A Proof-Carrying Code Approach to Certificate Auction Mechanisms

W. Bai[1,2], E. M. Tadjouddine[2], T. R. Payne[1], and S.U. Guan[2]

[1] Department of Computer Science
University of Liverpool
England, UK
{Wei.Bai,T.R.Payne}@liverpool.ac.uk
[2] Department of Computer Science and Software Engineering
Xi'an Jiaotong-Liverpool University
SIP, Suzhou, China
{Emmanuel.Tadjouddine,Steven.Guan}@xjtlu.edu.cn

Abstract. Whilst it can be highly desirable for software agents to engage in auctions, they are normally restricted to trading within known auctions, due to the complexity and heterogeneity of the auction rules within an e-commerce system. To allow for agents to deal with previously unseen protocols, we present a proof-carrying code approach using Coq wherein auction protocols can be specified and desirable properties be proven. This enables software agents to automatically certify claimed auction properties and assist them in their decision-making. We have illustrated our approach by specifying both the English and Vickrey auctions; have formalized different bidding strategies for agents; have certified that up to the valuation is the optimal strategy in English auction and truthful bidding is the optimal strategy in Vickrey auction for all agents. The formalization and certification are based on inductive definitions and constructions from within Coq. This work contributes to solving the problem of open societies of software agents moving between different institutions and seeking to make optimal decisions and will benefit those engaged in agent-mediated e-commerce.

Keywords: Coq, proof-carrying code, certification, e-commerce, software agents.

1 Introduction

One of the major challenges in developing agents that are capable of rational decision making within open, heterogeneous environments, is that of comprehending the rules and social norms that govern the behavior of new institutions. Although much work has addressed interoperability at the communication level (with agent communication languages such as FIPA-ACL, and RDF to underpin recent developments within the Semantic Web [1]) 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 bid strategically 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 based on Vickrey auctions. Within an open and dynamic environment (such as e-commerce), agents might encounter a variety of auction houses, that could form part of an agent mediated e-commerce scenario. It is therefore important for the agent to be able to acquire a deeper model of the marketplaces that they could engage in (other than simply relying in simple classifications) so that they can 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 that can be relevant to privacy, security, or economics. This paper focuses on the economic properties, by looking at specifying and verifying game-theoretic properties for single item online auctions. An important game-theoretic property is *strategy-proofness* namely, the existence of a dominant strategy for the players meaning a strategy that is optimal regardless of the game configuration. For example, truthful bidding can be the dominant strategy in certain auction settings. The aim of this paper is to present an approach to help agents to automatically verify desirable properties in online auctions. To this end, we rely on the proof-carrying code (PCC) paradigm [2] to allow for:

– the auctioneer to publish the auction mechanism along with the proofs of desirable properties in a machine readable formalism,
– the potential buyer agent to read the published protocol, make sense of it, and at will, check the proof of a given property by using a simple trusted checker, which makes the automatic checking procedure computationally reasonable.

Our current work focuses on expressing the mechanism and game-theoretic proofs in a machine checkable formalism. We have used Coq [3], an interactive theorem prover based on inductive definitions and construction wherein the formalizations of English and Vickrey auctions are carried out. Then, different bidding strategies are specified followed by the proofs of a dominant strategy for each bidder.

Previous efforts have explored the use of automatic checking of auction properties. The strategy-proofness property was checked using model checking in [4, 5] but the related computational complexity can be exponential [6]. To handle the computational limits of exhaustive model checking, two property-preserving abstractions are proposed. One is the classical program slicing technique [7]. The other is abstract interpretation [5]. In [8], a distributed computer system infrastructure with a rationality authority that allows for safe consultations among parties is presented. A rationality authority includes the game inventor, participating agents and verifiers, which provide verification services. Game inventors advise the agents about actions and their optimality. Verifiers send their verification procedures to the agents. A typed language which allows for automatic verification that an allocation algorithm is monotonic and therefore truthful is introduced in [9]. Then, a more general-purpose programming language is defined to capture a collection of basic truthful allocation algorithms. This is similar to our current approach as we rely upon the proof-carrying code paradigm and Coq to allow software agents to achieve reasonable automatic checking of game properties.

Moreover, interactive theorem proving is used to express the proof of desirable properties in a machine-checkable manner. There are two advantages in using an interactive theorem prover [10]. One is that the specification of the desirable properties can be precisely described by the designer. The other is that the proof of a property is machine-checkable. We use the interactive theorem prover Coq because it has been developed for more than twenty years [11] 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 model and verify sequential programs [12] and concurrent programs [13]. In [14], Coq was used to develop and certify a compiler. A fully computer-checked proof of the Four Colour Theorem was created in [15]. In [16], a Coq-formalised proof that all non-cooperative, sequential games have a Nash equilibrium point is presented.

This paper is organized as follows. Section 2 describes our certification framework and the scenario of single item auctions. Section 3 describes the formalization of auction mechanisms followed by proofs of desirable properties in Section 4. Section 5 discusses the evaluation of our approach and Section 6 concludes.

2 Our Certification Framework

The ability for heterogeneous software agents to interoperate between different and open auction houses raise two main questions: how to get agents to operate on previously unseen protocols and how to get agents to automatically check desirable properties that are central to their decision making. In order to solve this difficult problem, we start by looking at models or scenarios allowing us to use a divide-and-conquer paradigm for an incremental solution. A brief overview of our scenario can be stated as follows. Online protocols can be described using some web-based description language; the resulting description is abstracted into Coq specifications that are used to provide machine-checkable proofs of desirable properties for the protocol at hand. Such a Coq specification can be turned back into the original web description so as to be read, understood, and checked by a software agent. Such mappings back and forth can be carried out using *abstract interpretation* [17]. Abstract interpretation enables us to analyze the behaviors of a computer system by *safely* approximating its concrete semantics into an abstract one involving a smaller set of values. Note that by safe approximations, we mean approximations that are at least sound allowing us to transpose properties that are true in the abstract domain into the concrete one. For the abstraction, from a web-based description of an auction, we can build up a Coq-based specification of that auction known as the abstract mapping so that desirable properties can be proved from within the Coq system. This abstraction approach can solve the problem of heterogeneity of different auction houses by providing a uniform and formalized format of protocols to software agents.

An abstract interpretation is defined as a sound approximated program semantics obtained from a concrete one by replacing the concrete domain of computation and its concrete semantic operations with an abstract domain and corresponding abstract semantic operations. An abstraction is sound if any property that holds in the abstracted program holds also in the concrete program. In the architecture of abstract interpretation, the abstract domain can be concretized back into the concrete domain which means that the concretized abstract context includes the concrete context. The success of abstraction and concretization leads to the correctness of interpretation. Based on abstract interpretation, program transformation frameworks were proposed in [18]. Figure 1 illustrates our

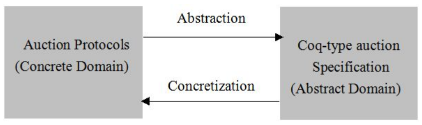

Fig. 1. Framework of Abstract Interpretation

use of the abstract interpretation framework. Once a web based auction protocol is abstracted into Coq, desirable properties can be formally proven and the resulting proof is machine-checkable and therefore verifiable by software agents.

In this work, we focus on the verification procedures for some desirable properties of auction mechanisms, which can be specified in Coq. The Coq system is based on a typed lambda calculus [19], which can be taken as a glue specification language into and from which any auction mechanism can be mapped to.

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 may be constrained in their reasoning. On one hand, it is difficult for a software agent 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 humans might. On the other hand, if the specification of auction protocols and proofs are published in a machine-readable formalism, then automatic checking by software agents can be facilitated and the computational complexity will be reduced. For that purpose, we have relied upon the Proof-Carrying-Code (PCC) ideas since 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 so that it can be checked by any agent willing to join the auction house.

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) [20] provides us with stronger semantic foundations to PCC by generating verification conditions directly from the operational semantics. Figure 2 illustrates our framework that uses FPCC to certify auction properties. At the producer or auctioneer's 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. After the proof checker is installed to the consumer side, the buyer can now perform all verifications of claimed properties of the auction before deciding to join and with which bidding strategy.

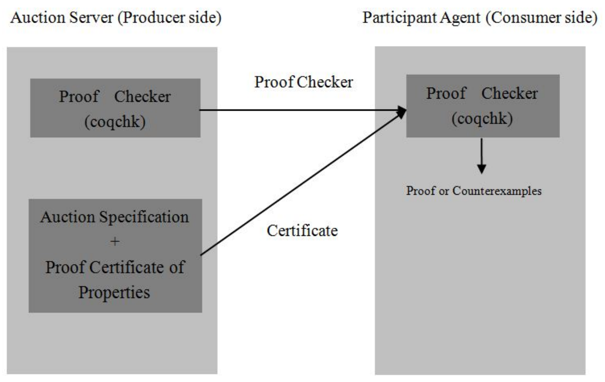
Auction Server (Producer side)
Participant Agent (Consumer side)
Proof Checker (coqchk)
Proof Checker
Proof Checker (coqchk)
Proof or Counterexamples
Auction Specification + Proof Certificate of Properties
Certificate

Fig. 2. Applying FPCC to certify Auction properties

We have implemented this FPCC framework from within the Coq system. In our current implementation, we have considered a one-to-many scenario. A single item is allocated using an online auction house. Various buyer agents can enter or leave this house at will, make sense of its mechanism along with some recommended strategies and their associated proofs. Such a recommendation can be for example, truthful bidding is the dominant or optimal strategy for a buyer agent. We then showed how such a desirable property can be proved using two examples of a single item auction: the English and Vickrey auctions. In the remainder of this paper, we basically show how to specify such auctions and its possible strategy-proofness property and how to prove it within Coq. The specifications and proofs are split into different Coq files [3].

3 Formalization of Auction Mechanisms within Coq

In this section, we define the framework to specify single item auctions. Then, the English and Vickrey auctions are specified respectively. For simplicity, we assume

[3] Our Coq code is available upon request.

no agents submit the same bid. To specify the English and Vickrey auctions, we start by a framework that is used to describe a single item auction within Coq. Coq uses the keyword `Definition` to define a variable or a function. The keyword `Inductive` is used to provide inductive definitions and `Fixpoint` can be used to define recursive functions in Coq. Coq provides library to define data types, such as the type `nat` which represents natural numbers, the type `Z` which represents integers and the type `bool` of booleans. When defining a function, pattern-matching construct `match ... with` can be used to describe different cases. Coq also provides functions to compare different numbers. For example, function `Z_gt_dec` can be used to compare two integers and decide whether one integer is greater than the other one or not.

3.1 Specifying Single Item Auction

To specify a single item auction in Coq, we define the following objects as types: `Agents, Bid, Utility` to represent respectively the set of agents, their bids, and their utilities. Note that `Bid` is declared as an integer to simplify the calculation of the utility function but can be viewed as a natural number.

```
Definition Agents: = nat.
Definition Bid: = Z.
Definition Utility: = Z.
```

We then describe an inductive relation `aRb` binding agents with their bids and provide two functions `Agent_aRb` and `Bid_aRb` that return respectively the agent and the bid for a given relation.

```
Inductive aRb : Type :=
     Binding : Agents -> Bid -> aRb.

Definition Agent_aRb (r:aRb):Agents :=
   match r with
   | Binding a b => a
   end.

Definition Bid_aRb (r:aRb):Bid :=
   match r with
   | Binding a b => b
   end.
```

To enable us reasoning on the agents' utilities, we define a relation `aRu` binding agents to their utilities and a handle function `Utility_aRu` to extract the utility of a given agent.

```
Inductive aRu : Type :=
     AUtility : Agents -> Utility -> aRu.

Definition Utility_aRu (au:aRu):Utility :=
   match au with
   | AUtility a u => u
   end.
```

To eliminate negative bidding, we define a function `TestBid` allowing us to set any bid that is smaller than zero to zero.

```
Definition TestBid (b:Z):Bid :=
  match Z_gt_dec b 0 with
  |  left _ => b
  |  right _ => 0
  end.
```

To enable agents to decide whether to bid or not, we have defined a relation `flag` binding an agent with a boolean value indicating the choice of this agent. If the value is true, then the agent wants to bid, otherwise the agent gives up bidding in the current round. Agents can set their choices based on their bidding strategies by using the function `Set_flag`.

```
Inductive flag : Type :=
   Choice : Agents -> bool -> flag.

Definition Set_flag (a:Agents)(b:bool) : flag :=
   match b with
   | true => Choice a b
   | false => Choice a b
  end.
```

With the help of `flag`, we can build up the state of the auction by a fixpoint definition of the function `AuctionState`. We use the function `Bool_flag` to get the boolean value associated to each agent. We then store all the flag values into a *List* structure `flaglist`, which is the input to the function `AuctionState`. If `AuctionState` returns true, then the auction will continue, otherwise it stops.

```
Definition Bool_flag (f:flag) : bool :=
   match f with
   | Choice a b => b
   end.

Inductive flaglist : Type :=
  | nil : flaglist
  | cons : flag -> flaglist -> flaglist.

Fixpoint AuctionState (fl:flaglist) : bool :=
   match fl with
   | nil => false
   | cons h nil => match (Bool_flag h) with
                 | false => false
                 | true => true
                 end
   | cons h t => match (Bool_flag h) with
                 | false => AuctionState t
                 | true => true
                 end
   end.
```

Next, we will illustrate our single item auction specification by using the English and Vickrey auctions to show how to specify agents' strategies and how a given strategy profile can be shown to be a dominant strategy equilibrium.

3.2 The English Auction Case

In the English auction, we consider two strategies: First, the agent starts to bid from a lower price up to its valuation termed as `bid_below_to_value`. Second, the agent bids beyond its valuation termed as `bid_beyond_value`.

```
Definition bid_below_to_value (b : Bid) (v : Bid): bool :=
   match Z_le_dec b v with
   | left _ => true
   | right _ => false
   end.

Definition bid_beyond_value (b : Bid) (v : Bid): bool :=
   match Z_gt_dec b v with
   | left _ => true
   | right _ => false
   end.
```

The English auction is a type of sequential auction in which bidders have to beat the current bid. A new bid must be higher than the current one, otherwise it is rejected. To take this into account, we have defined the relation `aRboption` and used it to return "Accept" or "Reject" for each new bid via the function `Compare`. Two functions `Agent_flag` and `Find_flag` are used to build up the `Compare` function. `Agent_flag` returns an agent from one `flag`. `Find_flag` searches for the flag of an agent from the list `flaglist`. The return value ($Choice \quad 0\%nat \quad false$) is a default value when the flag of an agent cannot be found. The function `CurrentWinner` returns the winner and its associated bid `aRb`.

```
Inductive aRboption : Type :=
   | Accept : aRb -> aRboption
   | Reject : aRboption.

Definition Agent_flag (f:flag) : Agents :=
  match f with
  | Choice a b => a
  end.

Fixpoint Find_flag (a:Agents) (fl:flaglist) : flag :=
  match fl with
  | nil => (Choice 0%nat false)
  | cons h t => match beq_nat a (Agent_flag h) with
              | true => h
              | false => Find_flag a t
              end
  end.

Definition Compare (fl:flaglist)(new_aRb current_aRb : aRb) : aRboption :=
   match Bool_flag (Find_flag  (Agent_aRb new_aRb) fl) with
   | true => match Z_gt_dec (Bid_aRb new_aRb) (Bid_aRb current_aRb) with
             | left _ => Accept new_aRb
             | right _ => Reject
            end
   | false => Reject
   end.

Definition CurrentWinner (fl:flaglist)(new_aRb current_aRb : aRb) : aRb :=
  match  Compare fl new_aRb current_aRb with
  | Accept n' => n'
  | Reject => current_aRb
  end.
```

The auction ends when all agents have a flag value of false and the winner can be found as the one with the highest bid. Given the agent's valuation v and a payment p, the utility u of an agent is defined as $v - p$ if the agent wins and zero otherwise. This utility function is formalized in `Utility_Eng` wherein the variable `winbid` represents the highest bid in the auction.

```
Definition Utility_Eng (winbid:Bid) (b:Bid) (v:Bid) : Utility :=
   match Z_lt_dec b winbid with
   | left _ => 0
   | right _ => v - b
   end.
```

3.3 The Vickrey Auction Case

In a Vickrey auction, also known as second-price sealed-bid auction, all the bidders submit their bids at a time without any knowledge of other bidders' bids. The highest bidder wins but pays the second-highest bid. There are three bidding strategies in this auction: bid truthfully (or its valuation) encoded in the function `bid_value`, bid below the valuation encoded in the function `bid_below_value`, and bid beyond the valuation through the function `bid_beyond_value`.

```
Definition bid_value (b : Bid) (v : Bid): bool :=
   match Z_eq_dec b v with
   | left _ => true
   | right _ => false
   end.

Definition bid_below_value (b : Bid) (v : Bid): bool :=
   match Z_lt_dec b v with
   | left _ => true
   | right _ => false
   end.

Definition bid_beyond_value (b : Bid) (v : Bid): bool :=
   match Z_gt_dec b v with
   | left _ => true
   | right _ => false
   end.
```

We have used the *List* data structure to store the basic elements of `aRb`. In the definition list of `aRb`, the `binlist` type can be described as follows: it is either an empty (`bnil`) or else a pair of a `aRb` element and a `binlist`. This can be described using the notation :: as an infix *bcons* operator for constructing binding lists.

```
Inductive binlist : Type :=
  | bnil : binlist
  | bcons : aRb -> binlist -> binlist.

Notation "x :: l" := (bcons x l) (at level 60, right associativity).
```

The function `addsortbid` allows us to add and sort a `binlist` in a descending order. In this recursively defined function, all bindings ($Agents \to Bid$) are added to the list one by one. Also, the function `winbid` is used to calculate the winning bid (the head of the sorted `binlist`). When `binlist` is empty, it returns a default value ($Binding \quad 0\%nat \quad 0$). The utility u of an agent is defined as $v - sb$ if the agent wins and zero otherwise, where v is the agent's valuation and sb is the second highest bid in the sorted `binlist`. To calculate the utility of each agent, we need to know the second highest bid in the sorted `binlist`. The function `se_hi_bid` finds the second highest bid when there are at least two elements in the sorted `binlist`. Otherwise, it will return a default value ($Binding \quad 0\%nat \quad 0$).

```
Fixpoint addsortbid (b : aRb) (l : binlist) : binlist :=
   match l with
   | bnil => b :: bnil
   | bcons a l' => match Z_lt_dec (Bid_aRb b)
                     (Bid_aRb a) with
                   | left _ => a :: (addsortbid b l')
                   | right _ => b :: a :: l'
                   end
   end.

Definition winbid (l : binlist) : aRb :=
   match l with
   | bnil => (Binding 0%nat 0)
   | a :: l' => a
   end.

Definition se_hi_bid (l : binlist) : aRb :=
   match l with
   | bnil => (Binding 0%nat 0)
   | a :: l' => match l' with
                | bnil => (Binding 0%nat 0)
                | h :: l'' => h
                end
   end.
```

The `UtilityOfTruthfulBidding` function defines the utility for an agent bidding its valuation v. Recall that the variable sb in this function stands for the second highest bid.

```
Definition UtilityOfTruthfulBidding (v : Bid)
(sb : Bid) : Utility :=
   match Z_le_dec sb v with
   | left _ => v - sb
   | right _ => 0
   end.
```

The utility for an agent in the other two strategies is presented in Algorithm 1. It summarizes the six different conditions giving rise to an agent's utility and is encoded in the function `Utility_OfOtherStrategies`.

```
Definition Utility_OfOtherStrategies (b : Bid) (v : Bid)
      (sb : Bid) : Utility :=
   match Z_gt_dec b v with
   | left _ => match Z_gt_dec sb b with
               | left _ => 0
               | right _ => match Z_le_gt_dec sb v with
                            | left _ => v - sb
                            | right _ => v - sb
                            end
               end
   | right _ => match Z_le_gt_dec sb b with
                | left _ => v - sb
                | right _ => match Z_ge_lt_dec sb v with
                             | left _ => 0
                             | right _ => 0
                             end
                end
   end.
```

Algorithm 1 Computation of *Utility_OfOtherStrategies*

```
  **Variables:**
  $v$: valuation of one agent
  $b$: bid of one agent
  $sb$: second highest bid in the bid list
  $u$: utility of one agent
  **Different Cases in the definition of** *Utility_OfOtherStrategies* **:**
  1. $b > v$
     1.1 $sb > b, u = 0$;
     1.2 $sb \leq v, u = v - sb$;
     1.3 $v < sb \leq b, u = v - sb, u < 0$.
  2. $b < v$
     2.1 $sb \leq b, u = v - sb$;
     2.2 $sb \geq v, u = 0$;
     2.3 $b < sb \leq v, u = 0$.
```

4 Certifying Desirable Properties

In the English auction with private values setting, in the sense that bidders know only their own valuation, buyers sequentially submit their bids. The dominant bidding strategy is for a buyer to start bidding from a lower price and keep increasing its bid until its valuation. In a Vickrey auction, the buyers simultaneously submit their bids and a dominant strategy is for the bidder to bid its valuation. We may be interested in additional auction properties, including *collusion-proofness* meaning that agents cannot collude to achieve a favourable outcome to them, or *false-name bidding free* meaning that agents cannot manipulate the outcome by using fictitious names. We may also be interested in showing that the auction is well-defined function and that it is implemented in line with its specification. In this section, we focus on the certification of dominant strategy in both English and Vickrey auctions. To carry out the Coq proof, all different bidding strategies and their related utilities are examined for comparison. The keyword `Variables` can be used to define local variables in Coq. We can use the keywords `Hypotheses` and `Lemma` to define Hypotheses and Lemma in a Coq proof respectively.

4.1 Certification of Dominant Strategy in the English Auction

For the English auction, the dominant strategy is for each buyer to bid up to its valuation. To provide a machine-checkable proof of this fact, we will use the previously defined utility function `Utility_Eng` along with some hypotheses. Algorithm 2 is used to construct the certificate. This algorithm compares two strategies: bid beyond the valuation (`b > v`) and bid up to the valuation (`b <= v`). In total, there are three cases of comparison using different hypotheses. In all cases, we see that for a buyer to bid up to its valuation yields an utility that is higher or equal to that obtained when a buyer adopts any other strategy.

Algorithm 2 Proving the Dominant Strategy in the English auction

```
**Variables:**
$v$: valuation of one agent
$b$: bid of one agent
$winbid$: the highest bid
$u$: utility of one agent
**Comparison Cases:**
1. $b = winbid$,
   $b > v \to u = v - b < 0$ ( If $b \leq v \to u = v - b \geq 0$, Better);
2. $b < winbid$,
   $b > v \to u = 0$ ( If $b \leq v \to u = 0$, Same);
3. $b > v, b = winbid \to u = v - b < 0$ ( If $b \leq v, b < winbid \to u = 0$, Better).
```

In here, we provide a detailed proof for the first case. The remaining two cases are proved in a similar way. To carry out the Coq proof of the first case, we started by defining the three variables `v`, `b`, and `winbid`. Recall that `v` is the valuation of one agent, `b` is the bid of one agent and `winbid` is the highest bid in one auction.

Variables $v \quad b \quad winbid \quad : Z.$

As seen in Algorithm 2, the first comparison case is on the condition that one agent wins the auction with bid `b`. By relying upon this condition, we introduce the hypothesis `b = winbid` , which means that the bid `b` is the winning bid in the auction. This hypothesis is defined in Coq as:

Hypotheses English_hy1 : $b = winbid.$

All of the Lemmas that are proved in this part rely upon this hypothesis. A tactic *omega*, which is a solver of quantifier-free problems in Presburger Arithmetic, i.e. a universally quantified formula made of equations and inequations, is used in the following proofs. In the next step, we prove Lemma 1 to show that bid `b` is not less than the winning bid `winbid`.

Lemma 1 (not_b_lt_win). $\sim b < winbid.$

Proof. In **English_hy1**, we have bid `b` equals to the winning bid `winbid`. Therefore, bid `b` is not less than the winning bid `winbid`. The proof is carried out by using `English_hy1` and the tactic *omega* in Coq. □

The following Lemma 2 expresses the fact that if one agent bids up to its valuation (`b <= v`), it will get an utility of `v - b`.

Lemma 2 (U_below_to_v). $b <= v \rightarrow \sim b < winbid \rightarrow Utility\_Eng\ winbid\ b\ v = v - b.$

Proof. In here, we use the premises: one agent bids up to its valuation (`b <= v`) and the previously proved Lemma 1. According to the definition of `Utility_Eng`, if bid `b` is less than the winning bid `winbid`, this agent gets the utility of zero. Otherwise, it gets the utility of `v-b`. Furthermore, we have proved that bid `b` is not less than the winning bid `winbid` in Lemma 1. Consequently, this agent gets the utility of `v-b`. The proof is finished by a case-splitting following the definition of function `Utility_Eng` in Coq. □

The next Lemma 3 shows that, under the premise (`b <= v`), the value of `v - b` is greater or equal to 0.

Lemma 3 (v_min_b_ge_O). $b <= v \rightarrow v - b >= 0.$

Proof. The proof is constructed by using the premise `b <= v` and the tactic *omega*. □

Lemma 4 takes Lemma 2 and Lemma 3 as premises, and proves that the utility that the agent gets is greater or equal to 0 when it bids up to its valuation.

Lemma 4 (U_ge_O). $Utility\_Eng\ winbid\ b\ v = v - b \to v - b >= 0 \to Utility\_Eng\ winbid\ b\ v >= 0$.

Proof. Lemma 2 indicates that an agent gets the utility of `v-b`, and Lemma 3 establishes that the value of `v-b` is greater or equals to 0. By using these two lemmas, we can draw the conclusion that this agent gets a nonnegative utility. The proof is built up by combining the Lemma 2 and Lemma 3 in Coq. □

Next, we will calculate and prove that the utility that an agent gets when it bids beyond its valuation under the hypothesis **English_hy1**.
The premises of Lemma 5 are an agent bids beyond its valuation (`b > v`) and the previously proved Lemma `not_b_lt_win`. Under these two premises, we can derive the fact that the agent should get the utility of `v - b`.

Lemma 5 (U_beyond_v). $b > v \to \sim b < winbid \to Utility\_Eng\ winbid\ b\ v = v - b$.

Proof. The proof is carried out by combining the premise `b > v` and Lemma 1. By the definition of `Utility_Eng`, if bid `b` is not less than the winning bid `winbid`, then the agent gets the utility of `v-b`. Lemma 1 establishes that `~b < winbid` is true. So, we have proved that when an agent bids beyond its valuation, it gets utility of `v-b`. We finish this proof by a case-splitting following the definition of function `Utility_Eng` in Coq. □

Lemma 6 shows that under the premise `b > v`, the value of `v - b` is smaller than 0.

Lemma 6 (v_min_b_lt_O). $b > v \to v - b < 0$.

Proof. The proof is constructed by using the premise `b > v` and the tactic *omega*. □

Lemma 7 shows that if an agent bids beyond its valuation, then it will get negative utility.

Lemma 7 (U_lt_O). $Utility\_Eng\ winbid\ b\ v = v{-}b \to v{-}b < 0 \to Utility\_Eng\ winbid\ b\ v < 0$.

Proof. Lemma 5 shows one agent getting the utility of `v-b`, and Lemma 6 establishes that the value of `v-b` is less than 0. Based on these two lemmas, we can conclude that this agent gets a negative utility. The proof is constructed by combining both Lemma 5 and Lemma 6. □

On the basis of **English_hy1**, Lemma 4 establishes that if one agent bids up to its valuation, then it gets nonnegative utility whereas Lemma 7 shows that an agent will get negative utility if it bids beyond its valuation. As a consequence, we can conclude that for an agent to start bidding from a lower price up to its valuation is a better strategy than for that agent bidding beyond its valuation. This terminates the first case. By proving all the remaining cases, we complete the proof of dominant strategy in the English auction.

4.2 Certification of the Dominant Strategy in Vickrey Auction

Our certification is based on the proof in [21]. Six different cases of bidding s-trategies are considered and defined in `Utility_OfOtherStrategies`. They are compared against the outcome of the truthful bidding strategy (bidding its valuation). The schema used to construct our machine-checkable proof is shown in Algorithm 3. As in the case of the English auction, we only demonstrate how to construct the Coq proof of the first case in Algorithm 3, since the remaining cases are dealt with in a similar fashion.

Algorithm 3 Proving the Dominant Strategy in Vickrey auction

```
Variables:
$v$: valuation of one agent
$b$: bid of one agent
$sb$: second highest bid in the bid list
$u$: utility of one agent
Comparison Cases:
1. $sb > b$,
   $b > v \to u = 0$ ( If $b = v \to u = 0$, Same);
2. $sb \leq v$,
   $b > v \to u = v - sb$ (If $b = v \to u = v - sb$, Same);
3. $v < sb \leq b$,
   $u = v - sb < 0$ (If $b = v \to u = 0$, Better);
4. $sb \leq b$,
   $b < v \to u = v - sb$ ( If $b = v \to u = v - sb$, Same);
5. $sb \geq v$,
   $b < v \to u = 0$ (If $b = v \to u = 0$, Same);
6. $b < sb < v$,
   $u = 0$ (If $b = v \to u = v - sb > 0$, Better).
```

Let us start by introducing three variables `v`, `b` and `sb`. The meanings of these variables are listed in Algorithm 3.

Variables $v \quad b \quad sb \quad : Z$.

In the first case of Algorithm 3, we have the hypothesis $sb > b$, meaning that an agent's bid is less than the second highest bid. All of the Lemmas that are proved below are based on this hypothesis.

Hypotheses Vickrey_hy1 : $sb > b$.

The Lemma 8 shows that if one agent bids beyond its valuation (`b > v`), it will get the utility of zero.

Lemma 8 (Utility_of_CaseOne). $b > v \to Utility\_OfOtherStrategies\ b\ v\ sb = 0$.

Proof. We have the premise `b > v`. The definition of `Utility_OfOtherStrategies` states that if an agent bids beyond its valuation (`b > v`) and the second highest bid is greater than this agent's bid (`sb > b`), then it gets the utility of zero. The proof is completed by a case-splitting following the definition of function `Utility_OfOtherStrategies` in Coq. □

So far, we have proved that based on `Vickrey_hy1`, one agent gets the utility of zero if it bids beyond its valuation. Then, we will prove that if one agent bids its valuation, it also gets the utility of zero. To finish this proof, we introduce Lemma 9 in the first step. Lemma 9 shows that `sb` is not smaller or equal to `v` under the premise: `b = v`.

Lemma 9 (not_sb_le_v). $b = v \to sb > v \to\sim sb <= v$.

Proof. The Coq proof is constructed by combining the hypothesis `Vickrey_hy1`, the two premises `b = v, sb > v` and the tactic *omega*. □

The following Lemma 10 shows that when an agent bids its valuation, it gets the utility of zero.

Lemma 10 (Utility_of_Valuation). $\sim sb <= v \to$ `UtilityOfTruthfulBidding` $v\ sb = 0$.

Proof. The conclusion of Lemma 9 is used as a premise. Based on the definition of `Utility_OfTruthfulBidding`, if an agent bids its valuation and the second highest bid `sb` is not less than or equal to its valuation `v`, this agent gets the utility of zero. The proof is carried out by a case-splitting following the definition of the function `Utility_OfTruthfulBidding` in Coq. □

Lemma 11 establishes that under the hypothesis `Vickrey_hy1`, the utility associated with the truthful bidding strategy is the same as that of bidding beyond the valuation for an agent.

Lemma 11 (V_E_SOne). `Utility_OfOtherStrategies` $b\ v\ sb = 0 \to$
`UtilityOfTruthfulBidding` $v\ sb = 0 \to$
`Utility_OfOtherStrategies` $b\ v\ sb =$ `UtilityOfTruthfulBidding` $v\ sb$.

Proof. Using the hypothesis `Vickrey_hy1`, we have proved that an agent gets the utility of zero if it bids beyond its valuation in Lemma 8. Moreover, in Lemma 10, if an agent bids its valuation, then it gets the utility of zero. That is to say, this agent gets the same utility, no matter which strategy it uses. The proof is completed by combining Lemma 8 and Lemma 10 in Coq. □

As mentioned earlier in this section, we do not present the Coq proofs related to the remaining five cases in Algorithm 3 for simplicity of the presentation because these five cases are proved in a similar way. This then completes the Coq certification of truthful bidding be a dominant strategy in Vickrey auction.

5 Discussion

In our current implementation of the FPCC framework to certify auction properties, we have enabled a participating agent to find out desirable properties held by the auction house and to recognize whether a given recommendation is correct or not. For example, suppose a buyer agent visits a *first-price sealed-bid auction* (each agent independently submits a single bid, the highest bidder wins and pays her bid). The server side of this auction house provides this agent with a COQ proof that truthful bidding is a dominant strategy derived from the Vickrey auction. Our system ensures that the proof checker will find a mismatch between the auction specification and the given proof. Thus our implementation enables the buyer agent to find out that *strategyproofness* is not a property of this auction house and that the given proof is wrong. The agent can only check the proof that is related to a well-defined specification, which means that the certificate of dominant strategy in Vickrey auction cannot be used for the English auction for instance. This helps agents distinguish the properties of different auction mechanisms. Our approach can be extended to a broad range of agent-mediated e-commerce systems. For example, we can use this approach to certify whether the winner of the auction is the highest bidder. It also can be applied to verify the communication protocols used by autonomous agents. For the customer who may be concerned by security issues, this approach can be used to verify transaction protocols implemented in an e-commerce system.

One of the limitations of our current work is that an agent cannot understand a previously unseen mechanism unless the specification is part of the common knowledge of this agent. For example, an agent with the knowledge of English auction specification is roaming in the Internet. After this agent arrives at an auction house, it checks the specification of this auction house. The agent can recognize this auction if the specification is an English auction. Otherwise, this agent cannot figure out the type of the auction house. Assume that a human being delegates a task to bid for one item in an English house to a buyer agent. The buyer agent with the knowledge of English auction will join in the English auction house but will ignore any other unrecognized auction house. But, an agent with all the specifications of widely used auction mechanisms can recognize different kinds of auction houses although it requires more computational resources. Nonetheless, it is our intention to extend this implementation by enabling agents to operate on previously unseen protocols by using the semantic web technology so as to build up a shared ontology by the agents and connect this ontology with the COQ formalism in order to enable the verification. Seemingly, Semantic Web Service Language OWL-S is a good Logic-based Language candidate to describe auction mechanisms in a machine understandable formalism.

Note that although, COQ is an interactive theorem prover, we have utilised it to enable automated verification since the proof is constructed only once and agents have to check the correctness of given certificates automatically. Moreover, our approach can be generalised in any kind of auction by making use of ontology based formalism to describe an auction and mapping this description to our COQ specification.

6 Conclusion and Future Work

In this paper, we have used the FPCC framework to e-commerce systems so as to provide certification abilities for software agents. The setting is that of on-line auction markets wherein agents can move between auction houses. Auction houses can publish their mechanism (auction rules) along with proofs of some desirable properties. Buyer agents can download the auction rules, inquire for a property and get the proof for that property so that the agent can check that a proof is indeed correct. We have demonstrated the feasibility of this FPCC approach by formalizing and checking strategy-proofness for the English and Vickrey auctions from within Coq. The ability for an agent to verify auction protocols will increase the trust to an online auction house, which in turn may render this kind of trading attractive and boost its market value.

As future work, we will continue implementing the framework that is proposed in this article. We plan to build an auction house using both Semantic Web [1] and the Java Agent DEvelopment Framework (JADE) [22]. Semantic web provides us with a mechanism that can be used by agents to communicate and understand each other. It also enables software agent to provide intelligent access to heterogeneous and distributed information. In this situation, a software agent is an encapsulated computer system in some environment, capable of perceiving and autonomously acting in that environment. JADE 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. Semantic Web agents can take benefits from Semantic Web technologies in two parts:

– Metadata will be used to identify and extract information from Web sources.
– Ontologies will be used to assist in Web searches, to interpret retrieved information, and to communicate with other agents.

In our scenario, all the information of agents, which are created by JADE, will be translated into an OWL file. Combining the generated auction ontology file with previously defined auction protocol ontology, we can generate an integral Semantic Web Auction system, which is expressed in Semantic Web Languages. Then, this Semantic Web Auction system can be abstracted into Coq specifications. Wherein FPCC can be used for the verification process.

References

1. Berners-Lee, T., Hendler, J., Lassila, O., et al.: The semantic web. Scientific american **284**(5) (2001) 28–37
2. Necula, G.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM (1997) 106–119
3. The Coq Development Team: The coq proof assistant reference manual: Version 8.4. `http://coq.inria.fr` (2012)

4. Tadjouddine, E., Guerin, F.: Verifying dominant strategy equilibria in auctions. Multi-Agent Systems and Applications V (2007) 288–297
5. Tadjouddine, E., Guerin, F., Vasconcelos, W.: 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, International Foundation for Autonomous Agents and Multiagent Systems (2008) 1613–1616
6. Tadjouddine, E.M.: Computational complexity of some intelligent computing systems. International Journal of Intelligent Computing and Cyberneticsl **4**(2) (2011) 144 – 159
7. Tip, F.: A survey of program slicing techniques. Journal of programming languages **3**(3) (1995) 121–189
8. Dolev, S., Panagopoulou, P., Rabie, M., Schiller, E., Spirakis, P.: Rationality authority for provable rational behavior. In: Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing, ACM (2011) 289–290
9. Lapets, A., Levin, A., Parkes, D.: A typed language for truthful one-dimensional mechanism design. Technical report, Boston University Computer Science Department (2008)
10. Sălcianu, A., Arkoudas, K.: Machine-checkable correctness proofs for intra-procedural dataflow analyses. Electronic Notes in Theoretical Computer Science **141**(2) (2005) 53–68
11. Dowek, G., Felty, A., Herbelin, H., Huet, G., Werner, B., Paulin-Mohring, C., et al.: The coq proof assistant user's guide: Version 5.6. (1991)
12. Affeldt, R., Kobayashi, N.: Formalization and verification of a mail server in coq. Software SecurityTheories and Systems (2003) 283–288
13. Affeldt, R., Kobayashi, N., Yonezawa, A.: Verification of concurrent programs using the coq proof assistant: A case study. IPSJ Digital Courier **1**(0) (2005) 117–127
14. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7) (2009) 107–115
15. Gonthier, G.: The four colour theorem: Engineering of a formal proof. Computer Mathematics (2008) 333–333
16. Vestergaard, R.: A constructive approach to sequential nash equilibria. Information Processing Letters **97**(2) (2006) 46–51
17. Cousot, P., Cousot, R.: Basic concepts of abstract interpretation. Building the Information Society (2004) 359–366
18. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation. ACM SIGPLAN Notices **37**(1) (2002) 178–190
19. Barendregt, H.: Lambda calculi with types, handbook of logic in computer science vol. ii (1992)
20. Appel, A.: Foundational proof-carrying code. In: Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on, IEEE (2001) 247–256
21. Fudenberg, D., Tirole, J.: Game theory. 1991 (1991)
22. Bellifemine, F., Caire, G., Greenwood, D.: Developing multi-agent systems with jade (wiley series in agent technology). (2007)