Computer-Aided Verification for Mechanism Design

Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, Pierre-Yves Strub

Computer-aided Verification in Mechanism Design

Gilles Barthe   Marco Gaboardi   Emilio Jesús Gallego Arias   Justin Hsu
Aaron Roth   Pierre-Yves Strub

Abstract

In mechanism design, the gold standard solution concepts are *dominant strategy incentive compatibility* and *Bayesian incentive compatibility*. These solution concepts relieve the (possibly unsophisticated) bidders from the need to engage in complicated strategizing. While incentive properties are simple to state, their proofs are specific to the mechanism and can be quite complex. This raises two concerns. From a practical perspective, checking a complex proof can be a tedious process, often requiring experts knowledgeable in mechanism design. Furthermore, from a modeling perspective, if unsophisticated agents are unconvinced of incentive properties, they may strategize in unpredictable ways.

To address both concerns, we explore techniques from computer-aided verification to construct formal proofs of incentive properties. Because formal proofs can be automatically checked, agents do not need to manually check the properties, or even understand the proof. To demonstrate, we present the verification of a sophisticated mechanism: the generic reduction from Bayesian incentive compatible mechanism design to algorithm design given by Hartline, Kleinberg, and Malekian. This mechanism presents new challenges for formal verification, including essential use of randomness from both the execution of the mechanism and from the prior type distributions. As an immediate consequence, our work also formalizes Bayesian incentive compatibility for the entire family of mechanisms derived via this reduction. Finally, as an intermediate step in our formalization, we provide the first formal verification of incentive compatibility for the celebrated Vickrey-Clarke-Groves mechanism.

1   Introduction

Recent years have seen a surge of interest in mechanism design, as researchers explore connections between computer science and economics. This fruitful collaboration has produced many sophisticated mechanisms, including mechanism deployed in high-stakes auctions. Many mechanisms satisfy properties that *incentivize* agents to behave in a straightforward and easily modeled manner; the gold standard properties are *dominant strategy truthful* (in settings of complete information) and *Bayesian incentive compatible* (in settings of incomplete information). While existing mechanisms are impressive achievements, their increasing complexity raises two concerns.

The first concern is correctness. As mechanisms become more sophisticated, proofs of their incentive properties have also grown in complexity, sometimes involving delicate reasoning about randomization or tedious case analysis. Complex mechanisms are also more prone to implementation errors. The second concern is more subtle. At its heart, mechanism design is algorithm design together with a predictive model of how agents will decide to behave. Unlike algorithm design, where

1

correctness can be verified in a vacuum, the success of a mechanism requires a realistic behavioral model of the participants. How will agents behave when faced with a complex mechanism?

Different behavioral models assume different answers to this question. At one extreme, we may assume that agents will coordinate to play a Nash equilibrium of the game and we can study concepts like the *price of anarchy* [24, 8]. However, Nash equilibria are generally not unique, requiring coordination and communication to achieve [17]. Even if information is centralized, equilibria can be computationally hard to find [12]. Assuming that agents play at a Nash equilibrium may be unrealistic unless agents possess strong computational resources.

At the other extreme, we may ask for mechanisms which are dominant strategy truthful or Bayesian incentive compatible. In such mechanisms, agents can do no better than truthfully reporting their type, even in the worst case or in expectation over the other agents' types. These solution concepts assume little about the bidders: When interacting with truthful mechanisms, agents do not have to engage in complicated counter-speculation, communication, or computation—they merely have to tell the truth!

However, even with mechanisms that are dominant strategy truthful or Bayesian incentive compatible, participating agents must still *believe* that the mechanism is truthful. For complicated mechanisms this is no small matter, as the incentive properties may require significant domain expertise to verify. We are not the first to raise these concerns. Li [21] argued for simplicity as a desired feature of auctions, proposing a formal definition; when designing the FCC auction for reallocating radio spectrum, Milgrom and Segal [22] advocated an "*obviously* strategy-proof" mechanism.

However, some useful mechanisms are just too complex to be obvious. Gross, DeArmond, and Denice [14], reporting on experiences with the Denver and New Orleans school choice system, describe the problem:

> Both Denver and New Orleans leaders aggressively conveyed the optimal choosing strategy to parents, and many of the parents we spoke with had received the message. Parents reported to us that they were told to provide the full number of choices in their true order of preference. The problem was that few parents actually trusted this message. Instead, they commonly pursued strategies that matched their own inaccurate explanations of how the match worked.

Hassidim, Marciano-Romm, Romm, and Shorrer [19] report similar behavior in a system for matching Psychology students to slots in graduate programs in Israel. Even though the mechanism is dominant-strategy incentive compatible, nearly 20% of applicants obviously misrepresented their preferences, with possibly more applicants manipulating their reports in more subtle ways. Instead of restricting mechanisms, can we give users evidence for the incentive properties?

In this work, we consider using *formal proofs* as certificates. Formal proofs bear a resemblance to pen-and-paper proofs, but they are constructed in a rigorous fashion: They use a formal syntax, have a precise interpretation as a mathematical proof, and can be built with a rich palette of computer-assisted proof-construction tools. Compared to pen-and-paper proofs, the major benefit of formal proofs is that once constructed, they can be checked independently and fully automatically by a *proof checker* program.

Several previous works have explored formal methods for verifying mechanisms; Kerber, Lange, and Rowat [20] provide an extensive survey. Broadly speaking, prior work falls into two groups. *Automated* approaches check properties via extensive search, guided by intelligent heuristics. These

2

techniques are more suited to verifying simpler properties of mechanisms, perhaps instantiated on a specific input; properties like BIC lie beyond the reach of existing approaches.

More manual (sometimes called *interactive*) techniques divide the verification task into two separate stages. In the first stage, the formal proof is *constructed*. This step typically involves human assistance, perhaps encoding the mechanism in a specific form or constructing a formal proof. With the help of the human, these techniques can prove rich properties like BIC and support the level of generality that is typical of existing proofs—say, for an arbitrary number of agents, or for any type space. In the second stage, the formal proof is *checked*. This step proceeds fully automatically: a proof checking program verifies that the formal proof is constructed correctly. This neat division of the verification task is a natural fit for mechanism design. We could imagine that the mechanism designer—a sophisticated party who is intimately familiar with the details of the proof—has the resources and knowledge to construct the formal proof. This proof could then be transmitted to the agents, who can automatically check the proof with no knowledge of the details.

The main difference between manual techniques is in the amount of human labor for proof construction, the most challenging phase. Existing verification approaches formalize the proof at a level that is far more detailed than existing proofs on paper, requiring extensive expertise in formal methods. Furthermore, existing works focus on general correctness properties—the output of a mechanism should be a partition, the prices should be non-negative, etc., rather than incentive properties.

In our work, we look to combine the best of both worlds: enabling a high level of automation during proof construction, while supporting formal proofs that can capture rich incentive properties. To demonstrate our approach, the primary technical contribution of our paper is a challenging case study: a formal proof of Bayesian incentive compatibility (BIC) for the generic reduction from algorithm design to mechanism design by Hartline, Kleinberg, and Malekian [18]. This example is an attractive proof-of-concept for several reasons.

1. Both the reduction and the proof of Bayesian incentive compatibility are complex. The mechanism is far from obviously strategy proof—indeed, the proof is a research contribution first published at SODA 2011.

2. It is a general reduction, so certifying its correctness once certifies the incentive properties for any instantiation of the reduction.

3. It relies on truthfulness of the Vickrey-Clarke-Groves (VCG) mechanism. As part of our efforts, we provide the first formal verification of truthfulness for this classical mechanism.

4. It employs randomization both within the algorithm and within the agent behavior—agent types are drawn from the known Bayesian prior.

The formal proofs bear a resemblance to the original proof, both easing formalization and making the proofs more accessible to the mechanism design community.

To formalize the proofs, we adapt techniques from program verification. We view incentive properties as a property of the mechanism and the agent's payoff function, both expressed as programs. Formal verification has developed sophisticated tools and techniques for verifying program properties, but general-purpose tools require significant manual work. Verifying even moderately complex mechanisms seems well beyond the reach of current technology. To ease the task, we view incentive properties as *relational properties*: statements about the relationship between the outputs

in two runs of the same program. Specifically, consider the program which calculates an agent's payoff under the mechanism and assume agents play their true value in the first run, while an agent may deviate arbitrarily in the second run. If the output in the first run is at least the output in the second run, then the mechanism is incentive compatible.

With this point of view, we can use tools specialized for relational properties. Such tools are significantly easier to use and have achieved notable successes for verifying proofs from differential privacy and cryptography. We use HOARe^2, a recently-developed programming language that can express and check relational properties [4]. HOARe^2 has been used to verify differential privacy and basic truthfulness in simple mechanisms under complete information, like the fixed price auction and the random sampling mechanism of Goldberg, Hartline, Karlin, Saks, and Wright [13] for digital goods.

Our work goes significantly beyond prior efforts in several respects. First, the mechanism we verify is significantly more complex than previously considered mechanisms, and we analyze all uses of the reduction, rather than just a single instance. Second, the mechanism operates in the partial information setting, so the proof requires careful reasoning about randomization (from both the mechanism and from the prior distribution on types).

The main strength of our approach lies in the high degree of automation during *proof construction*. Once the mechanism and payoff functions have been encoded as programs, and once we have supplied some annotations, we can construct most of the formal proof automatically with the aid of automated solvers. However, there are a handful of particularly complex steps that HOARe^2 fails to automatically prove. To finish the proof, we manually build a formal proof for these missing pieces using EasyCrypt, a proof assistant for relational properties, and Coq, a general purpose proof assistant.[1]

Related work. Closely related to our work, a recent paper by Caminati, Kerber, Lange, and Rowat [7] uses the theorem prover Isabelle to verify basic properties of the celebrated Vickrey-Clarke-Groves (VCG) mechanism. They consider general auction properties: the prices should be non-negative, VCG should produce a partition of goods, etc. Moreover, their framework can be used to automatically produce a correct, executable implementation of the mechanism. While their work demonstrates that formal verification can be applied to verify properties of mechanisms, their results are limited in two respects. First, they do not consider incentive properties, arguably the properties at the heart of mechanism design. Second, they apply general techniques from computer-aided verification that are not specifically tailored to mechanism design, requiring substantial effort to produce the machine-checked proof. Our work uses verification techniques that are particularly suited for incentive properties.

In the appendix we provide a primer on formal verification and discuss related work; a recent survey by Kerber et al. [20] provides a comprehensive review of formal methods for verifying mechanism design properties. The algorithmic game theory literature has for the most part ignored the problem of *verifying* incentive properties, with a few notable exceptions. Recently, Brânzei and Procaccia [6] define *verifiably truthful mechanisms*. Informally, such a mechanism is selected from a fixed family of mechanisms such that for every truthful mechanism in that family, a certificate showing truthfulness can be found in polynomial time. Brânzei and Procaccia [6] consider mechanisms represented as decision trees and show that for the one-dimensional facility location problem, truthfulness for mechanisms in this class can be efficiently verified by linear programming. In contrast, we

---

[1]Our formal proofs, along with code for the HOARe^2 tool, are available online: https://github.com/ejgallego/HOARe2/tree/master/examples/bic

investigate significantly more complex mechanisms in exchange for forgoing worst-case polynomial time complexity.

Mu'alem [23] considers the problem of *property testing* for truthfulness in single parameter domains, which reduces to testing for a variant of monotonicity. Mu'alem [23] gives a tester that can test whether there exist payments that guarantee that truthful reporting is a dominant strategy with probability $1 - \epsilon$, given a poly$(1/\epsilon)$ number of arbitrary evaluations of an allocation rule and assuming agents have uniformly random valuations. In contrast, we assume direct access to the code specifying the auction instead of merely black box access to the allocation rule, and we achieve verification of exact truthfulness, not just approximate truthfulness. We are also able to verify mechanisms in more complex settings, e.g., arbitrary type spaces, randomized mechanisms, and arbitrary priors.

Our work is also related to the literature on automated mechanism design, initiated by Conitzer and Sandholm [11] (see Sandholm [25] or Conitzer [10, Chapter 6] for an introduction). In broad strokes, automated mechanism design seeks to generate truthful mechanisms which optimize the designer's objectives. This is often accomplished by solving explicitly for the distribution on outcomes defining a mechanism using a mixed integer linear program encoding the incentive constraints and objective, an NP hard problem that can often be solved efficiently on typical instances [10]. Automated mechanism design targets a more difficult problem than we do: it seeks not just to *verify* the truthfulness of a given mechanism, but to *optimize* over all truthful mechanisms. However, these techniques have some limitations: they produce explicit representations of mechanisms requiring size exponential in the number of bidders, and they use an explicit integer linear program, requiring a finite type space. In contrast, by only requiring full automation for proof verification and not proof construction, we are able to use a much more sophisticated toolkit—including symbolic manipulation, not just numeric optimization—and verify significantly more complex mechanisms that can have infinite outcome and type spaces.

2 Main example: RSM

As our main proof of concept, we verify that the Replica-Surrogate-Matching (RSM) mechanism due to Hartline et al. [18] is Bayesian incentive compatible. The RSM mechanism reduces mechanism design to algorithm design: given an algorithm A that takes in agents' reported types and selects an outcome, the RSM mechanism turns A into a Bayesian incentive compatible mechanism. Accordingly, our formal proof will carry over to any instantiation of RSM. We first review the original proof by Hartline et al. [18]. Then, we describe our verification process, from pseudocode to a fully verified mechanism.

Let's begin with the standard notion of Bayesian incentive compatibility. We assume there are n agents, each with a *type* t_i drawn from some set of types T. Furthermore, we have access to a distribution μ on types, the *prior*. A *mechanism* is a (possibly randomized) function from the inputs—one per agent—to a single *outcome* o from set O, and a real-valued *payment* p_i for each agent. Without loss of generality, we will assume that the agents each report a type from T as their input. Agents have a valuation $v(t, o)$ for type t and outcome o. Agents have *quasi-linear utility*: their utility for outcome o and payment p is $v(t, o) - p$. We will write (s, t_{-i}) for the vector obtained by inserting s into the ith slot of t. Then, we want to check the following property.

Definition 2.1. *A mechanism M is* Bayesian incentive compatible (BIC) *if for every agent i and*

1. Pick i uniformly at random from $[m]$;

2. Build a *replica type profile* $\vec{r}$ by taking $m - 1$ samples from μ for $\vec{r}_{-i}$, setting $r_i = t$;

3. Build a *surrogate type profile* $\vec{s}$ by taking m samples from μ;

4. Build a bipartite graph with nodes $\vec{r}$, $\vec{s}$, and edges with weight

$$w(r, s) = \mathbb{E}_{t_{-i} \sim \mu^{n-1}}[v(r, A(s, t_{-i}))];$$

5. Run the VCG procedure on the generated graph, and return the surrogate s that is matched to the replica in slot i, and the appropriate payment p.

Figure 1: Procedure R with parameter m

---

types t_i, t_i', we have

$$\mathbb{E}_{t_{-i} \sim \mu^{n-1}}[v(t_i, M(t_i, t_{-i})) - p_i(t_i, t_{-i})] \geqslant \mathbb{E}_{t_{-i} \sim \mu^{n-1}}[v(t_i, M(t_i', t_{-i})) - p_i(t_i', t_{-i})].$$

The expectation is taken over the types t_{-i} of the other agents (drawn independently from μ) and any randomness used by the mechanism.

2.1 The RSM mechanism

Now, let's consider the mechanism: the RSM mechanism in the "idealized model" by Hartline et al. [18]. We will first recapitulate their proof, before explaining in detail how we verify it.

RSM is a construction for turning an *algorithm* $A : T^n \to O$ into a BIC mechanism. The process is easy to describe: each agent individually transforms their type t_i to a *surrogate type* s_i by applying the Replica-Surrogate-Matching procedure R. This procedure also produces a payment p_i for the agent. Then, the surrogates s are fed into the algorithm A, which produces the final outcome.

The procedure R is described in Figure 1. Let m be an integer parameter—the number of replicas. Given input type t, we take $m - 1$ independent samples from μ, the *(r)eplicas*. We then take m independent samples from μ, the *(s)urrogates*. Finally, we select an index i uniformly at random from $[m]$, and place the original type t in the ith "slot" of the replicas $\vec{r}$. We will consider the replicas as "buyers", and the surrogates as "goods", and assign a numeric "value" for every pair of buyer and good. The value of replica r for surrogate s is set to be

$$w(r, s) = \mathbb{E}_{t_{-i} \sim \mu^{n-1}}[v(r, A(s, t_{-i}))], \tag{1}$$

that is, the expected utility of an agent with true type r reporting type s. Finally, RSM runs the well-known Vickrey-Clarke-Groves mechanism [26, 9, 15] to match each replica with a surrogate in this market. The output is the surrogate matched to replica in slot i (the original type t), along with the payment charged.

The original proof. The proof of BIC from Hartline et al. [18] proceeds in two steps. First, they show that R is *distribution preserving*.

6

Lemma 2.1 (Hartline et al. [18]). *Sampling a type $t \sim \mu$ as input to R gives the same distribution (μ) on the surrogates output.*

Proof. When R constructs the list of buyers before applying VCG, the distribution over buyers is simply m independent samples from μ, no matter the value of i. So, we can delay sampling i and selecting the surrogate until after running VCG (via the principle of deferred decision). VCG produces a perfect matching of replicas to surrogates, and the surrogates are also m independent samples from μ. So, sampling a random replica i and returning the matched surrogate is an unbiased sample from μ. □

With the lemma in hand, Hartline et al. [18] show that RSM is BIC.

Theorem 2.1 (Hartline et al. [18]). *The RSM mechanism is BIC.*

Proof. Consider bidder i with type t_i, and fix the randomness for bidder i. In the VCG procedure of R, the value of i's replica for surrogate s is $w(t_i, s)$: the expected utility for submitting s to A while having true type t_i, assuming that all other inputs to A are drawn from μ.

In the RSM mechanism, the other inputs to A are computed by sampling a type $t_j \sim \mu$, and taking the surrogate produced by $R(t_j)$. By Lemma 2.1, the distribution over surrogates is μ. Therefore, $w(t_i, s)$ is bidder i's expected utility in the RSM mechanism for ending up matched to s. Since VCG is incentive compatible, bidder i has no incentive to deviate to any other bid t_i'. By taking expectation over the randomness of i, we get the result. □

Crucially, Theorem 2.1 relies on the truthfulness property of the VCG mechanism. We have also verified this property but we postpone our discussion to Appendix B; the verification of RSM is more interesting.

3 Verifying RSM

Now that we have seen the mechanism, we present our verification step by step.

1. We write the RSM mechanism as a program in the HOARe^2 programming language.

2. We annotate the program with assertions expressing the BIC property, and some additional intermediate facts (lemmas).

3. The tool automatically generates the *verification conditions* (VCs), which imply BIC.

4. The tool uses automatic solvers to check the verification conditions; they may fail to prove some assertions.

5. Finally, we prove the remaining verification conditions by using an interactive prover.
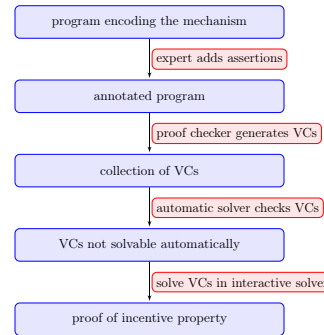
The outcome of these five steps is a formal proof that the RSM mechanism enjoys the BIC property. In the following, we will combine the description of different steps in the same subsection.

```
1  def Rsmdet(j, coins, truety, report) =
2    (rs_{-i}, ss, i) = coins;
3    vcgbuyers = (report, rs_{-i});
4    (surrs, pays) = Vcg(vcgbuyers, ss);
5    return (surrs[j], pays[j])
```

Figure 2: Defining RSM

```
1  def Expwts(j, r, s) =
2    sample others_{-j} = mu^{n-1};
3    algInput = (s, others_{-j});
4    outcome = alg(algInput);
5    return expect_num { value(r, outcome) }
```

Figure 3: Defining weights

Step 1: Modeling the mechanism

To express RSM as a program, we will code a single agent's utility function when running the RSM mechanism, when all the other agents report truthfully and have types drawn from μ. Remembering that we consider truthfulness as a *relational property*, we will then reason about what happens when the agent reports truthfully, compared to what happens when the agent deviates.

We model types and outcomes as drawn from (unspecified) sets T and O, and we assume an algorithm $\mathtt{alg}$ mapping $T^n \to O$. We will consider what happens when the first bidder deviates. This is without loss of generality: if j deviates, we can consider the RSM mechanism with $\mathtt{alg}$ replaced by a version $\mathtt{alg'}$ that first rotates the jth bidder to the first slot, when proving BIC for the first bidder under $\mathtt{alg'}$ implies BIC for the jth bidder under A. For the values, we will assume an arbitrary valuation function $\mathtt{value}$ mapping $T \times O \to \mathbb{R}$. In the code, we will write $\mathtt{mu}$ for the prior distribution μ.

Let's begin by coding the RSM transformation R, which transforms an agent's type into a surrogate type and a payment. It will be convenient to separate the randomness from R. We encode R as a deterministic function $\mathtt{Rsmdet}$, which takes as input the agent number $\mathtt{j}$, the random coins $\mathtt{coins}$, and the input type $\mathtt{report}$. We will have $\mathtt{Rsmdet}$ take an additional parameter $\mathtt{truety}$ representing an agent's true type. This variable does not show up in the code–RSM does not have access to this information—but will be useful later for expressing Bayesian incentive compatibility as a relational property. We will model the slot as a natural number.

In Appendix B we will discuss our treatment of VCG in more detail, but it is enough to know that VCG takes a list of buyers and a list of goods. VCG will output a permutation of goods (representing the assignment), and a corresponding list of payments. In Figure 2, bolded words are keywords and primitive operations of $\mathtt{HOARe^2}$. For a brief explanation, line (2) names the three components of $\mathtt{coins}$: the replicas $\mathtt{rs_{-i}}$, the surrogates $\mathtt{ss}$, and the slot $\mathtt{i}$; line (3) puts the agent's input type $\mathtt{report}$ in the proper slot for the replicas; line (4) calls VCG on the list of buyers $\mathtt{vcgbuyers}$ produced at line (3) and the list of surrogates $\mathtt{ss}$ as goods; and line (5) returns the surrogate and payment.

The $\mathtt{Expwts}$ function in Figure 3 implements the w function from Equation (1), with the additional parameter $\mathtt{j}$ to indicate the agent. In Figure 3, line (2) samples $n-1$ types $\mathtt{others_{-j}}$ from μ for the other agents. These are the types on which the expectation is taken in Equation (1). Line (4) uses the algorithm $\mathtt{alg}$ to compute the outcome $\mathtt{outcome}$ when the agent $\mathtt{j}$ report type $\mathtt{s}$. Finally, the $\mathtt{expect\_num}$ on line (5) takes the expected value of the distribution over reals defined by evaluating the value function $\mathtt{value}$ on the true type $\mathtt{r}$ and on the randomized outcome of the $\mathtt{alg}$.

To check the BIC property, we will code the expected utility for the first bidder and then check that it is maximized by truthful reporting. To break down the code, we will suppose that the function takes in a list of functions $\mathtt{othermoves}$ that transform each of the other bidder's type.

The distribution $\mathtt{rsmcoins}$ defines the distribution over the coins to R, i.e., sampling the replicas

```
1  def Util(othermoves, myty, mybid) =
2    return (expect rsmcoins Helper)
3
4  def Helper(coins) =
5    (mysur, mypay) =
6      Rsmdet(1, coins, myty, mybid);
7    myval = expect_num {
8      for i = 1...n - 1:
9        sample othersurs[i] =
10         (sample otherty = mu;
11          othermoves[i](otherty));
12     algInput = (mysur, othersurs);
13     outcome = alg(algInput);
14     value(myty, outcome) };
15   return (myval - mypay)
```

```
1  def Others(j, t) =
2    sample coins = rsmcoins;
3    (s, p) = Rsmdet(j, coins, t);
4    return s
5
6  def MyUtil(ty,bid) = Util(Others,ty,bid)
```

Figure 4: Defining utility   Figure 5: Defining other reports

$\vec{r}_{-i}$, the surrogates $\vec{s}$, and the coin i. We encoded this distribution in HOARe^2, but we elide it for lack of space. In the code in Figure 4, on line (2) we take expectation of the function Helper over the distribution rsmcoins, with expect. In Helper, we then call Rsmdet on line (6) to compute the surrogate and payment for the agent, passing 1 since we are calculating the utility for the first agent. We sample the other agents' types and transform them on lines (9–11), and we take expectation of the first agent's value for the outcome on lines (7–14). Finally, we subtract off the payment on line (15), giving the final utility for the first agent.

To complete our modeling of RSM, in Figure 5 we plug in Others into the utility function: it simply takes an agent number and a type as input, samples the coins from rsmcoins, and returns the surrogate from calling Rsmdet. So far, we have just written code describing how to implement the RSM mechanism and how to calculate the utility for a single bidder. Now, we express the BIC property as a property about this program and check it with HOARe^2.

Step 2: Adding assertions

We specify properties in HOARe^2 by annotating variable and functions with assertions of the form $\{x :: Q \mid \phi\}$, read as "x is an element of set Q and satisfies the logical formula ϕ". These assertions serve two purposes: (1) they express facts to be proved about the code and (2) they assert mathematical facts about primitive operations like expect and expect_num. The system will then formally verify that the first kind of annotations are correct, while assuming the assertions of the second kind as axioms.

A key feature of HOARe^2 is that the assertion ϕ is *relational*: it can refer to two copies of each variable x, denoted x_1 and x_2. Roughly, we may make assertions about two runs of the same program where in the first program we use variables x_1, and in the second run we use variables x_2.[2] For instance, truthfulness corresponds to the following assertions:

$$\{ty :: T \mid ty_1 = ty_2\} \qquad \text{(true type is equal on both runs)}$$
$$\{bid :: T \mid bid_1 = ty_1\} \qquad \text{(bid is the true type in the first run)}$$
$$\{utility :: \mathbb{R} \mid utility_1 \geqslant utility_2\} \qquad \text{(utility is higher in the first run)}$$

Our goal is to check these assertions for the function MyUtil, which computes an agent's utility in expectation over the other types. Along the way we will use several intermediate facts, encoded

---

[2]These annotations are known as *relational refinement types* in the programming language literature. We will call them assertions or annotations to avoid clashing with agent types.

as assertions in HOARe[2]. Assertions on primitive operations, like `expect` and `expect_num`, are the axioms. Assertions on larger chunks of code are proved correct from the assertions on the subcomponents.

Monotonicity of expectation.   Since the BIC property refers to *expected* utility, we use an expectation operation `expect` when computing an agent's utility (line (2) of the `Util` code). To show BIC, we need a standard fact about *monotonicity* of expected value: for functions $f \leqslant g$, $\mathbb{E}[f] \leqslant \mathbb{E}[g]$ taken over the same distribution. This can be encoded with an annotation for `expect`:

$$\mathsf{distr}\,\{c :: C \mid c_1 = c_2\} \to \{f :: C \to \mathbb{R} \mid \forall x.\ f_1(x) \leqslant f_2(x)\} \to \{e :: \mathbb{R} \mid e_1 \leqslant e_2\}.$$

This annotation indicates that `expect` is a function that takes two arguments and returns a real number. In each of the three components, the annotation before the bar specifies the type of the value: The first argument is a distribution over C, the second argument is a real-valued function $C \to \mathbb{R}$, and the return value is a real number. The logical formulas after the pipe describe how two runs of the expectation function are related. The first component states that in the two runs, the distributions are the same. The second component states that the function f in the first run is pointwise less than f in the second run. The final component asserts that the expected value—a real number—is less on the first run than on the second run.

If think of the distribution as being over the coins `rsmcoins`, this fact allows us to prove deterministic truthfulness for each setting of the coins, then take expectation over the coins in order to show truthfulness in expectation. This is what we need to prove for the BIC property, and is precisely the first step in the original proof of Theorem 2.1.

Distribution preservation.   When we consider a single agent, truthful bidding may not be BIC for arbitrary transformations of the other agents' types (`othermoves` in the `Util` code). As indicated by Lemma 2.1, we also need the transformation to be distribution preserving: the output distribution on surrogates must be the same as the distribution on input types.

Much as we did above, we can capture this property with appropriate annotations. While we have so far used rather simple formulas ϕ that only mention variables in $\{x :: T \mid \phi\}$, in general the formulas ϕ can describe assertions about programs.[3] We can annotate the `othermoves` argument to `Util` to require distribution independence:

$$\left\{\texttt{othermoves} : \mathsf{list}\,(T \to \mathsf{distr}\,T) \mid \forall j \in [n].\ \left(\texttt{sample ot = mu; othermoves[j](ot)}\right) = \texttt{mu}\right\}$$

To read this, `othermoves` is a list of functions f_j that take a type and return a distribution on types, such that if we sample a type from `mu` and feed it to f_j, the resulting distribution (including randomness over the initial choice of type) is equal to `mu`. In other words, this asserts the distribution preservation property of Lemma 2.1 for each of the other agent's transformations.

Facts about VCG.   Recall that `Vcg` takes a list of bidders and a list of goods, and produces a permutation of the goods and a list of payments as output. In our case, the bidders and goods are both represented as types in T, so we can annotate the `Vcg` as:

$$\{buys :: \mathsf{list}\,T\} \to \{goods :: \mathsf{list}\,T\} \to \{(alloc, pays) :: \mathsf{list}\,T \times \mathsf{list}\,\mathbb{R} \mid \texttt{vcgTruth} \wedge \texttt{vcgPerm}\}.$$

---

[3]Of course, we need to actually *check* the assertions eventually, whether by automated solvers or manual techniques.

The two assertions $\mathtt{vcgTruth}$ and $\mathtt{vcgPerm}$ reflect two facts about VCG. The first is that VCG is incentive compatible; this can be encoded like we have already seen, with a slight twist: We require that VCG is IC for a deviation by *any* player rather than just the first player, since the possibly deviating player may be in any slot. More precisely, we define the formula

$$\mathtt{vcgTruth} := \forall_j \in [\mathtt{m}]. \left(\mathtt{bids}_{-j,1} = \mathtt{bids}_{-j,2}\right) \Longrightarrow$$
$$\mathtt{Expwts}(j, \mathtt{bids}_1[j], \mathtt{alloc}_1[j]) - \mathtt{pays}_1[j] \geqslant \mathtt{Expwts}(j, \mathtt{bids}_1[j], \mathtt{alloc}_2[j])) - \mathtt{pays}_2[j].$$

We treat the bid in the first run ($\mathtt{bids}_1[j]$) as the true type, and the bid on the second run ($\mathtt{bids}_2[j]$) as a possible deviation—this is why we evaluate the jth bidder's expected utility using the same true type in the two runs. The second fact we use is that VCG *matches* buyers to the goods. In fact, since the number of goods (surrogates) and the number of buyers (replicas) are equal, VCG produces a perfect matching. We express this by asserting that VCG outputs an assignment that is a permutation of the goods:

$$\mathtt{vcgPerm} := \mathtt{isPerm}\ \mathtt{goods}_1\ \mathtt{alloc}_1 \wedge \mathtt{isPerm}\ \mathtt{goods}_2\ \mathtt{alloc}_2.$$

We verify these properties for a general version of VCG. The verification follows much like the current verification; we discuss the details in Appendix B.

Step 3: Handling proof obligations

After providing the annotations, HOARe^2 is able to automatically check most of the annotations with *SMT solvers*[4]—fully automated solvers that check the validity of logical formulas. Such solvers are a staple of modern formal verification. While the underlying problem is often undecidable, modern solvers employ sophisticated heuristics that can efficiently handle large formulas in practice.

We are able to use SMT solvers to automatically check all but three proof obligations; for these three facts the SMT solvers time out without finding a proof. The first two are uninteresting, and we manually construct the formal proof using the Coq proof assistant. The last obligation is more interesting: it corresponds to Lemma 2.1. Concretely, when we define an agent's expected utility

```
def MyUtil(ty,bid) = Util(Others,ty,bid),
```

recall that $\mathtt{Util}$ asserts that $\mathtt{Others}$ is distribution preserving. This is precisely Lemma 2.1, and the automated solvers fail to prove this automatically.

To handle this assertion we use a more manual tool called EasyCrypt [2, 3], a proof assistant that allows the user prove equivalence of two programs A and B by manually transforming the source code of A until the source code is identical to B.[5] We prove that $\mathtt{Others}$ is equivalent to the program that simply samples from $\mathtt{mu}$ by transforming the code for $\mathtt{Others}$ (including the code sampling the coins of the mechanism, $\mathtt{rsmcoins}$) in several stages. We present the code in Figure 6 with two replicas, for simplicity.

The proof boils down to showing that each step transforms a program to an equivalent program. Our starting point is $\mathtt{stage1}$, the program that samples an agent's type from $\mathtt{mu}$ and runs $\mathtt{Others}$ on the sampled value. Unfolding the definition of $\mathtt{Others}$, $\mathtt{Rsmdet}$, $\mathtt{rsmcoins}$ and including the code that puts the agent's input type in the proper slot for the replicas, we obtain program $\mathtt{stage2}$.

---

[4]Satisfiability-Modulo-Theory, see e.g., [1] for a survey.
[5]This is a common proof technique in cryptographic proofs, known as *game hopping* [5, 16].

```
def stage1 =            def stage2 =            def stage3 =            def stage4 =
 sample ot = mu;         sample ot = mu;         sample ot = mu;         sample s1 = mu;
 Others(ot)              sample r' = mu;         sample r' = mu;         sample s2 = mu;
                         sample s1 = mu;         sample s1 = mu;         sample i = flip;
                         sample s2 = mu;         sample s2 = mu;         if i then s1
                         sample i = flip;                                      else s2
                                                 (r1,r2) = (ot,r');
                         if i then
                          (r1,r2) = (ot,r');      bs = (r1,r2);
                         else                     gs = (s1,s2);
                          (r1,r2) = (r',ot);
                                                 (ss,ps) = Vcg(bs,gs);
                         bs = (r1,r2);           (o1,o2) = ss;
                         gs = (s1,s2);
                                                 sample i = flip;
                         (ss,ps) = Vcg(bs,gs);   if i then o1 else o2
                         (o1,o2) = ss;

                         if i then o1 else o2
```

Figure 6: Code transformations to prove Lemma 2.1.

From there, the main step is to show that we don't need to place the replicas in a random order before calling `Vcg`. Then, we can move the sampling for `i` down past the `Vcg` call, giving `stage3`. Finally, using the fact that the output assignment `ss` of `Vcg` is a permutation of the goods (`s1, s2`), we obtain the program `stage4` and conclude that this is equivalent to taking a single sample from `mu`. This chain of transformations has been verified with EasyCrypt.

4 Perspective

Now that we have presented our verification of the RSM mechanism, what have we learned and what does formal verification have to offer mechanism design going forward? In our experience, while formal verification of game theoretic mechanisms is by no means trivial, verification tools are maturing to a point where practical verification of complex mechanisms can be envisioned. Our verification of RSM, for instance, involved only coding the utility function and adding annotations, most of which can be checked automatically. The most time-consuming part was manually proving the last few assertions.

At the same time, the range of mechanisms that can be verified is less clear. There is an art to encoding a mechanism in the right way, and some mechanisms are easier to verify than others. Since we are trying to verify proofs, the crucial factor is the complexity of the *proof* rather than the complexity of the mechanism. Clean proofs where, each step reasons about localized parts of the program, are more amenable to verification; proof patterns—like universal truthfulness—also help.

In sum, formal verification can manage the increasing complexity of mechanisms by formally proving incentive properties for everyone—mechanism designers, mechanism users, and even mechanism programmers. We believe that the tools to verify one-shot mechanisms are already here. So, we propose a challenge: Try using tools like HOARe[2] to verify your own mechanisms, putting formal verification techniques to the test. We hope that one day soon, verification for mechanisms will be both easy and commonplace.

References

[1] C. Barrett, R. Sebastini, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of satisfiability*, volume 185. IOS press, 2009.

[2] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-aided security proofs for the working cryptographer. In *IACR International Cryptology Conference (CRYPTO), Santa Barbara, California*, pages 71–90, 2011.

[3] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer-Verlag, 2014.

[4] G. Barthe, M. Gaboardi, E. J. Gallego Arias, J. Hsu, A. Roth, and P.-Y. Strub. Higher-order approximate relational refinement types for mechanism design and differential privacy. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Mumbai, India*, pages 55–68, 2015.

[5] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer-Verlag, 2006.

[6] S. Brânzei and A. D. Procaccia. Verifiably truthful mechanisms. In *ACM SIGACT Innovations in Theoretical Computer Science (ITCS), Princeton, New Jersey*, 2014.

[7] M. B. Caminati, M. Kerber, C. Lange, and C. Rowat. Sound auction specification and implementation. In *ACM SIGecom Conference on Economics and Computation (EC), Portland, Oregon*, pages 547–564, 2015.

[8] G. Christodoulou and E. Koutsoupias. The price of anarchy of finite congestion games. In *ACM SIGACT Symposium on Theory of Computing (STOC), Baltimore, Maryland*, pages 67–73. ACM, 2005.

[9] E. H. Clarke. Multipart pricing of public goods. *Public Choice*, 11(1):17–33, 1971.

[10] V. Conitzer. *Computational aspects of preference aggregation*. PhD thesis, IBM, 2006.

[11] V. Conitzer and T. Sandholm. Complexity of mechanism design. In *Conference on Uncertainty in Artificial Intelligence (UAI), Edmonton, Alberta*, pages 103–110. Morgan Kaufmann Publishers Inc., 2002.

[12] C. Daskalakis, P. W. Goldberg, and C. H. Papadimitriou. The complexity of computing a Nash equilibrium. *SIAM Journal on Computing*, 39(1):195–259, 2009.

[13] A. V. Goldberg, J. D. Hartline, A. R. Karlin, M. Saks, and A. Wright. Competitive auctions. *Games and Economic Behavior*, 55(2):242–269, 2006.

[14] B. Gross, M. DeArmond, and P. Denice. Common enrollment, parents, and school choice: Early evidence from denver and new orleans. making school choice work series. Technical report, Center on Reinventing Public Education (CRPE), University of Washington, 2015.

[15] T. Groves. Incentives in teams. *Econometrica: Journal of the Econometric Society*, 41(4): 617–631, 1973.

[16] S. Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005.

[17] S. Hart and Y. Mansour. The communication complexity of uncoupled nash equilibrium procedures. In *ACM SIGACT Symposium on Theory of Computing (STOC), San Diego, California*, pages 345–353. ACM, 2007.

[18] J. D. Hartline, R. Kleinberg, and A. Malekian. Bayesian incentive compatibility via matchings. In *ACM–SIAM Symposium on Discrete Algorithms (SODA), San Francisco, California*, pages 734–747. SIAM, 2011.

[19] A. Hassidim, D. Marciano-Romm, A. Romm, and R. I. Shorrer. 'strategic behavior' in a strategy-proof environment. Working paper, 2016.

[20] M. Kerber, C. Lange, and C. Rowat. An introduction to mechanized reasoning. *CoRR*, abs/1603.02478, 2016.

[21] S. Li. Obviously strategy-proof mechanisms. *SSRN Electronic Journal*.

[22] P. Milgrom and I. Segal. Deffered acceptance auctions and radio spectrum reallocation, 2014.

[23] A. Mu'alem. A note on testing truthfulness. In *Electronic Colloquium on Computational Complexity (ECCC)*, number 130, 2005.

[24] T. Roughgarden. *Selfish routing and the price of anarchy*, volume 174. MIT Press, 2005.

[25] T. Sandholm. Automated mechanism design: A new application area for search algorithms. In *International Conference on Principles and Practice of Constraint Programming (CP), Kinsale, Ireland*, pages 19–36. Springer-Verlag, 2003.

[26] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance*, 16(1):8–37, 1961.

A A note about worst-case complexity

As is typical in program verification, we distinguish between constructing a proof and checking it. Constructing the proof is hard: we do not assume that a proof (or some representation, like a certificate) can be found automatically in worst-case polynomial time, and we will even allow a human to play a limited part in this process. However, checking the proof must be easy: agents should be able to take the formal proof and check it in polynomial time.

While worst-case polynomial time for the entire process would be ideal, it is not very realistic as we cannot expect an algorithm to prove the incentive properties automatically—the proof may be a research contribution; deciding whether an incentive property holds at all may be an undecidable problem. However, relaxing the running time condition when constructing the proof is well-motivated in our application. Unlike the mechanism itself, the proof construction procedure will not be run many times on inputs of unknown origin and varying size. Instead, for a particular mechanism, the proof is constructed just once. In exchange for relaxing worst-case running time, we can verify rich classes of mechanisms.

B Verifying the VCG Mechanism

The celebrated VCG mechanism is a foundational result in the mechanism design literature. It calculates an outcome maximizing social welfare (i.e., the sum of all the agents' valuations) and payments ensuring that truthful bidding is incentive compatible. Let's briefly review the definition.

Definition B.1 (Vickrey [23], Clarke [8], Groves [14])**.** *Let O be a space of outcomes, and let $v : T \times O \to \mathbb{R}$ map agent types and outcomes to real values. Given a reported type profile t from n agents, the VCG mechanism selects the* social-welfare *maximizing outcome:*

$$o^* := \arg\max_{o \in O} \sum_{i \in [n]} v(t_i, o),$$

and computes payments

$$p_j := \max_{o \in O} \sum_{i \in [n] \setminus \{j\}} v(t_i, o) - \sum_{i \in [n] \setminus \{j\}} v(t_i, o^*).$$

That is, the payment for agent j is the difference between the welfare for the other agents without j present, and the welfare for the other agents with j present.

As Vickrey, Clarke, and Groves showed, this mechanism is incentive compatible. Let's consider how to verify this fact in HOARe². Like for RSM, we will start by coding the utility function for a single bidder. We will call it VcgM to distinguish it from the more special case we need for RSM; Figure 7 presents the full code. The parameters to VcgM are a list of valuation functions (values) and a set of possible outcomes (range). We use two helper functions: sumFuns takes a list of valuation functions and sums them to form the social welfare function, while findMax takes a objective function and a set of outcomes, and returns the outcome maximizing the objective.

To encode the incentive property, we will consider two runs of VcgM. We allow any single agent to deviate on the two runs. For the deviating agent, we will model her report in the first run will be her "true" valuation. Then, we want to give VcgM the following annotation:

$$\{\text{values} : O \to \mathbb{R}\} \to \{\text{range} : \text{list } O\} \to \{(\text{out}, \text{pays}) : O \times \text{list } \mathbb{R} \mid \text{out} \in \text{range} \wedge \text{vcgTruth}\}.$$

```
1  def VcgM(values, range) =
2    welfare = sumFuns(values);
3    outcome = findMax(welfare, range);
4
5    for i = 1 ... n:
6      welfWithout = sumFuns(values_−i);
7      outWithout = findMax(welfWithout, range);
8      prices[i] = welfWithout(outWithout) - welfWithout(outcome)
9    end
10
11   (outcome, prices)
```

Figure 7: Encoding the VCG mechanism in HOARe^2

The predicate vcgTruth captures truthfulness, like the assertion in § 3:

$$\mathsf{vcgTruth} := \forall_j \in \big[\mathsf{m}\big]. \; \big(\mathsf{values}_{-j,1} = \mathsf{values}_{-j,2}\big) \implies$$

$$\mathsf{values[j]}_1(\mathsf{out}_1[\mathsf{j}]) - \mathsf{pays}_1[\mathsf{j}] \geqslant \mathsf{values[j]}_1(\mathsf{out}_2[\mathsf{j}]) - \mathsf{pays}_2[\mathsf{j}].$$

With appropriate annotations on findMax, sumFuns, and the "all-but-j" operation $(-)_{-j}$, HOARe^2 verifies VCG automatically.

C A primer on program verification

Program correctness and program verification have a venerable history. In a visionary article, Turing [22] presents a rigorous proof of correctness for a computer routine; although very short, this note prefigures the current trends in deductive program verification and introduces many fundamental ideas and concepts that still remain at the core of program verification today. In particular, Turing makes a clear distinction between the programmer and the verifier, and argues that in order to alleviate the task of the verifier, the programmer should annotate his code with *assertions*, i.e. predicates on program states. Moreover, Turing argues that it should be possible to verify assertions locally and that the correctness of the routine should be expressed by the initial and final assertions, i.e. the assertions attached to the entry and exit points, which respectively capture *hypotheses* on the program inputs and *claims* about the program outputs.

Leveraging contemporary developments in programming language theory, the seminal works of Floyd [12] and Hoare [15] formalize verification methods that adhere to the program proposed by Turing. Both formalisms share similar principles and make a central use of invariants for reasoning about programs with complex control-flow; for instance, both methods use *loop invariants*—assertions that hold when the program enters a loop and remain valid during loop iterations. However, the methods differ in the specifics of proving program correctness. On the one hand, Hoare logic provides a *proof system*—a set of axioms and rules for combining axioms—that can be used to build valid formal proofs that establish program correctness. On the other hand, Floyd calculus computes—from an annotated program—a set of *verification conditions*: formulas of some formal language such as first-order logic, whose validity implies correctness of the program. Despite their differences, the two approaches can be proved equivalent, and assuming that the underlying language of assertions is sufficiently expressive, are *relatively complete* [10]; relative completeness reduces the validity of program specifications to the validity of assertions.

Both Floyd [12] and Hoare [15] are designed to reason about *properties*, i.e. sets of program executions. They cannot reason about the larger class of *hyperproperties* [9], which characterize sets

of sets of program executions. Continuity (small variations on the input induce small variations on the output), and truthfulness (pay-off is maximized when agents play their true value) are prominent *binary* instances of hyperproperties, also called *relational properties*. Reasoning about relational properties is challenging and the subject of active research in programming languages. One way for reasoning about such properties is by using relational variants of Floyd [12] and Hoare [15]. These variants [6] reason about two programs (or two copies of the same program) and use so-called *relational assertions*, assertions which describe pairs of states.

Another challenge in program verification is to deal with probabilistic programs. Starting from the seminal work of Kozen [16], numerous logics have been proposed to reason about properties of probabilistic programs, including [18, 7]. Recently, Barthe, Grégoire, and Zanella-Béguelin [3] propose a relational logic for reasoning about probabilistic programs. Barthe et al. [5] extend and generalize the relational logic to the setting of a higher-order programming language.

In recent years, the theoretical advances in program verification have been matched by the emergence of computer-aided verification tools that have successfully validated large software developments. Most tools implement algorithms for computing verification conditions; the algorithms are similar in spirit to Floyd [12], although they typically use optimizations [11]. Moreover, most systems use fully automated tools to check that verification conditions are valid. However, there is a growing trend to complement this process with an interactive phase, where the programmer interactively builds a proof of difficult verification conditions that cannot be proved automatically. Contrary to automated tools, which try to find a proof of validity, interactive tools try to check that the proof of validity built interactively by the programmer is indeed a valid proof. This interactive phase is often required for proving rich properties of complex programs. It is also often helpful for proving relational properties of probabilistic programs [4].

So far, our account of formal verification has focused on so-called deductive methods: methods where the verification corresponds to build formal proofs that can be constructed using a finite set of rules starting from a given set of axioms. However, there are many alternative methods for proving program correctness. In particular, algorithmic methods, such as model-checking, have been highly successful for analyzing properties of large systems. Algorithmic methods are fundamentally limited by the state explosion problem, since the methods become intractable when the state space becomes too large. Modern tools based on algorithmic verification exploit a number of insights for alleviating the state explosion problem, including symbolic representations of the state space, partial order reduction techniques, and abstraction/refinement techniques.

D   Related Work in Computer-aided Program Verification

There is a small amount of work in the programming languages and computer-aided program verification literature on verification of truthfulness in mechanism design. Lapets, Levin, and Parkes [17] give an interesting approach, by presenting a programming language for automatically verifying simple auction mechanisms. A key component of the language is a type analysis to determine if an algorithm is *monotone*; if bidders have a single real number as their value (*single-parameter domains*), then truthfulness is equivalent to a monotonicity property (e.g., see Mu'alem and Nisan [19]). Their language can be extended by means of user-defined primitives that preserve monotonicity. The paper shows the use of the language for verifying two simple auction examples, but it is unclear how this approach scales to larger auctions, and does not extend beyond single parameter domains.

Wooldridge, Agotnes, Dunne, and van der Hoek [24] promote the use of automatic verification techniques where mechanism design properties are described by means of *specification logics* (like Alternating Temporal Logic [1]), and where the verification is performed in an automatic way by using the *model checking* technique. Similarly, Tadjouddine and Guerin [20] propose a similar approach where first order logic is used as a specification logic. This approach works well for simple auctions with few numbers of bidders but suffers from a state explosion problem when the auctions are complex or the number of bidders is large. This situation can be alleviated by combining different engineering techniques [21], but it is unclear if this approach can be scaled to handle complex auctions with a large number of bidders. Moreover, these automatic approaches do not work in setting of incomplete information.

An alternative approach based on *interactive theorem proving* has been explored by Bai, Tadjouddine, Payne, and Guan [2]. Interactive theorem provers allows specifying and formally reasoning about arbitrary auctions and different truthfulness properties. More generally, they have been used to formalize large theorems in mathematics [13]. Unfortunately, verifying the required properties can require advanced proof engineering skills, even for very simple auctions; Bai et al. [2] consider the English and Vickrey auctions.

References

[1] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, Sept. 2002.

[2] W. Bai, E. M. Tadjouddine, T. R. Payne, and S.-U. Guan. A proof-carrying code approach to certificate auction mechanisms. In *FACS*, volume 8348 of *Lecture Notes in Computer Science*, pages 23–40. Springer-Verlag, 2013.

[3] G. Barthe, B. Grégoire, and S. Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Savannah, Georgia*, 2009.

[4] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub. EasyCrypt: A tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer-Verlag, 2014.

[5] G. Barthe, M. Gaboardi, E. J. Gallego Arias, J. Hsu, A. Roth, and P.-Y. Strub. Higher-order approximate relational refinement types for mechanism design and differential privacy. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Mumbai, India*, pages 55–68, 2015.

[6] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Venice, Italy*, pages 14–25, 2004.

[7] R. Chadha, L. Cruz-Filipe, P. Mateus, and A. Sernadas. Reasoning about probabilistic sequential programs. *Theoretical Computer Science*, 379(1–2):142–165, 2007.

[8] E. H. Clarke. Multipart pricing of public goods. *Public Choice*, 11(1):17–33, 1971.

[9] M. Clarkson and F. Schneider. Hyperproperties. In *IEEE Computer Security Foundations Symposium (CSF), Pittsburgh, Pennsylvania*, 2008.

[10] S. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.

[11] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), London, England*, pages 193–205, 2001.

[12] R. W. Floyd. Assigning meanings to programs. In *Symposium on Applied Mathematics*. Amer. Math. Soc., 1967.

[13] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving (ITP)*, pages 163–179, 2013.

[14] T. Groves. Incentives in teams. *Econometrica: Journal of the Econometric Society*, 41(4): 617–631, 1973.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.

[16] D. Kozen. A probabilistic PDL. *Journal of Computer and System Sciences*, 30(2), 1985.

[17] A. Lapets, A. Levin, and D. Parkes. A Typed Truthful Language for One-dimensional Truthful Mechanism Design. Technical Report BUCS-TR-2008-026, 2008.

[18] C. Morgan, A. McIver, and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems*, 18(3):325–353, 1996.

[19] A. Mu'alem and N. Nisan. Truthful approximation mechanisms for restricted combinatorial auctions. *Games and Economic Behavior*, 64(2):612–631, 2008.

[20] E. M. Tadjouddine and F. Guerin. Verifying dominant strategy equilibria in auctions. In *CEEMAS 2007*, volume 4696 of *Lecture Notes in Computer Science*, pages 288–297. Springer-Verlag, 2007.

[21] E. M. Tadjouddine, F. Guerin, and W. Vasconcelos. Abstracting and verifying strategy-proofness for auction mechanisms. In *Declarative Agent Languages and Technologies VI*, volume 5397 of *Lecture Notes in Computer Science*, pages 197–214. Springer-Verlag, 2009.

[22] A. M. Turing. Checking a large routine. In *Report on a Conference on High Speed Automatic Computation, June 1949*, pages 67–69, 1949.

[23] W. Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance*, 16(1):8–37, 1961.

[24] M. Wooldridge, T. Agotnes, P. E. Dunne, and W. van der Hoek. Logic for automated mechanism design—A progress report. In *AAAI Conference on Artificial Intelligence, Vancouver, British Colombia*, pages 9–17, 2007.