Automated Program Translation in Certifying Online Auctions

Wei Bai
Department of Computer Science
University of Liverpool
England, UK
Wei.Bai@liverpool.ac.uk

Emmanuel M. Tadjouddine
Department of Computer Science and Software Engineering
Xi'an Jiaotong-Liverpool University
SIP, Suzhou, China

Building up trust in agent-mediated online auctions requires agents to have a deeper understanding of the rules of engagement and to be able to certify desirable properties of a given auction mechanism. In previous work [3], we have shown how these mechanisms can be formalised as semantic web services in OWL-S, a good enough expressive machine-readable formalism enabling software agents, to discover, invoke, and execute a web service. In this paper, we have automated the translation of a subset of OWL-S into a COQ imperative language specification in a way that preserves the semantics of the auction mechanism. The semantics equivalence between the source and target codes is guaranteed using a one-to-one mapping whose correctness can be established by using relational Hoare logic. We have also illustrated how we can use Hoare logic to certify auction properties by presenting a COQ proof of the property: the highest bidder always wins in an English auction setting. This kind of trustworthy agent-mediated trading is likely to further increase the volume of transactions in eCommerce systems.

1 Introduction

We envision an e-market wherein human beings can own software agents that can be used for online shopping. A major issue is whether we can trust our software agent to spend our money on our behalf. In general, trust is an important issue in building up markets for rational agents. A market is determined by a mechanism that spells out the rules of engagement for any participant. If the rules are flawed, then participants may exploit the perceived flaws to their benefit. A good example is that of the LIBOR (London Inter Bank Offered Rate) scandal, see for example [17]. The LIBOR relies on a simple mechanism wherein leading banks in London report estimates of interest rates that they would be charged if they borrow from other banks; we then drop the four highest and the four lowest rates and then calculate the arithmetic mean of the remaining rates. This mechanism clearly relies upon trust: banks will report their rates truthfully. However, a small change in the LIBOR rate could generate large payments to a bank giving incentives to certain banks to influence the LIBOR rate. The so-called LIBOR scandal arose from misreporting of interest rates by collusion and manipulation. The question is whether we can ensure that participants are truthful in their reports. In a much more general setting, can we ensure trust in the market?

In this paper, we consider electronic markets based on single item auctions whose mechanisms are described in a 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. A desirable property can be that the highest bidder wins, bidding its true valuation is the optimal strategy or the auction is free from cheating. To carry out such a verification, we rely upon the PCC (Proof-Carrying Code) paradigm:

Submitted to:
VPT 2015

an auctioneer can produce a proof certificate of a given property and a buyer agent can check that the proof is correct, see our previous paper [4] for more details. To implement this approach, we have used the proof assistant COQ [26]. This implies that the auction mechanism needs 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 hardly gained by certifying certain desirable auction properties. We therefore need to ensure that the transformed mechanism is semantically equivalent to the original one.

There is already a large body of literature on certifying program transformations [16]. Leroy's compositional approach in certifying a compiler is most relevant to our work: certify that all phases of the transformation are correct. In our case, we have defined the target language (a WHILE language) within COQ to mirror the input language (OWL-S) and then have used relational Hoare logic [7] to establish the correctness of the transformations. This is carried out by 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 our previous work [4], wherein we have relied upon COQ's constructive approach to carry out the verification of some game-theoretic properties of the auction mechanism.

In terms of prototype implementation, we have used ANTLR [22] 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 our verification approach, we can proceed as in Leroy's paper [16] by implementing all transformation phases from 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 is enough to understand the semantics-preserving nature of our transformation framework. Besides, note that Leroy's approach [16] separates the algorithmic and implementation issues in the certification framework. The contributions of this ongoing work are two-fold: i) we have defined a COQ imperative language to which we can correctly translate an auction mechanism specification based on a machine readable formalism e.g., OWL-S so as to enable verification from within COQ; ii) we have shown how to carry out a verification of a desirable property as a program verification by using the Hoare logic within COQ.

2 Background and Model

We consider single item online auctions that run over a fixed time period in which the seller may have a 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 semantic web e.g., 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 aimed at ensuring 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.

2.1 Language Used

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 [16]. 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.

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.

2.2 A Hoare Logic for Program Translation

A program translation consists in replacing a piece of code C_1 by a new one C_2 in a target programming language. For example, we can transform a C program into a Java or Fortran one. In this case, we must ensure that $C_1 \sim C_2$ meaning C_1 and C_2 are *semantically equivalent*. Two code fragments C_1 and C_2 are semantically equivalent iff for any states σ, σ', if $\sigma \rightarrowtail C_1 \rightarrowtail \sigma'$, then $\sigma \rightarrowtail C_2 \rightarrowtail \sigma'$.

The inference rules for our WHILE language are given in Fig. 1. They use a variant of the relational Hoare logic [7], wherein commands are run over one state in lieu of a couple of states as in [7]. The judgment $\vdash C_1 \sim C_2 : \phi \Rightarrow \psi$ means simply $\{\phi\}C_1\{\psi\} \Rightarrow \{\phi\}C_2\{\psi\}$. In the assignment rule (*asgn*), the lhs variable may be different but is kept the same for clarity. Also, notice that the same conditional branches must be taken (see the *if* rule) and that loops be executed the same number of times (see the *while* rule) on the source and target to guarantee their semantics equivalence. The relational Hoare logic is appropriate in the sense that it is both sound and complete with respect to the intended interpretation. We define $\sigma \models \phi$ to mean ϕ holds at the state σ.

Theorem 1 (Soundness of the translation)**.** *If $C_1 \sim C_2 : \phi \Rightarrow \psi$, then for any states σ, σ' such that $\sigma \rightarrowtail C_1 \rightarrowtail \sigma'$, we have $\sigma \rightarrowtail C_2 \rightarrowtail \sigma'$, and $\sigma \models \phi \Rightarrow \sigma' \models \psi$*

Proof. The proof of this statement is carried out by induction on the relation $\sim$: $C_1 \mapsto C_2$ and subordinate induction on $\sigma \rightarrowtail c \rightarrowtail \sigma'$ for the while loop. ☐

Theorem 2 (Completeness of the translation)**.** *If, for any states σ, σ' such that $\sigma \rightarrowtail C_1 \rightarrowtail \sigma'$, we have $\sigma \rightarrowtail C_2 \rightarrowtail \sigma'$ and $\sigma \models \phi \Rightarrow \sigma' \models \psi$, then $C_1 \sim C_2 : \phi \Rightarrow \psi$.*

Figure 1: Hoare logic for a WHILE language translation

$$\frac{}{\vdash v := e_1 \ \sim \ v := e_2 : \ \phi[e_1/v] \wedge \phi[e_2/v] \Rightarrow \phi} \ asgn$$

$$\frac{\vdash s_1 \ \sim \ c_1 : \ \phi \Rightarrow \phi_0 \quad \vdash s_2 \ \sim \ c_2 : \ \phi_0 \Rightarrow \psi}{\vdash s_1; s_2 \ \sim c_1; c_2 : \ \phi \Rightarrow \psi} \ seq$$

$$\frac{\vdash s_1 \sim c_1 : \ \phi \wedge (b_1 \wedge b_2) \Rightarrow \psi \quad \vdash s_2 \sim c_2 : \ \phi \wedge \neg(b_1 \vee b_2) \Rightarrow \psi}{\vdash \text{if } b_1 \text{ then } s_1 \text{ else } s_2 \ \sim \text{if } b_2 \text{ then } c_1 \text{ else } c_2 \ : \ \phi \wedge (b_1 = b_2) \Rightarrow \psi} \ if$$

$$\frac{\vdash s \sim c : \phi \wedge (b_1 \wedge b_2) \Rightarrow \phi \wedge (b_1 = b_2)}{\vdash \text{while } b_1 \text{ do } s \ \sim \text{while } b_2 \text{ do } c \ : \ \phi \wedge (b_1 = b_2) \Rightarrow \phi \wedge \neg(b_1 \vee b_2)} \ while$$

$$\frac{\vdash \phi \Rightarrow \phi_0 \quad \vdash s \sim c : \ \phi_0 \Rightarrow \psi_0 \quad \vdash \psi_0 \Rightarrow \psi}{\vdash s \sim c : \ \phi \Rightarrow \psi} \ imp$$

3 Machine Readable Description

This section comes from our previous work [3]. 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 us to specify behavioral requirements of components of a system and formulate desirable properties for an individual component or the entire system. The semantic web [8] 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 [19], a W3C standard, is a description logic-based language that enables us 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 [5, 11]. In this work, we advocate using OWL 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.

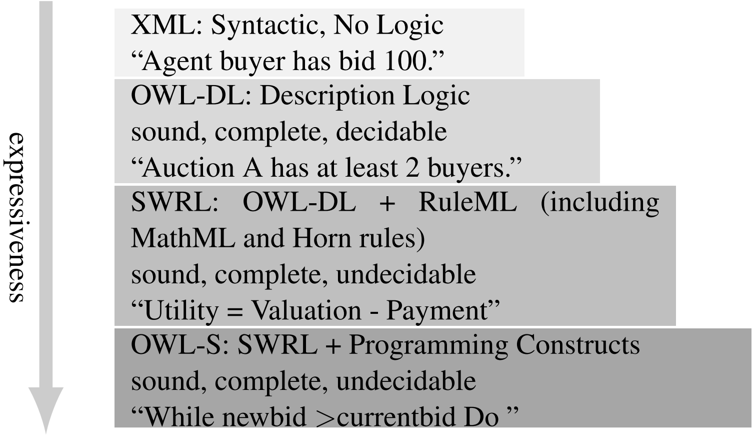
We view an auction as a web service with enough logical attachments enabling a software agent to understand the auction rules and to carry out verifications of claimed properties.

Logic-based languages are usually chosen for their *expressivity* or on the fact that their underlying logic is *sound*, *complete* or *decidable*. Expressivity provides us with 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 for any statement we can construct an algorithm that decides if the statement is true or false.

To enable automated reasoning in the auction system, we have used the ontology language OWL-S, which is based on Description Logics (DL), to build up the auction ontology, See [3] for further details.

3.1 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 a 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.


expressiveness
XML: Syntactic, No Logic
“Agent buyer has bid 100.”
OWL-DL: Description Logic
sound, complete, decidable
“Auction A has at least 2 buyers.”
SWRL: OWL-DL + RuleML (including MathML and Horn rules)
sound, complete, undecidable
“Utility = Valuation - Payment”
OWL-S: SWRL + Programming Constructs
sound, complete, undecidable
“While newbid >currentbid Do ”

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 payment minus valuation’. 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 [14]. However, in SWRL, we cannot express the statement that ’while newbid >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            <process:CompositeProcess rdf:ID="EnglishAuction">
(inputs: (...)                              <process:hasInput> ...
outputs: (...)                              <process:hasOutput>
preconditions: (...)                        <process:hasPrecondition> ...
results: (...)                              <process:hasResult> ...
)                                           ...
{ // Process’s Body                         <process:AtomicProcess rdf:ID="WinDetermAlgo"> ...
WinDetermAlgo(...);                         <process:AtomicProcess rdf:ID="PayeAndUtil"> ...
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.

4 Automated Program Translation

In Section 3, 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 [4] 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 [4]. 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).

4.1 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 [22] 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 so-called 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.

$$\begin{tikzcd}
M^{\text{OWL-S}} \arrow[r, "Translation"] \arrow[d, "Semantics"] & M^{\text{Coq}} \arrow[d, "Semantics"] \\
[\![M^{\text{OWL-S}}]\!] \arrow[r] & [\![M^{\text{Coq}}]\!]
\end{tikzcd}$$

4.1.1 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 2.1. 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 defining a variable and a state, we have then defined the arithmetic and Boolean expressions. Each definition is composed of two functions: one function to define the syntax and the other 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.

4.1.2 Translating Pre- and Post- conditions and Effect

Hoare Logic is used as a paradigm to implement program verification. On 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 2.2, 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 5. The precondition and effect in OWL-S is mapped to the precondition and post-condition of Hoare triples respectively.

4.1.3 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, see Section 3. The translation of these operations into our COQ imperative language is shown in Table 1.

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))`.

Table 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`

4.1.4 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 an unique identifier for this variable.

4.1.5 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 is 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.

4.1.6 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'
```

5 Certifying Desirable Properties

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. In previous work [4], 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 a software agent willing to join the auction house. 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 installs the proof checker, which is a standalone verifier for COQ proofs. After the proof checker is installed on the consumer side, the buyer can now perform all verifications of claimed properties of the auction before deciding whether to join and with which bidding strategy.

COQ [26] is an interactive theorem prover based on the calculus of inductive constructions allowing definitions of data types, predicates, and functions. It also enables us to express mathematical theorems and to interactively develop proofs that are checked from within the system. We have used COQ because it has been developed for more than twenty years [12] 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 and concurrent programs [1]. In [16], COQ was used to develop and certify a compiler. A fully computer-checked proof of the Four Colour Theorem was developed in [13]. In [27], a COQ-formalized proof that all non-cooperative and sequential games have a Nash equilibrium point is presented. It turns out that COQ is a good example of a combination of logic and computation as it allows for the formalization of different types of logic (e.g.,first order logic, typed lambda-calculus, etc.) while providing the possibility of defining functions, which are the cornerstone of computation [9].

```
Definition ListBP : id := Id 0.
Definition newBid : id := Id 1.
Definition sample_englishAuction :=
   WHILE BIsCons (AId ListBP) DO
        newBid ::= (AHeadb (AId ListBP));
    IFB (BLt (AId CurrentBid )(AId newBid)) THEN
     CurrentBid ::= (AId newBid)
    ELSE
     SKIP
    FI;
    ListBP ::= ATail (AId ListBP)
END.
```

Table 2: Proof Sketch of a Sample of an English Auction

```
(1)   {{ListBP = 1 ∧ CurrentBid = 0 ∧ newBid = 0}}
(2)   {{CurrentBid ≥ newBid}}                                 Invariant ∧ Hoare_consequence
         WHILE BIsCons (AId ListBP) DO
(3)            {{CurrentBid ≥ newBid ∧ (BIsCons (AId ListBP))}}   Invariant ∧ The WHILE guard
(4)            {{TRUE}}                                       Hoare_asgn
              newBid ::= (AHeadb (AId ListBP));
(5)            {{TRUE}}                                       Precondition of IF sentence
              IFB (BLt (AId CurrentBid) (AId newBid)) THEN
(6)               {{TRUE ∧ (BLt (AId CurrentBid) (AId newBid))}}  Precondition of IF sentence ∧
                                                              The IF guard
(7)               {{newBid ≥ newBid}}                         Hoare_asgn
                 CurrentBid ::= (AId newBid)
(8)               {{CurrentBid ≥ newBid}}                     Invariant
              ELSE
(9)               {{TRUE ∧ !(BLt (AId CurrentBid) (AId newBid))}} Precondition of IF sentence ∧
                                                              The negation of the IF guard
(10)              {{CurrentBid ≥ newBid}}                     Hoare_skip
                 SKIP
(11)              {{CurrentBid ≥ newBid}}                     Invariant
              FI;
(12)           {{CurrentBid ≥ newBid}}                        Hoare_asgn
              ListBP ::= ATail (AId ListBP)
(13)           {{CurrentBid ≥ newBid}}                        Invariant
         END.
(14)  {{CurrentBid ≥ newBid ∧ !(BIsCons (AId ListBP))}}       Invariant ∧
                                                              The negation of the WHILE guard
(15)  {{CurrentBid ≥ newBid}}                                 Hoare_consequence
```

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 pseudo-linear functions. Not only, do 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 above, *ListBP* is a list of bidders with their own bid variable. *newBid* is the bid for one bidder in one round of the auction. The auction runs from the first bidder in the bidders' list to the last bidder of the list. In a round, a bidder, who does not want to bid in that round, is supposed to have a lower bid for the round. This way, we can simulate the English auction.

Table 2 illustrates a Coq proof of the property "the winner has the highest bid" by using Hoare logic. The invariant in this proof is the relation `CurrentBid ≥ newBid`. The precondition of this program is that `ListBP` is a bidding list, the values of *CurrentBid* and *newBid* are set as zero, which is shown in line (1). The post-condition of this program is the same as the invariant as in line (15). The proof starts as the while guard holds in line (3). In the while loop it is a sequence of commands, which is constructed using the rule of `Hoare_seq`. The precondition of the inner if-sentence is set as `TRUE` in line (4). The proof in the if-sentence is branched as the guard holds in line (5) and not holds in line (9). Line (2) is implied from line (1), while line (15) is implied from line (14). Note that if a property is proven from within Coq, then this property holds in our OWL-S specification since the OWL-S-to-Coq translation is sound and complete.

We can prove more complicated game-theoretic properties from within Coq. In previous work [4], 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 or English 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 [10]. More challenging properties to be checked might be that the auction mechanism is collusion-free or that it is free from fictitious bidding. In this work, we focus on the mapping of auction specifications from OWL-S to COQ so as to enable the verification in a more dedicated and powerful proof development tool.

6 Related Work

There is by now a large body of literature on program transformations and their verification. For example, we may be interested in translating a C code into a Java code or vice versa. More importantly, we may have to ensure 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 for example [16] and the references therein. Usually, the verification of these kind of program transformations rely upon the concept of refinement; that is that the set of behaviors of the target program is a subset of that 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 semantics equivalence between the source and target specifications can be established using the relational Hoare logic in [7].

In the context of specifications translation, the work reported in [20], 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 [24, 25] by using a model checking approach. The computational complexity of such costly verification procedures are investigated in [23]. A typed language which allows for automatic verification that an allocation algorithm is monotonic and therefore truthful was introduced in [15]. More recently, a proof-carrying code approach [21, 2] relying on proofs development tools to enable automatic certification of auction mechanisms have been investigated in [4, 10]. Our work is aimed 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.

7 Concluding Remarks

In this paper, 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 a 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 [7] 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 future work, we will introduce dialogue games [18], which are rule-governed interactions between two or more participants, to build up a framework that enable software agents to automatically query information from a semantic web service holder. The proof-carrying code paradigm will be integrated within an inquiry dialogue, so that the service holder can provide a proof for the buyer agent when she is queried if a desirable property is held in the specific service. Another interesting direction is to explicitly consider the acceptance of properties without formal proofs (e.g., the reputation of a service) in a dialogue. This framework will be instantiated in JADE [6], which is a java platform to develop agent applications for interoperable intelligent multi-agent systems.

References

[1] R. Affeldt, N. Kobayashi & A. Yonezawa (2005): *Verification of Concurrent Programs Using the Coq Proof Assistant: A Case Study*. *IPSJ Digital Courier* 1(0), pp. 117–127, doi:10.2197/ipsjdc.1.117.

[2] A.W. Appel (2001): *Foundational proof-carrying code*. In: *Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on*, IEEE, pp. 247–256, doi:10.1109/LICS.2001.932501.

[3] Wei Bai, Emmanuel M Tadjouddine & Yu Guo (2014): *Enabling Automatic Certification of Online Auctions*. *arXiv preprint arXiv:1404.0854*, doi:10.4204/EPTCS.147.9.

[4] Wei Bai, Emmanuel M. Tadjouddine, Terry Payne & Steven Guan (2013): *A Proof-Carrying Code Approach to Certificate Auction Mechanisms*. In: *The 10th International Symposium on Formal Aspects of Component Software*, doi:10.1007/978-3-319-07602-7_4.

[5] Christopher JO Baker & Kei-Hoi Cheung (2007): *Semantic web: Revolutionizing knowledge discovery in the life sciences*. Springer, doi:10.1007/978-0-387-48438-9.

[6] Fabio Bellifemine, Agostino Poggi & Giovanni Rimassa (2001): *JADE: a FIPA2000 compliant agent development environment*. In: *Proceedings of the fifth international conference on Autonomous agents*, ACM, pp. 216–217, doi:10.1145/375735.376120.

[7] Nick Benton (2004): *Simple relational correctness proofs for static analyses and program transformations*. In: *ACM SIGPLAN Notices*, 39, ACM, pp. 14–25, doi:10.1145/982962.964003.

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

[9] Yves Bertot & Pierre Castéran (2004): *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. springer, doi:10.1007/978-3-662-07964-5.

[10] Marco B. Caminati, Manfred Kerber, Christoph Lange & Colin Rowat (2013): *Proving soundness of combinatorial Vickrey auctions and generating verified executable code*. *CoRR* abs/1308.1779. Available at `http://arxiv.org/abs/1308.1779`.

[11] A-S Dadzie, R Bhagdev, A Chakravarthy, S Chapman, J Iria, V Lanfranchi, J Magalhães, D Petrelli & F Ciravegna (2009): *Applying semantic web technologies to knowledge sharing in aerospace engineering*. *Journal of Intelligent Manufacturing* 20(5), pp. 611–623, doi:10.1007/s10845-008-0141-1.

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

[13] G. Gonthier (2008): *The four colour theorem: Engineering of a formal proof*. *Computer Mathematics*, pp. 333–333, doi:10.1007/978-3-540-87827-8_28.

[14] Ian Horrocks, Peter F Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, Mike Dean et al. (2004): *SWRL: A semantic web rule language combining OWL and RuleML*. W3C Member submission 21, p. 79.

[15] A. Lapets, A. Levin & D. Parkes (2008): *A Typed Language for Truthful One-Dimensional Mechanism Design*. Technical Report, Boston University Computer Science Department.

[16] X. Leroy (2009): *Formal verification of a realistic compiler*. Communications of the ACM 52(7), pp. 107–115, doi:10.1145/1538788.1538814.

[17] James McAndrews, Asani Sarkar & Zhenyu Wang (2008): *The effect of the term auction facility on the london inter-bank offered rate*. Technical Report, Staff Report, Federal Reserve Bank of New York, doi:10.2139/ssrn.1183671.

[18] Peter McBurney & Simon Parsons (2009): *Dialogue games for agent argumentation*. In: Argumentation in artificial intelligence, Springer, pp. 261–280, doi:10.1007/978-0-387-98197-0_13.

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

[20] Huaikou Miao, Tao He & Liping Li (2009): *Formal semantics of OWL-S with F-logic*. In: Computer and Information Science 2009, Springer, pp. 105–117, doi:10.1007/978-3-642-01209-9_10.

[21] G.C. Necula (1997): *Proof-carrying code*. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, pp. 106–119, doi:10.1145/263699.263712.

[22] Terence John Parr & Russell W. Quong (1995): *ANTLR: A Predicated- LL(k) Parser Generator*. Software - Practice and Experience 25, pp. 789–810, doi:10.1002/spe.4380250705.

[23] Emmanuel M. Tadjouddine (2011): *Computational Complexity of Some Intelligent Computing Systems*. International Journal of Intelligent Computing and Cybernetics 4(2), pp. 144 – 159, doi:10.1108/17563781111136676.

[24] Emmanuel M Tadjouddine & Frank Guerin (2007): *Verifying dominant strategy equilibria in auctions*. In: Multi-Agent Systems and Applications V, Springer, pp. 288–297, doi:10.1007/978-3-540-75254-7_29.

[25] Emmanuel M Tadjouddine, Frank Guerin & Wamberto Vasconcelos (2009): *Abstracting and Verifying Strategy-Proofness for Auction Mechanisms*. In: Declarative Agent Languages and Technologies VI, Springer, pp. 197–214, doi:10.1007/978-3-540-93920-7_13.

[26] The Coq Development Team (2012): *The Coq proof assistant reference manual: Version 8.4*. `http://coq.inria.fr`.

[27] R. Vestergaard (2006): *A constructive approach to sequential Nash equilibria*. Information Processing Letters 97(2), pp. 46–51, doi:10.1016/j.ipl.2005.09.010.