogue types are categorized: information-seeking dialogues, inquiry dialogues, persuasion dialogues, negotiation dialogues, deliberation dialogues and eristic dialogues. McBurrney and Parsons [71] identify five dialogue game rules in formal dialogue games. These rules include Commencement Rules, Locutions, Combination Rules, Commitments and Termination Rules.

Three-level hierarchical formalism is presented in [71] for agent dialogues. The lowest level is the topic layer which displays the subject of a dialogue. The middle level is the dialogue layer which is the combination of six primary types of dialogues. The top level is the control layer which represents the selection and transition of specific dialogue types. The control layer is composed of two components: *Atomic Dialogue-Types*, which include five dialogue types except eristic dialogue in the taxonomy of Walton and Krabbe; and *Control Dialogues* that have their discussion subjects of other dialogues. Then five dialogue combinations are defined to represent the combination of atomic or control dialogues. The five dialogue combinations are: *iteration, sequencing, parallelization, embedding and testing.* Given the above definitions, McBurrney and Parsons define an Agent Dialogue Framework (ADF) as a five-tuple $(A, L, \prod_{Atom}, \prod_{Control}, \prod)$, where:

- A is a set of agents.
- L is a logical language for representation of discussion topics.
- $\prod_{Atom}$ is a set of atomic dialogue-types.
- $\prod_{Control}$ is a set of Control dialogues.
- $\prod$ is the closure of $\prod_{Atom} \cup \prod_{Control}$ under the combination rules.

6.3 Formal Dialogue Model

In our scenario, two types of dialogue are considered: *information-seeking dialogue* and *persuasion dialogue*. In an *information-seeking dialogue*, we assume that there are one information requester and at least one information responder. While in a *persuasion dialogue*, we assume that there are two participants.

6.3.1 Locutions in the Dialogue Framework

We define Δ as the representation of atomic dialogue types, and t as the topic of a dialogue. Seven locutions are defined to control the dialogue, assuming $\delta \in \Delta$:

- **begin**(δ, t). It makes possible to start a dialogue with type δ on the topic of t.
- **agree**(δ, t). It is used to accept the request to start a dialogue whose type is δ and topic is t.
- **disagree**(δ, t). It is used to reject the request to start a dialogue whose type is δ and topic is t.
- **return_control()**. It is used to propose a request to jump from one basic type dialogue to a control dialogue.
- **agree_return_control()**. This accepts to return a control dialogue.
- **disagree_return_control()**. This rejects to return a control dialogue.
- **close()**. It is used to close a dialogue.

Let Φ be the set of variables used to express the statements uttered by agents. Assume that $\phi \in \Phi$. Three additional locutions are used in the information-seeking dialogue.

- **propose**(ϕ). It proposes a statement ϕ.
- **offer**(ϕ). It gives a response ϕ to a proposition.
- **pass()**. It gives an empty statement as a response.

Four extra locutions are defined in the persuasion dialogue.

- **assert**(ϕ). It gives an assertion of statement ϕ.
- **accept**(ϕ). It is used to accept an assertion.
- **argue**(ϕ). It gives an explanation of ϕ.
- **reject**(ϕ). Reject a statement ϕ.

6.3.2 Commitment Rules

The dialogue assumes that the commitments uttered by all participants are recorded in a *commitment store* (CS). The following table 6.1 shows the changes of the CS for each locution, where p represents the speaker in a move. Note that other locutions that are not contained in the table will not lead to the updating of the commitment store.

Table 6.1: Commitment rules

begin(δ, t)	$CS(p) = CS(p) \cup \{t\}$
agree(δ, t)	$CS(p) = CS(p) \cup \{t\}$
propose(ϕ)	$CS(p) = CS(p) \cup \{\phi\}$
offer(ϕ)	$CS(p) = CS(p) \cup \{\phi\}$
assert(ϕ)	$CS(p) = CS(p) \cup \{\phi\}$
accept(ϕ)	$CS(p) = CS(p) \cup \{\phi\}$
argue(ϕ)	$CS(p) = CS(p) \cup \{\phi\}$

6.3.3 Dialogue Rules

The protocol for an information seeking dialogue is described in Figure 6.1. Each node in this graph is a locution, and the outgoing arcs from one node describe the possible following moves. The nodes with gray background represent a requester while the nodes with white background represent a responder. We also assume that all the responders should give responses to the requester in one round of the dialogue so that the requester can make another iteration.

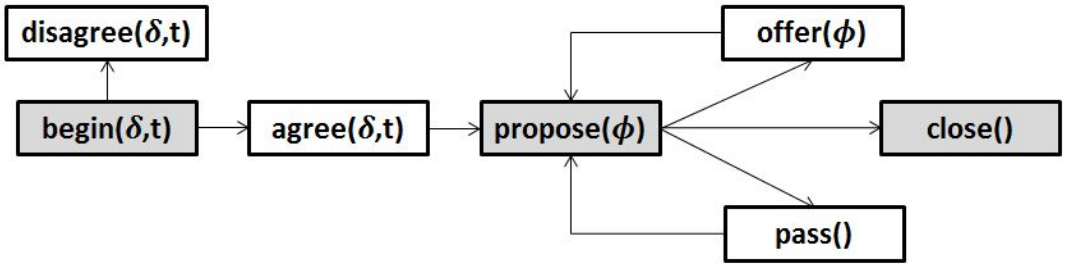


Figure 6.1: Rules for an information seeking dialogue (grey nodes are for the auctioneer, white nodes for the buyer)

Figure 6.2 shows the protocol for a persuasion dialogue in our model. In this graph, one participant represented by nodes with gray background tries to persuade the other participant to accept his assertions. When a dialogue is not in the control layer, the locutions **return_control()**, **agree_return_control()** and **disagree_return_control()** can be uttered in anytime. Another two moves **agree_return_control()** and **disagree_return_control()** must follow by the locution **return_control()** control. Note that our dialogue model supports for embedded dialogues, which means that we can insert one dialogue into another. We will not discuss embedded dialogues in this chapter, because the topic of this chapter focuses on service generation.

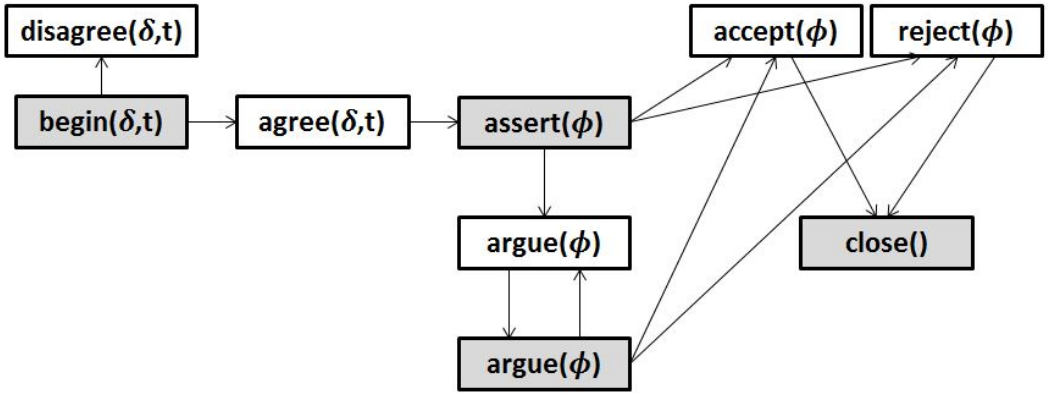


Figure 6.2: Rules for a persuasion dialogue (grey nodes are for the auctioneer, white nodes for the buyer)

6.4 Dialogue Service Automata

Service automata are used to solve services discovery and composition in a context-aware system [118], to perform planning for Web services composition [83] and to describe global behavior of services composition [35]. To improve the interactivity of Semantic Web Services, we define a Semantic Web Service as an automaton by applying the following agent dialogue framework (ADF).

Definition 14. *Given an agent dialogue framework (ADF), an Dialogue Service Automata is defined by a tuple:* $\langle A, \sum, Q, \delta, I, F \rangle$ *, where:*

- A *is an Agent Dialogue Framework (ADF).*
- $\sum = \prod \cup P$ *is the union of dialogues and inner operations of one service.* $\prod$ *represents the closure of* $\prod_{Atom} \cup \prod_{Control}$ *under the combination rules defined in the ADF,* P *represents the inner operations of the service.*
- Q *is the finite set of states of the automata* ($Q \neq \emptyset$).
- δ *is the transition function, that is,* $Q \times \Sigma \rightarrow Q$.
- I *is the initial state of a service and* $I \in Q$.
- F *is a finite set of final states and* $F \subseteq Q$.

In an ADF, $\sum$ is composed of two components. One is the dialogue among service providers and service consumers. For example, an auctioneer asks for new bids in a round of an English Auction. The other is the inner operation in one service. For example, after finding the winner of an auction, the service provider updates the value of last payment. The transition function δ associates current state $q \in Q$ with current

action $a \in \sum$ and set $\delta(q, a) \subseteq Q$ to next state. There may exist multiple initial states and final states in this automaton. The initial state I represents the precondition of a service, while the final state F represents the postcondition of a service. The implementation of a DSA is shown in Algorithm 2.

Algorithm 2 Implementation of DSA

Require: An Agent Dialogue Framework ADF.
Ensure: The postcondition of one service is satisfied.

```
The DSA starts when the precondition of one service is satisfied.
while The final state of one service is not reached do
    while Need information exchange among consumers and providers do
        Start a dialogue d until the termination of this dialogue.
        Perform the transition δ(q_i, d) to translate the system from current state q_i to
        next state q_{i+1}.
    end while
    if An inner action a is operated then
        Perform the transition δ(q_i, a) to reach next state q_{i+1}.
    end if
end while
```

Composite services provide higher value than individual services and can be predominant in the business world [70]. In order to synthesize two Semantic Web Services, we propose Composite Dialogue Service Automata (CDSA). We use the composite operator $\otimes$ to represent the composition of two Semantic Web Services. Given automata $\langle A^{S1}, \sum^{S1}, Q^{S1}, \delta^{S1}, I^{S1}, F^{S1} \rangle$ corresponding to Service S1 and automata $\langle A^{S2}, \sum^{S2}, Q^{S2}, \delta^{S2}, I^{S2}, F^{S2} \rangle$ corresponding to Service S2. S1 and S2 can be composite if

- $F^{S1} \cap I^{S2} \neq \emptyset$
- $\forall a \in F^{S1} \cap I^{S2}, a \in K(I^{S2})$, where $K(I^{S2})$ represents the Initial State of S2 which is denoted by K predicates.

Definition 15. *Given a specific Agent Dialogue Framework (ADF), a composition Service S of* $S1 \otimes S2$ *is defined as tuple* $\langle A^S, \sum^S, Q^S, \delta^S, I^S, F^S \rangle$*, where:*

- A^S *is an Agent Dialogue Framework (ADF).*
- $\sum^S = \prod^{S1} \cup \prod^{S2} \cup P^{S1} \cup P^{S2}$.
- $Q^S = Q^{S1} \otimes Q^{S2} = \{< u, v > | u \in Q^{S1} \wedge v \in Q^{S2}\}$.
- $\delta^S : Q^S \times \sum^S \rightarrow Q^S$.
- $I^S = \{< u, v > \mid u \in I^{S1} \wedge v \in I^{S2}\}$.
- $F^S = \{< u, v > \mid u \in F^{S1} \wedge v \in F^{S2}\}$.

6.5 Example of Dialogue Service Automata

We illustrate our model with dialogue occurrence among one auctioneer and two bidders in an English auction house. The auctioneer persuades the winner to choose a recommended payment approach and delivery company. In this example, the information seeking dialogue is used in the process of the English auction service. Two persuasion dialogues are used to compose the services: combine the English auction with a payment service, and combine the payment service with a delivery service. The dialogue model is implemented in the agent platform JADE [12]. The auctioneer is represented as A, while the two bidders are represented as B_1 and B_2.

(1) A: **begin**(information seeking,English Auction)
(2) B_1: **agree**(information seeking,English Auction)
(3) B_2: **agree**(information seeking,English Auction)
(4) A: **propose**(hasPrice(Book,15))
(5) B_1: **offer**(18)
(6) B_2: **offer**(20)
(7) A: **propose**(hasPrice(Book,20))
(8) B_1: **offer**(22)
(9) B_2: **pass**()
(10) A: **propose**(Winner(B_1), Payment(22))
(11) A: **close**()
The information seeking dialogue for an English auction is closed. Then the auctioneer persuades the winner to use the recommend payment approach.
(12) A: **begin**(persuasion, Payment Method)
(13) B_1: **agree**(persuasion, Payment Method)
(14) A: **assert**(PaymentMethod(Alipay))
The auctioneer suggests the buyer to use Alipay to pay for the item.
(15) B_1: **argue**(PaymentMethod(DebitCard))
The buyer would like to pay by a debit card.
(16) A: **argue**(inCondition(PaymentMethod(Alipay)),hasEffect(PaymentGift(gift)))
The auctioneer argues that if the buyer pays use Alipay, then this buyer will get a gift.
(17) B_1: **accept**(PaymentMethod(Alipay))
B_1 submits his payment information to the service.
(18) A: **close**()
The auctioneer closes the persuasion dialogue.
(19) A: **begin**(persuasion, Delivery Company)
(20) B_1: **agree**(persuasion, Delivery Company)
(21) A: **assert**(DeliveryCompany(EMS))
(22) B_1: **accept**(DeliveryCompany(EMS))
A invokes the EMS delivery service.
(23) A: **close**()
A closes the second persuasion dialogue.

6.6 Related Work

In [119], a framework called TAGA has been used to simulate an automated trading in dynamic markets. TAGA uses Semantic Web languages (RDF and OWL) to specify and publish underlying common ontologies and as a content language within the FIPA ACL messages. This framework extends the FIPA protocols to support open market auction services. However, the specification of auction services in this chapter does not describe the explicit process and rules of different auctions. For example, the rule

that a bidder cannot bid a lower price than the current bid in an English auction is not described. In [103], a common shared negotiation ontology has been used to help agents negotiate with each other. They provide a possible application of this approach to simulate online auction competition. The simulation is not a real implementation but they just illustrate the approach using negotiation ontology. In [96], they have integrated Semantic Web technologies into agent architecture to represent the knowledge and behavior of agents in a semantic manner. Although, their work is mainly about distributed knowledge management, the proposed approach can help us build an agent mediated and to automatically process online auction services. In [39], the authors have proposed a hybrid solution that uses OWL, SWRL rules, and a Java program to dynamically monitor and simulate a temporal evolution of social commitments. The shortage of this approach is that the action and time slice is fixed binding. In our work, we aim to build a system that is dynamically changing over time. The work of [85] has presented an inquiry dialogue and illustrated how agents negotiate in a scenario of ontological correspondences. The dialogue model provided in this paper inspired us to design the dialogue game that used in this chapter. In the work of [59], an agent architecture that integrates Semantic Web technologies and multi-agent systems has been proposed for developers to build knowledge management systems using software agents. They have implemented their system in JADE and utilized ACL to enable the communication. We extend their work by introducing difference types of dialogue games to achieve interaction among agents. A policy and contract extended agent model has been specified in [65]. In this model both systems and agents can be defined dynamically by means of policies. In [117], a policy driven model has been proposed to control the behaviours of agents according to high-level business requirements. In this model, ontology language is used to describe the business requirements, policies and the negotiation protocol. The difference is that our work focuses on the problem of Semantic Web Services composition.

6.7 Summary

The main advantage of a dialogue driven Semantic Web Service is that the message passing between the service provider (the auctioneer) and the consumer (the buyer) is dynamic. This means the consumer can challenge the provider while the provider can persuade the consumer to enter another service and consumers can ask questions or seek information from the service provider. The dialogue service automata approach can be used not only to describe the inner status of a specific service but also to describe services composition. The approach to deal with data flow within a process is not clearly defined in OWL-S, while arguments can contain data information for the next process. For example, a new bid can be passed to the auctioneer by an argument in the process of an English auction.

In this chapter, we have defined the dialogue service automata approach to extend the interactivity of the current Semantic Web Service framework. We have presented a semantic representation of the auction domain using OWL and a service-oriented approach for the semantic description of auction processes using OWL-S, which provides a formal and unified representation for online auction systems. By introducing semantics in the auction domain and combining the ontology with the Semantic Web Service technique, we have provided an approach for different software agents to automatically discover and invoke an online auction. We have presented our dialogue models that can generate information-seeking and persuasion dialogues. By combining the Semantic Web Service and dialogue games, dialogue service automata has been proposed. Finally, we have implemented an online auction service, which are written in OWL-S, by using JADE according to the dialogue service automata. In future work, we will illustrate our model by implementation embedded dialogues and introduce more locutions and rules to support negotiation dialogues.

Chapter 7

Verification of Combinatorial Auctions

7.1 Background

Imagine an open society of software agents owned by human beings and engaging into financial transactions through online auctions. Assuming that such software agents can understand auction protocols, we are interested in ensuring trust in the auction by enabling agents to formally verify desirable properties. More specifically, we consider combinatorial auctions and aim to establish well-known properties such as *incentive compatibility* by using the Coq [34] theorem prover, which can be used to generate machine-verifiable proofs. An auction is incentive compatible iff bidding its true valuation is the optimal strategy for every agent. Our motivation for this work is three-fold:

1. This can be used as a decision support system by enabling a participating agent to check properties of interest before deciding to join a given auction house or not.

2. The ability to build up machine-verifiable proofs of well-established properties of auction mechanisms will increase our confidence in proving previously unseen protocols.

3. This kind of capability is important in mechanism design wherein one aim to tailor a protocol with specific properties given the preferences or constraints of the participants.

There is by now a small but increasing body of literature on verifying or certifying auctions properties [101, 61, 9]. Most of this work is focused on single item auctions except the recent work reported in [24], which investigates the use of the Isabelle/HOL [82] theorem prover for the verification of combinatorial auctions. Combinatorial auctions involve combination of items making complex not only the bidding language but also the allocation and payment procedures. Moreover, for us to formally

verify a property of a given protocol, we need to specify both the protocol and the property. It turns out that Coq uses higher order logic, a logic language that is expressive enough to specify combinatorial auctions such as the VCG (Vickrey Clarke Groves) mechanism. Thus, we have specified the VCG protocol within Coq. In our previous work [7] OWL-S is used as the specification language but was limited to single item auctions. Note that the VCG protocol cannot be formalized within OWL-S since it involves quantification over valuation or bidding functions. Our formal specification of the VCG differs from that of [24] not only on the fact we have used different theorem provers but mainly on the fact we have relied upon set theory to explicitly define bidding pairs, allocation and payment functions, see Section 7.2 for more details.

We have then developed formal proofs for some incentive properties for the VCG mechanism. These incentive properties are:

- The payment of each agent is non-negative;
- The utility derived by a truthful agent is non-negative;
- The VCG mechanism is incentive compatible.

Observe that for the Vickrey auction, we have an allocation algorithm, which we have implemented and certified from within Coq in Chapter 1. In contrast, there is no known allocation algorithm for the general VCG mechanism since the optimal allocation problem for combinatorial auctions is NP-hard [104]. Instead, the allocation relies on a non-constructive existence of an optimal solution *argmax*. As a consequence, we have developed 1200 lines of Coq proof for the VCG against 200 lines for the Vickrey.

To sum up, the contributions of this work can be stated as follows:

1. We have fully specified VCG auctions by using set theory to represent bids, allocations, and payments.
2. We have developed machine-verifiable proofs of the fact that the VCG mechanism is incentive compatible.
3. We have also developed simple and elegant proofs for some basic properties proven before, see for example [7].

7.1.1 Game Theoretic Properties

Generally speaking a *combinatorial auction* [29] is composed of a set I of m items to be sold to n potential buyers. A bid is formulated as a pair $(B(x), b(x))$ in which $B(x) \subseteq I$ is a bundle of items and $b(x) \in Z^+$ is the price offer for the items in B. The *combinatorial auction problem* (CAP) is to find a set $X_0 \subseteq X$ such that, for a given a set of k bids, $X = \{(B(x_1), b(x_1)), (B(x_2), b(x_2)), \ldots, (B(x_k), b(x_k))\}$, the quantity $\sum_{x \in X_0} b(x)$ is maximal subject to the constraints expressed as for all

$x_i, x_j \in X_0 : \ B(x_i) \cap B(x_j) = \emptyset$ meaning an item can be found in only one accepted bid. We assume *free disposal* meaning that items may remain unallocated at the end of the auction.

As shown in [87], the CAP is NP-hard implying that approximation algorithms are used to find near-optimal solutions or restrictions are imposed in order to find tractable instances of the CAP [104] wherein polynomial time algorithms can be found for a restricted class of combinatorial auctions. A combinatorial auction can be *sub-additive* (for all bundles $B_i, B_j \subseteq I$ such that $B_i \cap B_j = \emptyset$, the price offer for $B_i \cup B_j$ is less than or equals to the sum of the price offers for B_i and B_j) or *super-additive* (for all $B_i, B_j \subseteq I$ such that $B_i \cap B_j = \emptyset$, the price offer for $B_i \cup B_j$ is greater than or equals to the sum of the price offers for B_i and B_j).

Game theory *mechanism*, see for example [29], is usually used to describe auctions, thus providing decision procedures that determine the set of winners for the auction according to some desired objective. An objective may be that the mechanism should maximize the *social welfare*, which can be for example the sum of all agents' utilities in the auction. Such a mechanism is termed *efficient*. For open multi-agent systems, another desirable property for the mechanism designer can be *strategyproofness* (truth telling is a dominant strategy for all agents). A mechanism that is strategyproof has a dominant strategy equilibrium.

A wellknown class of mechanisms that is efficient and strategyproof is the Vickrey-Clarke-Groves (VCG), see for example [29]. The VCG mechanism is performed by finding (i) the allocation that maximizes the social welfare and (ii) a pricing rule allowing each winner to benefit from a discount according to his contribution to the overall value for the auction. To formalise the VCG mechanism, let us introduce the following notations:

- $\mathcal{X}$ is the set possible allocations
- $v_i(x)$ is the true valuation of $x \in \mathcal{X}$ for bidder i
- $b_i(x)$ is the bidding value of $x \in \mathcal{X}$ for bidder i
- $x^* \in \text{argmax}_{x \in \mathcal{X}} \sum_{i=1}^{n} b_i(x)$ is the optimal allocation for the submitted bids.
- $x^*_{-i} \in \text{argmax}_{x \in \mathcal{X}} \sum_{j \neq i}^{n} b_j(x)$ is the optimal allocation if agent i were not to bid.
- u_i is the utility function for bidder i.

The VCG payment p_i for bidder i is defined as

$$\begin{aligned} p_i &= b_i(x^*) - \left(\sum_{j=1}^{n} b_j(x^*) - \sum_{j=1,\ j \neq i}^{n} b_j(x^*_{-i}) \right) \\ &= \sum_{j=1,\ j \neq i}^{n} b_j(x^*_{-i}) - \sum_{j=1,\ j \neq i}^{n} b_j(x^*), \end{aligned} \tag{7.1}$$

and then, the utility u_i for agent i is a quasi-linear function of its valuation v_i for the received bundle and payment p_i.

$$u_i(v_i, p_i) = v_i - p_i. \tag{7.2}$$

We illustrate a VCG auction as an example.

Example 7.1.1 (A Combinatorial VCG Auction)**.** *Let an auctioneer sells two items A and B in the Internet. Let three agents (b_1, b_2 and b_3) submit the bids as shown in Table 7.1, where we assume that all agents are bidding truthfully.*

Table 7.1: Bids from the agents for different bundles

	A	B	AB
b_1	5	6	8
b_2	6	4	9
b_3	4	7	10

This yields one value-maximizing allocation x^*, i.e. b_2 wins item A with bid 6 and b_3 wins item B with bid 7. Then, the payment of b_2 is:

$$p_2(A) = (b_1(A) + b_3(B)) - b_3(B) = (5 + 7) - 7 = 5$$

and the payment of b_3:

$$p_3(B) = (b_2(A) + b_1(B)) - b_2(A) = (6 + 6) - 6 = 6$$

Finally, the utilities of b_2 and b_3 are:

$$u_2 = v_2(A) - p_2(A) = 6 - 5 = 1$$

and

$$u_3 = v_3(B) - p_3(B) = 7 - 6 = 1.$$

Let p_i^* and p_i be the payments for agent i when it bids its true valuation v_i and any number b_i respectively. Note p_i, p_i^* are functions of b_{-i}. The strategyproofness of the mechanism amounts to the following verification:

$$\forall i,\ \forall v_i,\ \forall b_i\ u_i(v_i, p_i^*(b_{-i})) \geq u_i(v_i, p_i(b_{-i})). \tag{7.3}$$

In the equation (7.1), the quantity $\sum_{j=1,\, j\neq i}^{n} b_j(x^*_{-i})$ is called the *Clark tax*. Another interesting property of this VCG mechanism is that it is *weakly budget balanced* [27] meaning the sum of all payments is greater than or equal to zero. For the VCG properties (e.g. strategyproof) to hold, the auctioneer must solve n+1 hard combinatorial optimization problems (the optimal allocation in the presence of all bidders followed by n optimal allocations with each bidder removed) exactly.

Single item auctions are a particular case of combinatorial auctions wherein one item is sold at a time. Common single item auctions are the English, Dutch or Vickrey auctions, see for example [29]. For the Vickrey auction, the payment for agent i is defined as:

$$\begin{aligned} p_i &= b_i(x^*) - \left(\sum_{j=1}^{n} b_j(x^*) - \sum_{j=1,\, j\neq i}^{n} b_j(x^*_{-i})\right) \\ &= \sum_{j=1,\, j\neq i}^{n} b_j(x^*_{-i}) \;=\; \text{the second highest bid.} \end{aligned} \tag{7.4}$$

7.2 Formalization of VCG

Agent-mediated online auctions are applications wherein data can be processed by automated reasoning tools. Logic-based languages are useful tools to model and reason about systems. They allow us to specify behavioral requirements of components of a system and formulate desirable properties for an individual component or the entire system.

In general, logic-based languages are chosen to balance their *expressivity*, *completeness* and *decidability* of their underlying logic. For example, the expressivity of first-order logic includes quantifiers, predicates and functions to objects, which is higher than the complete and decidable propositional logic. In addition to this, quantification over functions and predicates included in higher-order logic is more expressive than first-order logic. First-order logic is complete but undecidable, whereas higher-order logic is neither complete nor decidable. In our work, we advocate using the Coq language for the following reasons:

- It is expressive enough so that we can specify combinatorial auctions within it.
- It provides us with a higher order logic therefore not decidable but enabling us to specify mechanisms such as the VCG involving quantification over functions.
- Formal proofs of desirable properties can be developed from within Coq.

We have formalized the VCG mechanism within Coq and we will describe the formalization of the VCG mechanism in the remaining section.

In the formalization, we use a Coq library *List* to define the operation of *Set*. For example, we have defined functions to implement the operations of *union* and *intersection*, the relation of *include*. Bidders and items are represented as typed set of natural numbers `Bidder` and `Item` respectively. Prices of items are coded as `Price` and are restricted to nonnegative integers in order to facilitate proof development. A bid is then coded as a couple formed by a set of items and a price for that set of items. To associate a bid with the corresponding bidder, we have defined a record `BiddingRecord` to represent a well-formed bidding data as follows.

```
(*define a bid record as a triple*)
```

```
Record BiddingRecord :=
    mk_bidding_record {
        record_bidder : Bidder;
        bid_items : ItemSet;
        bid_price : Price}.

(*define bidding data as a list of bid records*)
Definition BiddingData : Set :=
    (list BiddingRecord).
```

We can check the legality of a set of bids as follows:

```
(*define the well-formedness property of the bidding data*)
Definition NoDupItemSetEachBidder
    (data : BiddingData) :=
  forall (n1 n2 : nat) (r1 r2 : BiddingRecord),
   n1 <> n2 ->
   nth_error data n1 = Some r1 ->
   nth_error data n2 = Some r2 ->
   record_bidder r1 = record_bidder r2 ->
   ~ ItemSetEq (bid_items r1) (bid_items r2).
```

In addition to legality checks, we can create record of legal bidding data. This is useful for the allocation, which is defined as a set of legal bid records. We ensure this by checking the well-formedness of an allocation. More importantly, an allocation should form a partition of the set of the items in the bidding data. This is a much more involved procedure wherein we must check that the allocated bundles of items cover the set of items in the bidding data and that every two allocated bundles are disjoint.

```
Definition GoodAllocation (alloc : Allocation)
        (data : LegalBiddingData) : Prop :=
    (forall is,
     get_item_set (bidding_data data) = is ->
        Cover alloc is) /\
        Contain (bidding_data data) alloc.
```

wherein `Cover` and `Contain` are both `Fixpoint` Coq definitions, which respectively check whether the bundles in `alloc` is a cover of the set of items `is` and that the bundles in `alloc` are legal elements of `bidding_data` that are relatively disjoint. Likewise, we can check and create legal allocation.

Another important definition is that of the maximum handled as follows.

```
Parameter max_allocation :
    forall (d : LegalBiddingData),
        LegalAllocation (d:=d).

Hypothesis max_allocation_sound :
  forall (d : LegalBiddingData)
   (a : LegalAllocation (d:=d)),
    (sum_price (alloc a)) <=
        sum_price (alloc (max_allocation d)).
```

This enables us to state that the maximum allocation problem has a solution *argmax* even though we do not have an explicit algorithm to calculate it. In fact, the assurance for the existence of a solution is sufficient for the proofs we are interested in as can be seen in Section 7.3 of this chapter.

Then, we have also written some handling functions to perform some useful arithmetic operations such as `sum_price`, which adds all the prices in a legal allocation or `sum_price_sub_i` that calculates the quantity in equation (7.1). The formalization of equation (7.1) is as follows.

```
Definition payment (d : LegalBiddingData)
(i : Bidder) : Z :=
  sum_price (alloc (max_sub_i d i))
   - (sum_price_sub_i (alloc (max_allocation d)) i).
```

The subtractor of this function represents the maximum allocation without bidder i, while the minuend is the original maximum allocation which subtracts the bidding record given by bidder i.

Finally, the winner's utility is defined as the function of `utility`, which formalizes equation (7.2).

```
Definition utility (d : LegalBiddingData)
     (b : Bidder) : Z :=
        (sum_price_i (alloc (max_allocation d)) b)
             - (payment d b).
```

7.3 Proof of Desirable Properties

The first property that is established for a VCG auction is *non-negative payment.*

Theorem 7. *In a VCG auction, the payment of a bidder is greater or equal to 0.*

The mathematical proof can be represented as:

$$p_i = \sum_{j=1,\, j\neq i}^{n} b_j(x^*_{-i}) - \sum_{j=1,\, j\neq i}^{n} b_j(x^*) \geq 0$$

The related Coq formalization is as:

```
Theorem payment_greater_than_or_equal_to_zero :
 forall (d : LegalBiddingData) (b : Bidder),
     payment d b >= 0.
```

This theorem states that by given a legal bidding data, the payment of a bidder from a legal allocation is greater or equal to 0. The proof of this theorem is generated by unfolding the definition of `payment`.

Proof. According to the definition of `payment`, winner's payment equals to `sum_price (alloc (max_sub_i d b)) - (sum_price_sub_i (alloc (max_allocation d)) b)`, where function `alloc` returns legal allocations from different bidding records. By having the premise that `alloc (max_sub_i d b)` is an efficient allocation without Bidder `b`, which means `(sum_price_sub_i (alloc (max_allocation d)) b) <= sum_price (alloc (max_sub_i d b))`, the conclusion can be deduced. □

The second property that we will establish is that winner's utility is non-negative.

Theorem 8. *In a VCG auction, the utility derived by a truthful agent is non-negative.*

The mathematical proof of is as:

$$u_i(v_i, p_i) = v_i - p_i = \sum_{j=1}^{n} v_j(x^*) - \sum_{j=1,\, j\neq i}^{n} v_j(x^*_{-i}) \geq 0$$

The related Coq formalization is as:

```
Theorem utility_ge_zero :
    forall (d : LegalBiddingData) (b : Bidder),
        utility d b >= 0.
```

This theorem states that given a legal bidding data, the utility of any truthful bidder is greater or equal to 0.

Proof. The first step in the proof is to unfold the definition of `utility`. Then, we perform some algebraic manipulations on this inequality, by which we get:

```
sum_price_i (alloc (max_allocation d)) b +
    sum_price_sub_i (alloc (max_allocation d)) b
        - sum_price (alloc (max_sub_i d b))
>= 0
```

The above inequality can be expressed as the following inequality:

$$v_i(x^*) + \sum_{j=1,\, j\neq i}^{n} v_j(x^*) - \sum_{j=1,\, j\neq i}^{n} v_j(x^*_{-i}) >= 0$$

Then we combine the first two terms at the left of the inequality using a lemma `sum_price_combine`. By this combination we get the following inequality:

```
sum_price (alloc (max_allocation d))
- sum_price (alloc (max_sub_i d b))
>= 0
```

This inequality is the same as the following:

$$\sum_{i=1}^{n} v_i(x^*) - \sum_{j=1,\, j\neq i}^{n} v_j(x^*_{-i}) >= 0$$

By applying the hypothesis `max_allocation_sound` which states x^* is the efficient allocation, the above inequality can be proved directly. □

The last property that has been established is *incentive compatible*, i.e. declaring their true valuation function v_i is a weakly dominant strategy for all players.

Theorem 9. *The VCG mechanism is strategy-proof, i.e. incentive compatible.*

The mathematical proof of this theorem is as follows. Suppose bidders declare bids $b'_1, ..., b'_n$, and the true valuation of bidder i is v_i. Let $x^* = f(v_i, b'_{-i})$ and $x'' = f(b'_i, b'_{-i})$. As $x^* \in \text{argmax}_{x\in\mathcal{X}} v_i(x) + \sum_{j=1;j\neq i}^{n} b'_j(x)$. Thus, for all $x \in \mathcal{X}$,

$$v_i(x^*) + \sum_{j=1;j\neq i}^{n} b'_j(x^*) \geq v_i(x) + \sum_{j=1;j\neq i}^{n} b'_j(x).$$

Therefore,

$$u_i(x^*) = v_i(x^*) - p_i(x^*) \geq v_i(x'') - p_i(x'') = u_i(x'').$$

The related Coq definition of this theorem is as the following formalization.

```
Theorem strategy_proof :
  forall d1 d2 b,
    HonestBidding d1 b
    -> DiffByBidderPrice b (bidding_data d1)
                (bidding_data d2)
    -> utility d1 b >=
        utility d2 b.
```

In this theorem, `d1` represents the bidding data which are composed of truthful bidding, while `d2` includes the untruthful bidding of agent `b`. The bidding data of other agents are the same in both `d1` and `d2`. This theorem states that the utility of truthful bidding is greater or equal to the utility of untruthful bidding strategy for bidder `b`.

Proof. The first step is to unfold the definition of `utility` and `payment`, which yields the inequality as below:

```
sum_price_i (alloc (max_allocation d1)) b -
(sum_price (alloc (max_sub_i d1 b)) -
sum_price_sub_i (alloc (max_allocation d1)) b)
>=
sum_price_i (alloc (max_allocation d2)) b -
(sum_price (alloc (max_sub_i d2 b)) -
sum_price_sub_i (alloc (max_allocation d2)) b)
```

The above inequality is the same as the following mathematical notation:

$$v_i(x^*) - (\sum_{j=1,\, j\neq i}^{n} b_j(x^*_{-i}) - \sum_{j=1,\, j\neq i}^{n} b_j(x^*)) >= v_i(x'') - (\sum_{j=1,\, j\neq i}^{n} b_j(x''_{-i}) - \sum_{j=1,\, j\neq i}^{n} b_j(x''))$$

Then we perform some algebraic manipulations followed to simplify the above inequality as:

$$v_i(x^*) + \sum_{j=1,\, j\neq i}^{n} b_j(x^*) - \sum_{j=1,\, j\neq i}^{n} b_j(x^*_{-i}) >= v_i(x'') + \sum_{j=1,\, j\neq i}^{n} b_j(x'') - \sum_{j=1,\, j\neq i}^{n} b_j(x''_{-i})$$

The corresponding Coq representation is:

```
sum_price_i (alloc (max_allocation d1)) b +
sum_price_sub_i (alloc (max_allocation d1)) b -
sum_price (alloc (max_sub_i d1 b))
>=
sum_price_i (alloc (max_allocation d2)) b +
sum_price_sub_i (alloc (max_allocation d2)) b -
sum_price (alloc (max_sub_i d2 b))
```

By applying the lemma `sum_price_combine` on the above inequality, we get the following inequality:

```
sum_price (alloc (max_allocation d1)) -
sum_price (alloc (max_sub_i d1 b))
>=
sum_price (alloc (max_allocation d2)) -
sum_price (alloc (max_sub_i d2 b))
```

The corresponding mathematical notation is:

$$\sum_{i=1}^{n} v_i(x^*) - \sum_{j=1,\, j\neq i}^{n} b_i(x^*_{-i}) >= \sum_{i=1}^{n} v_i(x'') - \sum_{j=1,\, j\neq i}^{n} b_j(x''_{-i})$$

On one hand, the assumed function `max_sub_i` returns a legal allocation on a bidding data, which has the maximum summed prices without bidder `b`. On the other hand, by removing agent `b` from `d1` and `d2`, the remaining agents' reports are the same in both `d1` and `d2`. Hence, we get the equation:

```
sum_price (alloc (max_sub_i d1 b)) = sum_price (alloc (max_sub_i d2 b))
```

which means:

$$\sum_{j=1,\, j\neq i}^{n} b_i(x^*_{-i}) = \sum_{j=1,\, j\neq i}^{n} b_j(x''_{-i})$$

Hence, we can deduce the following inequality:

```
sum_price (alloc (max_allocation d1)) -
sum_price (alloc (max_sub_i d1 b))
>=
sum_price (alloc (max_allocation d2)) -
sum_price (alloc (max_sub_i d1 b))
```

The last step is to prove:

$$\sum_{i=1}^{n} v_i(x^*) >= \sum_{i=1}^{n} v_i(x'')$$

The inequality follows because we assume x^* is an efficient allocation with respect to the true profile of valuations in the hypothesis `max_allocation_sound`. □

7.4 Related Work

Interactive theorem proving is one approach to verify the correctness of an algorithm or a network protocol meets its specification. It can also be used to verify that a mathematical statement is true. In a theorem prover, such as Coq and Isabelle/HOL [82], the mathematical content is represented as a machine understandable manner. Users use tactics to create proof in a man-machine collaborated way and the proof can be automatically checked by a proof checker. Interactive theorem proving is applied to prove and verify some pure mathematics theorems. The prime number theorem, which described the asymptotic distribution of the prime numbers among the positive integers, was formally proved and verified by using Isabelle/HOL in the work of Avigad et al. [5]. In [46], John Harrison has formalized a complex-analytic proof of the prime number theorem in the HOL Light theorem prover by developing necessary analytic machinery including Cauchy's integral formula. The first major theorem that was proved using a computer is the four color theorem [3]. The four color theorem states that any map in a plane can be colored using four colors in such a way that regions sharing a common boundary do not share the same color. In [42], a computer-checked proof of the four color theorem has been constructed in Coq. In the work of [47], HOL Light theorem prover has been used to formalize the Dirichlet's theorem, which asserts that for all pairs of positive integers a and b that are coprime, there are infinitely many primes of the form $a + nd$, where n is a non-negative integer. More theorems that have been formalized using different theorem provers can be found in [113].

Formal reasoning has been applied to theoretical economics, especially in the area of social choice theory [81] and game theory [24]. In the work of [81], Arrow's impossibility theorem has been formalized in higher-order logic which provides a valid proof

of the theorem, while the original standard and textbook proofs of Arrow's general impossibility theorem are invalid [88]. In the work of [24], by formalizing and proving several desirable properties (e.g.,VCG auctions are pairwise disjoint and prices are non-negative) of a VCG auction in Isabelle, they provides a sound specification of the mechanism. Both of the above formalizations are represented in higher order logic and set theory which indicates that mechanical theorem proving is suitable for social choice theory and game theory.

Formal verification has been used to verify an operating system kernel using Isabelle/HOL in [54]. In this work, they not only provide a full specification of the kernel, but also proof for the kernel's precise behavior. By proving a formal and machine-checked verification of an operating-system kernel, the behavior of the kernel in every possible situation can be precisely predicted. In the project of CompCert [61], Coq has been used to program and prove the correctness of a compiler from Clight (a large subset of the C programming language) to PowerPC assembly code. This approach guarantees that the safety properties proved in Clight hold in PowerPC assembly code. A machine-verified software-defined networking controller has been designed and formalized in Coq, which provides a robust guarantee of desirable behaviors [44]. This prototype is implemented by extracting Coq code into OCaml.

7.5 Summary

In this chapter, we have formalized a VCG auction, and proved some of its incentive properties by using the interactive theorem prover Coq. Coq is a proof assistant that is based on higher-order logic and type systems. It has been used to verify famous pure mathematical theorems, economic theorems and software verification. Our work provides a machine-verifiable for a mathematical theorem on the VCG combinatorial auction. Such a formal verification can be used in automated mechanism design for online auction or Web services.

Chapter 8

Concluding Remarks

In agent-mediated e-commerce systems, software agents are responsible for selling or purchasing commodities and services on behalf of their owners. Software agents with limited computational resources and bounded rationalities must make decisions to optimize their payoffs in a given transaction. In our work, we focus on online auctions since they have been established as an efficient protocol to do business. Auction mechanisms are designed to have some desirable properties such as incentive compatibility. The following requirements must be implemented in agent-mediated e-commerce systems to enable automated decision-making and services composition.

- Trading protocol such as auction mechanisms should be published in a standard language which is machine-readable.
- The desirable properties of specific auction mechanisms and formal proofs of these properties should be described.
- The proofs of desirable properties should be automatically checked by software agents.
- The interaction protocol implemented in the system should support automated communication among software agents.
- An online trading may require the composition of different services to achieve its objective. Thus, an automated communication approach should be provided to realize service composition.

In order to meet the requirements, we have proposed a certification framework in Chapter 3, and to enable software agents to automatically check desirable properties of a specific auction through a formally designed communication protocol, and then make decisions according to the result of the communication. Furthermore, we extend the communication mechanism to the area of Semantic Web Service composition and explore the verification of combinatorial auction mechanisms. In the following, we

highlight the distinguishable contributions of this thesis by analyzing related work in existing literature.

Firstly, Chapter 4 exploits that an online trading protocol, such as single item auctions, can be expressed as web services using Semantic Web Service description languages (i.e., OWL-S). The specification of an auction provides a machine understandable presentation of trading mechanisms. Because of the expressive limitation of Semantic Web Service languages, we therefore rely on the technique of interactive theorem proving to construct proofs of properties for a trading mechanism. By translating the auction specification from OWL-S to an imperative program which is defined within an interactive theorem prover named Coq, we then apply Hoare Logic to prove the desirable properties of the original specification. There is some work related to the translation of service descriptions. Brogi et al. [22] present a translator that translating OWL-S process description in to Petri nets, so that the tools available for Petri nets can be used to analyze OWL-S services. Ankolekar et al. [1] describe a mapping from OWL-S process model into equivalent PROMELA statements that can be evaluated by a model checker SPIN. Then numerous properties (e.g. safety and liveness properties) of the OWL-S process model can be verified using SPIN. Feng and Kirchberg [37] propose an approach which translates an OWL-S process model into process algebra, and then use a model checker to check the properties of the process. In this thesis, we have defined the semantics of a subset of OWL-S ontology within the theorem prover Coq and then provided the proofs of desirable properties.

Secondly, it shows that the Proof-Carrying Code (PCC) paradigm can be implemented to make an auctioneer publish an auction specification and proofs of desirable properties of the auction, and then enable buyer agents to automatically check the correctness of these proofs using a proof checker in Section 3.2. PCC paradigm has been used by Leroy [62] to certify a compiler so that the safety properties proved on the source code also hold for the compiled code. Love et al. [68] present a framework based on PCC paradigm to enable consumer to verify security-related properties of hardware intellectual property. In this thesis, the PCC paradigm has been implemented using Coq, and then we have integrated PCC paradigm within an inquiry dialogue game that supports automatically communication between buyer agents and auctioneers in Chapter 5. Dialogue games have been used in multi-agent systems to support effective interaction among agents. Black and Hunter [17, 18] propose an inquiry dialogue system that not only provides protocols but also a strategy to generate dialogues. Our inquiry dialogue game is designed to follow the design of Black and Hunter [17, 18]. The difference is that both formal proofs and informal evidence can be used as arguments in our inquiry dialogue game. By using dialogue games, buyer agents can make their decisions about whether or not to participate in an auction, and even determine their bidding strategies according to the results of the communication.

Thirdly, dialogue games are extended to support the composition of Semantic Web Service in Chapter 6. On the one hand, dialogue games have been utilized to resolve conflicts and manage internal and external influences in multi-agent systems [52]. Different kinds of dialogues are proposed to achieve variety objectives. For instance, in a persuasion dialogue, participants aim to resolve a difference of opinion. Persuasion dialogues have been applied in various areas, such as artificial intelligence and law [13] and human-robot interaction [94]. On the other hand, automated composition of Web services at the process level is a difficult challenge as the nondeterministic and partially observable behaviors of component services. Bertoli et al. [16] formalize the automated process-level composition of services as a planning problem wherein services are represented as finite state automata. In this thesis, we have introduced dialogue games into Semantic Web Services and proposed Composite Dialogue Service Automata to synthesize services. Dialogue games extend the original client-server communication paradigm of Semantic Web Services to more complex manners. For example, the service provider can not only persuade buyer agents to choose its service but also composite services that fulfill the requirements of buyers by using dialogue games.

Finally, Chapter 7 shows the exploration of formalization and verification desirable properties of combinatorial auctions. Combinatorial auctions allow bidders bid on combinations of items in the presence of substitutes and complements. VCG mechanism is a classical combinatorial auction that satisfies some desirable properties, such as incentive compatibility. However, the VCG mechanism is computationally intractable. Hence, we introduce hypothesis to state the property of efficient allocations in a VCG mechanism. The VCG mechanism is specified within Coq directly, due to OWL-S has limited expressiveness of service description, which corresponds to its underlying Web Ontology Language (OWL). We present the proofs of three desirable properties of the VCG mechanism: the payment is non-negative, the utility of truthful bidder is non-negative, and incentive compatibility. There is some work related to our approach. In the work of Tadjouddine and Guerin [100], model checking is used to formalize and verify properties of a two player Vickrey auction and a quantity restricted multi-unit auction using the VCG mechanism. Brânzei et al. [21] construct a verification algorithm to verify a mechanism is strategy-proof. The focus of their work is that the agents are able to efficiently verify the truthfulness of a mechanism. Caminati et al. [24] use Isabelle to formally specify the combinatorial VCG auction and prove a set of desirable properties (e.g. VCG allocations are pairwise disjoint). We extend their work by verifying the incentive compatibility property of VCG mechanisms.

The work in this thesis can be extended in a number of promising directions for future work.

- *Formalization and verification of game theoretic properties.* We consider formalizing and proving more game theoretic properties of different mechanisms. The

well-known game theoretic properties include *pareto efficiency* [116] in which resources are allocated in the most efficient manner, *false-name-proof* [106, 107] where no agent has incentive to use fake accounts, *Nash equilibrium* [110] in which no single player can gain higher payoff by changing his strategy unilaterally, and security properties [20] such as *fairness* which states that all agents are treated equally.

- *Sound specification and implementation of programs.* The specification of services in this thesis is expressed in the language of OWL-S. The grounding OWL-S Services with WSDL or Java can be implemented using the OWL-S API [91]. However, we still face the problem that whether the specification is faithfully implemented or not. In the work of [24], Isabelle/HOL is used to generate verified executable code directly from its specification to a functional programming language. This requires the specification written in a constructive logic manner. Coq also has an extraction mechanism to generate certified programs to functional programming languages, such as Ocaml, Haskell or Scheme, out of Coq programs [63]. The faithful implementation of a specification can be studied to ensure that a program is consistent with its design.

- *Consistency checking of specifications.* One of the desiderata for specifications is that a specification should be consistent and should not contradict itself. Specifications are usually written in different languages. In the work of [92], a tableaux reasoner is designed to check the consistency of an OWL-DL ontology which is based on description logic. The consistency checking task of first-order formula is defined as a satisfiability checking task [19]. However, the satisfiability problem for first-order logic is undecidable [43], which makes it a hard problem to check the consistency of a specification. We would like to explore the method to check the consistency of specifications.

- *Interoperable multi-agent systems.* In our work, all of the agents are developed within JADE and a common ontology is shared among them. However, software agents can be developed using different platforms or languages in an open system. Therefore, we should design mechanism to handle the interoperable problem between heterogeneous software agents.

- *Argumentation-based dialogue games.* Argumentation can be used in dialogues to represent the reasonable conclusions by constructing pro and con arguments. In the original of Dung [36], an abstract argumentation framework has been proposed to formalize relations between arguments, different semantics are defined to solve the inherent conflicts between statements. Examples to compute argumentation semantics can be found in the work of Liao [64]. In the work of [67], a

division-based approach has been proposed to cope with the problem of changing arguments and their attack relations in dynamic argumentation systems. In [66], the authors have proposed methods to efficiently compute the partial semantics of argumentation using answer-set programming. We plan to integrate argumentation into dialogue games to support conversations among agents.

Bibliography

[1] Anupriya Ankolekar, Massimo Paolucci, and Katia Sycara. Towards a formal verification of owl-s process models. In *International semantic web conference*, pages 37–51. Springer, 2005.

[2] Andrew W Appel. Foundational proof-carrying code. In *Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on*, pages 247–256. IEEE, 2001.

[3] Kenneth I Appel and Wolfgang Haken. *Every planar map is four colorable*, volume 98. American mathematical society Providence, 1989.

[4] John Langshaw Austin. *How to do things with words*, volume 1955. Oxford university press, 1975.

[5] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Transactions on Computational Logic (TOCL)*, 9(1):2, 2007.

[6] Wei Bai, Emmanuel Tadjouddine, and Terry Payne. A dialectical approach to enable decision making in online trading. In *Multi-Agent Systems and Agreement Technologies*, pages 203–218. Springer, 2015.

[7] Wei Bai and Emmanuel M Tadjouddine. Automated program translation in certifying online auctions. In *ETAPS/VPT 2015, April 11-18, London, UK.*, 2015.

[8] Wei Bai, Emmanuel M Tadjouddine, and Yu Guo. Enabling automatic certification of online auctions. In *Proceedings 11th International Workshop on Formal Engineering Approaches to Software Components and Architectures*. 2014.

[9] Wei Bai, Emmanuel M Tadjouddine, Terry R Payne, and Sheng-Uei Guan. A proof-carrying code approach to certificate auction mechanisms. In *Formal Aspects of Component Software*, pages 23–40. Springer, 2013.

[10] Christopher JO Baker and Kei-Hoi Cheung. *Semantic web: Revolutionizing knowledge discovery in the life sciences*. Springer, 2007.

[11] Fabio Bellifemine, Federico Bergenti, Giovanni Caire, and Agostino Poggi. Jade – a java agent development framework. In *Multi-Agent Programming*, pages 125–147. Springer, 2005.

[12] F.L. Bellifemine, G. Caire, and D. Greenwood. Developing multi-agent systems with jade (wiley series in agent technology). 2007.

[13] Trevor J. M. Bench-Capon, T Geldard, and Paul H. Leng. A method for the computational modelling of dialectical argument with dialogue games. *Artificial Intelligence and Law*, 8(2-3):233–254, 2000.

[14] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN Notices*, volume 39, pages 14–25. ACM, 2004.

[15] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[16] Piergiorgio Bertoli, Marco Pistore, and Paolo Traverso. Automated composition of web services via planning in asynchronous domains. *Artificial Intelligence*, 174(3):316–361, 2010.

[17] Elizabeth Black and Anthony Hunter. A generative inquiry dialogue system. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 241. ACM, 2007.

[18] Elizabeth Black and Anthony Hunter. An inquiry dialogue system. *Autonomous Agents and Multi-Agent Systems*, 19(2):173–209, 2009.

[19] Patrick Blackburn and Johan Bos. Representation and inference for natural language. *A first course in computational semantics. CSLI*, 2005.

[20] Colin Boyd and Wenbo Mao. *Security issues for electronic auctions*. Hewlett-Packard Laboratories Bristol (UK), 2000.

[21] Simina Brânzei and Ariel D Procaccia. Verifiably truthful mechanisms. In *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science*, pages 297–306. ACM, 2015.

[22] Antonio Brogi, Sara Corfini, and Stefano Iardella. From owl-s descriptions to petri nets. In *International Conference on Service-Oriented Computing*, pages 427–438. Springer, 2007.

[23] Marco B. Caminati, Manfred Kerber, Christoph Lange, and Colin Rowat. Proving soundness of combinatorial vickrey auctions and generating verified executable code. *CoRR*, abs/1308.1779, 2013.

[24] Marco B. Caminati, Manfred Kerber, Christoph Lange, and Colin Rowat. Sound auction specification and implementation. In *Proceedings of the Sixteenth ACM Conference on Economics and Computation*, EC '15, pages 547–564, New York, NY, USA, 2015. ACM.

[25] Hans Chalupsky, Tim Finin, Rich Fritzson, Don McKay, Stu Shapiro, and Gio Wiederhold. An overview of kqml: A knowledge query and manipulation language. *KQML Advisory Group. April*, 1992.

[26] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (wsdl) 1.1, 2001.

[27] Edward H Clarke. Multipart pricing of public goods. *Public choice*, 11(1):17–33, 1971.

[28] Philip R Cohen and C Raymond Perrault. Elements of a plan-based theory of speech acts*. *Cognitive science*, 3(3):177–212, 1979.

[29] Peter Cramton, Yoav Shoham, and Richard Steinberg. Combinatorial auctions. 2006.

[30] A-S Dadzie, R Bhagdev, A Chakravarthy, S Chapman, J Iria, V Lanfranchi, J Magalhães, D Petrelli, and F Ciravegna. Applying semantic web technologies to knowledge sharing in aerospace engineering. *Journal of Intelligent Manufacturing*, 20(5):611–623, 2009.

[31] Partha Dasgupta and Eric Maskin. Efficient auctions. *Quarterly Journal of Economics*, pages 341–388, 2000.

[32] Jos De Bruijn, Christoph Bussler, John Domingue, Dieter Fensel, Martin Hepp, M Kifer, B König-Ries, J Kopecky, R Lara, E Oren, et al. Web service modeling ontology (wsmo). *Interface*, 5:1, 2006.

[33] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.

[34] G. Dowek, A. Felty, H. Herbelin, G. Huet, B. Werner, C. Paulin-Mohring, et al. The coq proof assistant user's guide: Version 5.6. 1991.

[35] Xutao Du, Chunxiao Xing, Lizhu Zhou, and Zhoujun Li. Nested web service interface control flow automata. In *Next Generation Web Services Practices, 2008. NWESP'08. 4th International Conference on*, pages 129–136. IEEE, 2008.

[36] Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial intelligence*, 77(2):321–357, 1995.

[37] Yuzhang Feng and Markus Kirchberg. Verifying owl-s service process models. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 307–314. IEEE, 2011.

[38] ACL Fipa. Fipa acl message structure specification. *Foundation for Intelligent Physical Agents, http://www. fipa. org/specs/fipa00061/SC00061G. html (30.6. 2004)*, 2002.

[39] Nicoletta Fornara and Marco Colombetti. Ontology and time evolution of obligations and prohibitions using semantic web technology. In *Declarative Agent Languages and Technologies VII*, pages 101–118. Springer, 2010.

[40] Alejandro J García and Guillermo R Simari. Defeasible logic programming: An argumentative approach. *Theory and practice of logic programming*, 4(1+ 2):95–138, 2004.

[41] G. Gonthier. The four colour theorem: Engineering of a formal proof. *Computer Mathematics*, pages 333–333, 2008.

[42] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[43] Erich Grädel. Decidable fragments of first-order and fixed-point logic. *KWLCS03*, 2003.

[44] Arjun Guha, Mark Reitblatt, and Nate Foster. Machine-verified network controllers. *SIGPLAN Not.*, 48(6):483–494, June 2013.

[45] Steve Harris, Andy Seaborne, and Eric Prudhommeaux. Sparql 1.1 query language. *W3C Recommendation*, 21, 2013.

[46] John Harrison. Formalizing an analytic proof of the prime number theorem. *Journal of Automated Reasoning*, 43(3):243–261, 2009.

[47] John Harrison. A formalized proof of dirichlet's theorem on primes in arithmetic progression. *Journal of Formalized Reasoning*, 2(1):63–83, 2010.

[48] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[49] Ian Horrocks, Bijan Parsia, Peter Patel-Schneider, and James Hendler. Semantic web architecture: Stack or two towers? In *Principles and practice of semantic web reasoning*, pages 37–41. Springer, 2005.

[50] Ian Horrocks, Peter F Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, Mike Dean, et al. Swrl: A semantic web rule language combining owl and ruleml. *W3C Member submission*, 21:79, 2004.

[51] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[52] Nishan C Karunatillake, Nicholas R Jennings, Iyad Rahwan, and Peter McBurney. Dialogue games that agents play within a society. *Artificial intelligence*, 173(9):935–981, 2009.

[53] Michael Kifer and Harold Boley. Rif overview. *W3C Working Group Note*, 2010.

[54] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, 2009.

[55] Graham Klyne and Jeremy J Carroll. Resource description framework (rdf): Concepts and abstract syntax. 2006.

[56] Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. A description logic primer. *arXiv preprint arXiv:1201.4089*, 2012.

[57] Yannis Labrou and Tim Finin. A proposal for a new kqml specification. Technical report, Technical Report Technical Report TR-CS-97-03, University of Maryland Baltimore County, 1997.

[58] Yannis Labrou and Tim Finin. Semantics and conversations for an agent communication language. *Readings in agents*, pages 235–242, 1998.

[59] Michal Laclavik, Zoltan Balogh, Marian Babik, and Ladislav Hluchỳ. Agentowl: Semantic knowledge model and agent architecture. *Computing and Informatics*, 25(5):421–439, 2012.

[60] A. Lapets, A. Levin, and D. Parkes. A typed language for truthful one-dimensional mechanism design. Technical report, Boston University Computer Science Department, 2008.

[61] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[62] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM SIGPLAN Notices*, volume 41, pages 42–54. ACM, 2006.

[63] Pierre Letouzey. Extraction in coq: An overview. In *Logic and Theory of Algorithms*, pages 359–369. Springer, 2008.

[64] Beishui Liao. *Efficient Computation of Argumentation Semantics*. Academic Press, 2013.

[65] Beishui Liao and Ji Gao. A model of multi-agent system based on policies and contracts. In *Multi-Agent Systems and Applications IV*, pages 62–71. Springer, 2005.

[66] Beishui Liao and Huaxin Huang. Partial semantics of argumentation: basic properties and empirical results. *Journal of Logic and Computation*, 23(3):541–562, 2013.

[67] Beishui Liao, Li Jin, and Robert C Koons. Dynamics of argumentation systems: A division-based method. *Artificial Intelligence*, 175(11):1790–1814, 2011.

[68] Eric Love, Yier Jin, and Yiorgos Makris. Proof-carrying hardware intellectual property: A pathway to trusted module acquisition. *IEEE Transactions on Information Forensics and Security*, 7(1):25–40, 2012.

[69] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, et al. Owl-s: Semantic markup for web services. *W3C member submission*, 22:2007–04, 2004.

[70] David Martin and John Domingue. Semantic web services, part 1. *Intelligent Systems, IEEE*, 22(5):12–17, 2007.

[71] Peter McBurney and Simon Parsons. Games that agents play: A formal framework for dialogues between autonomous agents. *Journal of Logic, Language and Information*, 11(3):315–334, 2002.

[72] Peter McBurney and Simon Parsons. Locutions for argumentation in agent interaction protocols. In *Agent Communication*, pages 209–225. Springer, 2005.

[73] Peter McBurney and Simon Parsons. Dialogue games for agent argumentation. In *Argumentation in artificial intelligence*, pages 261–280. Springer, 2009.

[74] Peter McBurney, Simon Parsons, and Michael Wooldridge. Desiderata for agent argumentation protocols. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, pages 402–409. ACM, 2002.

[75] Deborah L McGuinness, Frank Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(2004-03):10, 2004.

[76] Sheila A McIlraith, Tran Cao Son, and Honglei Zeng. Semantic web services. *Intelligent Systems, IEEE*, 16(2):46–53, 2001.

[77] Huaikou Miao, Tao He, and Liping Li. Formal semantics of owl-s with f-logic. In *Computer and Information Science 2009*, pages 105–117. Springer, 2009.

[78] Boris Motik, Peter F Patel-Schneider, Bijan Parsia, Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, et al. Owl 2 web ontology language: Structural specification and functional-style syntax. *W3C recommendation*, 27(65):159, 2009.

[79] G.C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119. ACM, 1997.

[80] George C Necula. *Proof-carrying code. design and implementation.* Springer, 2002.

[81] Tobias Nipkow. Social choice theory in hol. *Journal of Automated Reasoning*, 43(3):289–304, 2009.

[82] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic.* 2002.

[83] Traverso Paolo and Pistore Marco. Automated composition of semantic web services into executable processes. In *International Semantic Web Conference - ISWC*, 2004.

[84] Terence John Parr and Russell W. Quong. ANTLR: A Predicated- LL(k) Parser Generator. *Software - Practice and Experience*, 25:789–810, 1995.

[85] Terry R Payne and Valentina Tamma. Negotiating over ontological correspondences with asymmetric and incomplete knowledge. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, pages 517–524. International Foundation for Autonomous Agents and Multiagent Systems, 2014.

[86] Domenico Redavid, Luigi Iannone, Terry Payne, and Giovanni Semeraro. Owl-s atomic services composition with swrl rules. In *Foundations of Intelligent Systems*, pages 605–611. Springer, 2008.

[87] Michael H Rothkopf, Aleksandar Pekeč, and Ronald M Harstad. Computationally manageable combinational auctions. *Management science*, 44(8):1131–1147, 1998.

[88] R. Routley. Repairing proofs of arrow's general impossibility theorem and enlarging the scope of the theorem. *Notre Dame J. Formal Logic*, 20(4):879–890, 10 1979.

[89] John R Searle. *Speech acts: An essay in the philosophy of language*, volume 626. Cambridge university press, 1969.

[90] Yoav Shoham and Kevin Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.

[91] Evren Sirin and Bijan Parsia. The owl-s java api. In *Poster) In Proceedings of the Third International Semantic Web Conference (ISWC2004), Hiroshima, Japan*, 2004.

[92] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

[93] Sergej Sizov. What makes you think that? the semantic web's proof layer. *Intelligent Systems, IEEE*, 22(6):94–99, 2007.

[94] Elizabeth Sklar, Mohammad Q Azhar, Todd Flyr, and Simon Parsons. A case for argumentation to enable human-robot collaboration. *Proceedings of Autonomous Agents and Multiagent Systems (AAMAS), St Paul, MN, USA*, 2013.

[95] Jianwen Su, Tevfik Bultan, Xiang Fu, and Xiangpeng Zhao. Towards a theory of web service choreographies. In *Web Services and Formal Methods*, pages 1–16. Springer, 2007.

[96] Julien Subercaze and Pierre Maret. Programming semantic agent for distributed knowledge management. In *Semantic Agent Systems*, pages 47–65. Springer, 2011.

[97] E. Tadjouddine and F. Guerin. Verifying dominant strategy equilibria in auctions. *Multi-Agent Systems and Applications V*, pages 288–297, 2007.

[98] E.M. Tadjouddine, F. Guerin, and W. Vasconcelos. Abstractions for model-checking game-theoretic properties of auctions. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3*, pages 1613–1616. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[99] Emmanuel M. Tadjouddine. Computational complexity of some intelligent computing systems. *International Journal of Intelligent Computing and Cyberneticsl*, 4(2):144 – 159, 2011.

[100] Emmanuel M Tadjouddine and Frank Guerin. Verifying dominant strategy equilibria in auctions. In *Multi-Agent Systems and Applications V*, pages 288–297. Springer, 2007.

[101] Emmanuel M Tadjouddine, Frank Guerin, and Wamberto Vasconcelos. Abstracting and verifying strategy-proofness for auction mechanisms. In *Declarative Agent Languages and Technologies VI*, pages 197–214. Springer, 2009.

[102] Valentina Tamma, Steve Phelps, Ian Dickinson, and Michael Wooldridge. Ontologies for supporting negotiation in e-commerce. *Engineering applications of artificial intelligence*, 18(2):223–236, 2005.

[103] Valentina Tamma, Michael Wooldridge, Ian Blacoe, and Ian Dickinson. An ontology based approach to automated negotiation. In *Agent-Mediated Electronic Commerce IV. Designing Mechanisms and Systems*, pages 219–237. Springer, 2002.

[104] Moshe Tennenholtz. Some tractable combinatorial auctions. In *AAAI/IAAI*, pages 98–103, 2000.

[105] The Coq Development Team. The coq proof assistant reference manual: Version 8.4, 2012.

[106] Taiki Todo and Vincent Conitzer. False-name-proof matching. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 311–318. International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[107] Taiki Todo, Takayuki Mouri, Atsushi Iwasaki, and Makoto Yokoo. False-name-proofness in online mechanisms. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 753–762. International Foundation for Autonomous Agents and Multiagent Systems, 2012.

[108] Francesca Toni, Mary Grammatikou, Stella Kafetzoglou, Leonidas Lymberopoulos, Symeon Papavassileiou, Dorian Gaertner, Maxime Morge, Stefano Bromuri, Jarred McGinnis, Kostas Stathis, et al. The argugrid platform: an overview. In *Grid Economics and Business Models*, pages 217–225. Springer, 2008.

[109] Paolo Torroni, Marco Gavanelli, and Federico Chesani. Argumentation in the semantic web. *Intelligent Systems, IEEE*, 22(6):66–74, 2007.

[110] Hal R Varian. Position auctions. *international Journal of industrial Organization*, 25(6):1163–1178, 2007.

[111] Douglas N Walton and Erik CW Krabbe. Commitment in dialogue. *Basic Concepts of Interpersonal Reasoning. State University of New York Press, Albany, NY*, 35, 1995.

[112] Douglas Neil Walton and Erik CW Krabbe. *Commitment in dialogue: Basic concepts of interpersonal reasoning*. SUNY press, 1995.

[113] Freek Wiedijk. Formalizing 100 theorems.

[114] Michael Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.

[115] Michael Wooldridge and Nicholas R Jennings. Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(02):115–152, 1995.

[116] Makoto Yokoo, Yuko Sakurai, and Shigeo Matsubara. The effect of false-name bids in combinatorial auctions: New fraud in internet auctions. *Games and Economic Behavior*, 46(1):174–188, 2004.

[117] Fan Zhang, Ji Gao, and Bei-shui Liao. Policy-driven model for autonomic management of web services using mas. In *Machine Learning and Cybernetics, 2006 International Conference on*, pages 34–39. IEEE, 2006.

[118] Zhichao Zhang, Weiping Li, Zhonghai Wu, and Wei Tan. Towards an automata-based semantic web services composition method in context-aware environment. In *Services Computing (SCC), 2012 IEEE Ninth International Conference on*, pages 320–327. IEEE, 2012.

[119] Youyong Zou, Tim Finin, Li Ding, Harry Chen, and Rong Pan. Using semantic web technology in multi-agent systems: a case study in the taga trading agent environment. In *Proceedings of the 5th international conference on Electronic commerce*, pages 95–101. ACM, 2003.