Computer-aided Verification in Mechanism Design

Gilles Barthe* Marco Gaboardi† Emilio Jesús Gallego Arias‡ Justin Hsu§
Aaron Roth¶ Pierre-Yves Strub‖

August 19, 2015

Abstract

In mechanism design, the gold standard solution concepts are *dominant strategy incentive compatibility*, and *Bayesian incentive compatibility*. These simple solution concepts relieve the (possibly unsophisticated) bidders from the need to engage in complicated strategizing. This is a clean story when the mechanism is "obviously" incentive compatible, as with a simple second price auction. However, when the proof of incentive compatibility is complex, unsophisticated agents may strategize in unpredictable ways if they are not convinced of the incentive properties. In practice, this concern may limit the mechanism designer to mechanisms where the incentive properties are obvious to all agents.

To alleviate this problem, we propose to use 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 verify or even understand complicated paper proofs.

To confirm the viability of this approach, we present the verification of one sophisticated mechanism: the generic reduction from Bayesian incentive compatible mechanism design to algorithm design given by Hartline, Kleinberg, and Malekian [17]. This mechanism presents new challenges for formal verification, including essential use of randomness from both the execution of the mechanism and from prior type distributions. As a by-product, we also verify the entire family of mechanisms derived via this reduction.

*IMDEA Software Institute. Email: gilles.barthe@imdea.org
†University of Dundee. Email: m.gaboardi@dundee.ac.uk
‡MINES ParisTech. Email: gallego@cri.ensmp.fr
§University of Pennsylvania Department of Computer and Information Sciences.
¶University of Pennsylvania Department of Computer and Information Sciences. Email: aaroth@cis.upenn.edu
‖IMDEA Software Institute. Email: pierre-yves@strub.nu

1 Introduction

At its heart, mechanism design is algorithm design together with a predictive model of how agents will decide to behave. Unlike algorithm design, where correctness can be verified in a vacuum, the success or failure of a mechanism depends not just on the properties of the mechanism itself, but also on the behavioral model used to describe the participants. Specifically, we should wonder: how rational are the agents, and what can they be expected to do?

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* (see Roughgarden [23], Christodoulou and Koutsoupias [8] or Nisan, Roughgarden, Tardos, and Vazirani [22] for a textbook introduction). These works implicitly assume a very high degree of rationality on the part of participants, both information theoretically and computationally. Nash equilibria are generally not unique, and require coordination and a high degree of communication [16]; even in a centralized setting, they can be computationally hard to find [12].

At the other extreme, we may ask for mechanisms which are *dominant strategy truthful* (in settings of complete information) or *Bayesian incentive compatible* (in settings of incomplete information). These solution concepts are gold standards in mechanism design, because they require very minimal assumptions on agent rationality: Both solution concepts guarantee that in the worst case over an agent's type, she can do no better (in the Bayesian setting, in expectation over the draws of other agents) than truthfully reporting her type to the mechanism. Hence, when interacting with these types of mechanisms, agents do not have to engage in complicated counter-speculation, communication, or computation—they merely have to tell the truth!

Even with dominant strategy truthful (and Bayesian incentive compatible) mechanisms, however, there is a hidden assumption on agent rationality: participating agents must *trust* that the mechanism is truthful. For complicated mechanisms this is no small matter—the incentive properties of the mechanism may require significant domain expertise to verify or prove. This concern has been noted before. In the design of the FCC incentive auction for reallocating radio spectrum, Milgrom and Segal [19] advocate as a key desiderata that the auction be not merely strategy-proof, but "*obviously* strategy-proof"; formalizing this notion is an ongoing area of investigation [18]. However, many useful mechanisms are just too complex to be obvious. Furthermore, agents still need to be sophisticated enough to understand "obvious" properties.

Rather than making this assumption about the agents, there is another path: Can we somehow *convince* users to trust the incentive properties of the mechanisms, if they can't figure it out by themselves? Traditionally, the agents' trust is developed via social means, e.g., through the reputation of the platform running the mechanism. However, reputation is difficult to build up, and new market entrants may not have any reputation at all. A better solution would be to build an infrastructure that does not rely on reputation and instead provides digital evidence of truthfulness properties.

We propose using *formal proofs* as this digital evidence. Formal proofs are digital objects that provide evidence of mathematical claims. They can bear a resemblance to pen-and-paper proofs, but they are akin to computer programs in many respects: They use a formal syntax, have a rigorous semantics (i.e., interpretation as a mathematical object), and are built with computer assistance, using a rich palette of proof-construction tools.

Compared to pen-and-paper proofs, the major benefit of formal proofs is that they can be checked independently and fully automatically using a *proof checker* program, which plays the role of a neutral judge. Therefore, agents can simply run the proof checker to build trust in the mechanism.

The checking burden is quite low: Just like paper proofs, a single formal proof holds for arbitrarily many runs of the mechanisms, and arbitrary choices of parameters, including the number of agents, their types and true values, *etc*; proof checking is typically fast, requiring just seconds for most proofs. Furthermore, each agent only needs to verify the formal proof once, say, when they first use the mechanism. The agents trust just the correctness of the proof checker, a small program developed by a party that is not invested in the mechanism.

To show that this approach is viable, we certify that the generic reduction from algorithm design to Bayesian incentive compatible mechanism design given by Hartline et al. [17] indeed yields a BIC mechanism. We choose this as our proof-of-concept for several reasons.

1. It is complex, and its proof of incentive compatibility is non-trivial, and hence a case in which formal verification can help. It 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 mechanism generated as an instantiation of the reduction.
3. It relies on truthfulness of the Vickrey-Clarke-Groves mechanism. As far as we know, our work provides the first formal verification of truthfulness for VCG with multiple goods.
4. It employs randomization both within the algorithm, and in the input—agent types are drawn from the known Bayesian prior. Accordingly, our verification goes substantially beyond prior work in verification of game-theoretic properties. Indeed, our work is the first formal verification of BIC.

To apply techniques from formal verification, an initial idea is to view incentive properties as properties of the mechanism and the agent's payoff function, expressed as programs. Formal verification has developed many tools and techniques for verifying such program properties, and we might hope we can simply apply existing techniques. However, this approach requires significant manual work in constructing the formal proof, and scaling to verify even moderately complex mechanisms (like our target mechanism) seems well beyond the reach of current technology.

Our primary insight is to view incentive properties as specifically *relational properties*: statements about two runs of the same program. Intuitively, truthfulness is a relational property about the program which calculates an agent's payoff under the mechanism. Agents play their true value in the first run, while the agents may deviate in the second run. If the first pay-off is at least the second pay-off, then the mechanism is truthful.

With this point of view, we can use tools specialized for relational properties, which offer greater automation and significantly more concise proofs. In particular, we use HOARe^2, a recently-developed programming language designed to express and check relational properties [4]. HOARe^2 has been used to verify differential privacy and basic truthfulness, but only in simple mechanisms in settings of complete information like the fixed price auction, and the random sampling mechanism of Goldberg, Hartline, Karlin, Saks, and Wright [13] for digital goods.

The strength of HOARe^2 is its high degree of automation. Once the mechanism and payoff functions have been encoded as programs, and once we have supplied a few annotations, HOARe^2 can typically construct most of the formal proof automatically with the help of automated solvers. However, there are a handful of particularly complex steps where HOARe^2 cannot supply the proof automatically. To complete the verification, we construct the formal proof manually using EasyCrypt, a proof assistant for relational properties, and Coq, a general purpose proof assistant.

Our formal proofs, along with code for the HOARe^2 tool, are available online.[1]

2 Related work

In a recent paper, Caminati, Kerber, Lange, and Rowat [7] used 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. While their work demonstrates that formal verification can be applied to verify auction properties, 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 standard techniques from computer-aided verification that are not specifically tailored to auction properties. Accordingly, their work involves a considerable amount of formal proof engineering, limiting the sophistication of the mechanisms that they consider.

In the appendix, we provide an introductory primer on formal verification, as well as related work from the formal verification literature; interested readers may also consult a survey (e.g., Naumann [21]). Here, we describe the related work in algorithmic game theory. The algorithmic game theory literature has for the most part ignored the problem of *verifying* the incentive properties of mechanisms (instead relying on paper proofs), but there is a small body of related work.

Recently, Brânzei and Procaccia [6] defined *verifiably truthful mechanisms*, and studied them in the context of one-dimensional facility location games without money. Informally, a "verifiably truthful mechanism" is a mechanism selected from a fixed family of mechanisms, such that for every truthful mechanism in that family, there is a certificate proving the truthfulness of that mechanism which can be found in polynomial time. The family of mechanisms considered by Brânzei and Procaccia [6] are represented as polynomially sized decision trees, and they show that for the one-dimensional facility location problem, truthfulness for mechanisms in this class can be efficiently verified using linear programming. They also show that there is a truthful mechanism in this class obtaining a $(1+\epsilon)$-approximation to social welfare for the one-dimensional facility location problem. While this is a fascinating research direction, our work differs in that we handle significantly more complex mechanisms in exchange for forgoing worst-case polynomial time complexity.

Mu'alem [20] considers the problem of *property testing* for truthfulness in single parameter domains, which reduces to testing for a variant of monotonicity. Mu'alem [20] gives a tester that for a single parameter domain, given the ability to query a $\mathrm{poly}(1/\epsilon)$ number of arbitrary evaluations of an allocation rule, can test whether there exist payments that guarantee that truthful reporting is a dominant strategy with probability $1-\epsilon$, where the valuations of the agents are assumed to be drawn uniformly at random. In contrast, in our work, we assume that the verifier has direct access to the code specifying the auction (and not just black box access to the allocation rule), and we require verification of exact truthfulness, not only approximate truthfulness. We are also able to verify mechanisms beyond single parameter domains and in more complex settings—we can handle randomized mechanisms, and ask for Bayesian incentive compatibility over arbitrary priors.

Our work is also related to the literature on automated mechanism design, initiated by Conitzer and Sandholm [11] (see Sandholm [24] or Conitzer [10, Chapter 6] for an introduction). In broad strokes, automated mechanism design seeks to give algorithmic means for computing truthful mechanisms which optimize the designer's objectives, while taking advantage of known specifics about the setting (e.g., prior information about agent types). This is often accomplished by solving

[1] https://github.com/ejgallego/HOARe2/tree/master/examples/bic

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 sets out to solve a more difficult problem than we do: it seeks not just to *verify* the truthfulness of a given mechanism, but to optimize over the space of *all* truthful mechanisms—when mechanisms are given as explicit distributions over outcomes, verifying truthfulness reduces to just verifying a linear constraint over the distribution. As a result, these techniques have some limitations: they typically produce explicit representations of mechanisms that have size exponential in the number of bidders, and they need to write down 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 bring to bear the much more sophisticated toolkit (which includes symbolic manipulation, not just numeric optimization) from the computer-aided program verification literature, and verify significantly more complex mechanisms that don't have concisely defined—indeed, possibly infinite—outcome and type spaces.

3 Main example: RSM

As our main proof of concept, we verify that the Replica-Surrogate-Matching (RSM) mechanism due to Hartline et al. [17] 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 proof of Bayesian incentive compatibility, due to Hartline et al. [17]. Then, we present our verification by walking through the process from the pseudocode to a fully verified mechanism. Rather than providing all the details of the verification process, our aim is to give a sense of what it is like to verify a mechanism, in practice.

3.1 Preliminaries

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 will have *quasi-linear utility*: their utility for outcome o and payment p depends on their type t, and 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 3.1 *A mechanism M is* Bayesian incentive compatible (BIC) *if for every agent i and types t_i, t'_i, we have*

$$\mathbb{E}_{t_{-i}\sim\mu^{n-1}}[v(t_i, M(t_i, t_{-i})) - p_i(t_i, t_{-i})] \geqslant \mathbb{E}_{t_{-i}\sim\mu^{n-1}}[v(t_i, M(t'_i, t_{-i})) - p_i(t_i, t_{-i})].$$

1. Pick i uniformly at random from $[m]$;
2. Build a *replica type profile* $\vec{r}$ by sampling $m-1$ replica types from μ for $\vec{r}_{-i}$, setting $r_i = t$;
3. Build a *surrogate type profile* $\vec{s}$ by sampling m surrogate types from μ;
4. Build a bipartite graph with nodes the elements of $\vec{r}$ and $\vec{s}$ and weighted edges with weight

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

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

The expectation is taken over the types t_{-i} of the other agents (drawn independently from μ) and any randomness that may be used by the mechanism. In other words, the expected utility of any agent is maximized by reporting the true type (where other agents have type independently drawn from μ).

3.2 The RSM mechanism

Now, let's consider the mechanism we will verify: the RSM mechanism in the "idealized model" by Hartline et al. [17]. We will first reproduce their proof, before explaining in detail how we verify it.

The mechanism RSM is a construction for turning an *algorithm* $A : T^n \to O$ into a BIC mechanism. The idea is quite elegant: 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 obtained surrogates s are fed into the algorithm A, which selects the final outcome.

The procedure R is described in Figure 1. Let m be an integer parameter—the number of replicas. On 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 i'th "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

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

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 [25, 9, 14] to match each replica with a surrogate in this market. The final surrogate output by R 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. [17] proceeds in two steps. First, an auxiliary lemma shows that R is *distribution preserving*; this is the main ingredient in showing BIC.

Lemma 3.1 (Hartline et al. [17]) *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 equivalent to taking an unbiased sample from μ. ◻

With the lemma in hand, the proof of BIC follows directly.

Theorem 3.1 (Hartline et al. [17]) *The RSM mechanism is BIC.*

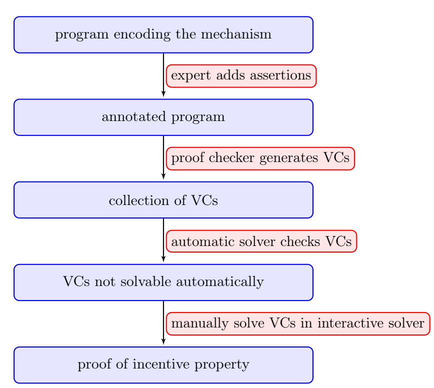
Proof: Consider bidder i with type t_i, and fix the randomness for bidder i. In the VCG procedure of R, the value of i's replica for surrogate s is $w(t_i, s)$: the expected utility for submitting s to A while having true type t_i, assuming that all other inputs to A are drawn from μ.

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 3.1, 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. ◻

Note that Theorem 3.1 relies crucially on the truthfulness property of the VCG mechanism. We have also verified this property but to avoid disrupting the exposition, we present our verification of RSM in the next section, postponing our discussion of VCG to Appendix B.

4 Verifying RSM

Now that we have reviewed the mechanism and the proof, we present our verification step by step. We follow a standard approach to program verification involving five stages:



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 facts that are used as lemmas.
3. The tool automatically generates the *verification conditions* (VCs), whose validity implies the BIC property.
4. The tool uses automatic solvers to check the verification conditions. These tools 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.

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. We will assume we are given an algorithm `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 `alg` replaced by a version `alg'` that first rotates the j'th bidder to the first slot, when proving BIC for the first bidder under `alg'` implies BIC for the j'th bidder under A. For the values, we will assume an arbitrary valuation function `value` mapping $T \times O \to \mathbb{R}$. In the code, we will write `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 for us to separate the randomness from R. We encode R as a deterministic function `Rsmdet`, which takes as input the agent number `j`, the random coins `coins`, and the input type `report`. We will have `Rsmdet` take an additional parameter `truety`. This variable does not show up in the code (as the RSM mechanism does not actually have access to this information), but will be useful later for expressing bayesian incentive compatibility as a relational property. We will model the slot i as a natural number.

In Appendix B we will discuss our treatment of VCG in more detail, but for now, 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.

```
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])
```

For a brief explanation, line (2) names the three components of `coins`: the replicas $\texttt{rs}_{-i}$, the surrogates `ss`, and the slot `i`; line (3) puts the agent's input type `report` in the proper slot for the replicas; line (4) call VCG on the list of buyers `vcgbuyers` produced at line (3) and the list of surrogates `ss` as goods; and the code at line (5) selects the surrogate and payment.

The `Expwts` function implements the w function from Equation (1), with the additional parameter `j` to indicate the agent:

```
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 {
6     value(r, outcome)
7   }
```

Line (2) samples $n-1$ types $\texttt{others}_{-j}$ from μ for the other agents. These are the types on which the expectation is taken in Equation (1). Line (4) uses the algorithm `alg` to compute the outcome `outcome` when the agent `j` report type `s`. Finally, the `expect_num` function at line (5) takes the expectation of a numeric distribution built by using, in line (6), the value function `value` on the true type `r` and on the outcome of the `alg`.

To check the BIC property, we will code the expected utility for the first bidder, and then check that this function is maximized by truthful reporting. To decompose the code a bit, we will suppose that the function takes in a list of functions `othermoves` that correspond to transforming each of the other bidder's type.

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

The distribution `rsmcoins` defines the distribution over the coins to R, i.e., sampling the replica r, the two surrogates s_1, s_2, and the coin i. We encoded this distribution in HOARe^2, but we elide it for lack of space. On line (2) we take expectation of the function `Helper` over these coins, with `expect`. In `Helper`, we then call `Rsmdet` on line (5) 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 (8–10), and we take expectation of the first agent's value for the outcome on lines (7–12). Finally, we subtract off the payment on line (14), giving the final utility for the first agent.

To complete our modeling of RSM, 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`.

```
def Others(j, t) =
  sample coins = rsmcoins;
  (s, p) = Rsmdet(j, coins, t);
  return s

Utility(Others)
```

Of course, 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 need to express the BIC property as a property about this code 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

$$\{x :: Q \mid \phi\}.$$

This should be read as: "x is a variable from some set Q, satisfying the logical formula ϕ". These assertions serve two purposes: (1) they express facts about the code (both the whole program and subprograms) 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, usually written x_1 and x_2. The idea is that 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, to assert truthfulness, the true type must be equal on both runs:

$$\{ty :: T \mid ty_1 = ty_2\},$$

the bid in the first run is equal to the true type (and is unrestricted on the second run):

$$\{bid :: T \mid bid_1 = ty_1\},$$

and the utility is higher on the first run than the second run:

$$\{utility :: \mathbb{R} \mid utility_1 \geqslant utility_2\}.$$

While this is the main property we care about, we need to annotate various facts throughout our verification. We briefly discuss the three main facts we need.

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

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

Like most assertions in HOARe^2, this is read as a statement about how two runs of the expectation function are related. The first component asserts that in the two runs, we are taking expectation over the same distribution. The second component asserts that the function f in the first run is pointwise less than f in the second run (written f_1, f_2 respectively). The final component asserts that the expected value—a real number—is less on the first run than on the second run (written e_1, e_2 respectively).

If we now think of 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 3.1.

Distribution preservation When we consider a single agent, we cannot expect that truthful bidding is BIC for arbitrary transformations of the other agents' types (`othermoves` in the `Utility` code). As indicated by Lemma 3.1, we need the transformation to be distribution preserving: the output distribution on surrogates must be the same as the distribution on input types.

We can again capture this property with appropriate annotations. While we have so far used rather simple formulas ϕ that only mention variables in $\{x :: T \mid \phi\}$, the formulas ϕ can actually make arbitrary assertions about programs.[3] As a result, we can annotate the `othermoves` argument to `Utility` to require distribution independence:

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

[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.

[3]Of course, we need to actually *check* the assertions, whether by automated solvers or more manual techniques. But a priori, there is no problem in asserting (and using) the facts.

To read this, `othermoves` is a list of functions f_j that take a type and returns 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 3.1 for each of the other agent's actions.

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:

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

The two assertions `vcgTruth` and `vcgPerm` in the last component 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 we are placing the possibly deviating player's in a random slot. More precisely, we define the formula

$$\begin{aligned}\texttt{vcgTruth} := \forall \texttt{j} \in [\texttt{m}].\ (\texttt{bids}_{-j,1} = \texttt{bids}_{-j,2}) \implies \\ \texttt{Expwts(j, bids}_1\texttt{[j], alloc}_1\texttt{[j])} - \texttt{pays}_1\texttt{[j]} \geqslant \texttt{Expwts(j, bids}_1\texttt{[j], alloc}_2\texttt{[j]))} - \texttt{pays}_2\texttt{[j]}.\end{aligned}$$

We treat the bid in the first run ($\texttt{bids}_1\texttt{[j]}$) as the true type, and the bid on the second run ($\texttt{bids}_2\texttt{[j]}$) as a possible deviation—this is why we evaluate the jth bidder's expected utility using the same "true type". 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:

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

We verify these properties for a general version of VCG. The verification follows much like the current verification; we will 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. 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 3.1. Concretely, this arises when we try to calculate the utility by plugging in the other agents' moves:

```
def Others(j, t) =
  sample coins = rsmcoins;
  (s, p) = Rsmdet(j, coins, t);
  return s

Utility(Others)
```

[4]Short for Satisfiability-Modulo-Theory, see Barrett, Sebastini, Seshia, and Tinelli [1] for a survey.

```
def stage1 =
 sample ot = mu;
 Others(ot)
```

```
def stage2 =
 sample ot = mu;
 sample r' = mu;
 sample s1 = mu;
 sample s2 = mu;
 sample i = flip;

 if i then
  (r1,r2) = (ot,r');
 else
  (r1,r2) = (r',ot);

 bs = (r1,r2);
 gs = (s1,s2);

 (ss,ps) = Vcg(bs,gs);
 (o1,o2) = ss;

 if i then o1 else o2
```

```
def stage3 =
 sample ot = mu;
 sample r' = mu;
 sample s1 = mu;
 sample s2 = mu;

 (r1,r2) = (ot,r');

 bs = (r1,r2);
 gs = (s1,s2);

 (ss,ps) = Vcg(bs,gs);
 (o1,o2) = ss;

 sample i = flip;
 if i then o1 else o2
```

```
def stage4 =
 sample s1 = mu;
 sample s2 = mu;
 sample i = flip;
 if i then s1
      else s2
```

Figure 2: Code transformations to prove Lemma 3.1.

For the last line (6), recall that we assert that `Others` is distribution preserving; we need to check this fact. This is precisely Lemma 3.1, and is a bit too complex to solve automatically.

To handle this last problematic assertion, we use a more manual tool called EasyCrypt [2, 3]. This tool is 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. This is a common proof technique in cryptographic proofs, known as *game hopping* [5, 15]. For our purposes, we used EasyCrypt to prove that `Others` is equivalent to the program that simply samples from `mu`. This involves transforming the code for `Others` (including the code sampling the coins of the mechanism, `rsmcoins`) in several stages. We present the code in Figure 2 with just two replicas, for simplicity.

The proof boils down to showing that each step transforms a program to an exactly equivalent program. Our starting point is `stage1`, the program that samples an agent's type from `mu` and runs `Others` on the sampled value. Unfolding the definition of `Others`, `Rsmdet`, `rsmcoins` and by making explicit the code that puts the agent's input type in the proper slot for the replicas we have the program `stage2`. 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, by using 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.

5 Perspective

Now that we have presented our verification of the RSM mechanism, it's worth asking: what have we learned, and what does formal verification have to offer mechanism design going forward?

Through our experience, we have found that while formal verification of game theoretic mechanisms is by no means trivial, practical verification of complex mechanisms is within reach. By "practical", we mean that the formal proof should be (1) constructible by people familiar with the proof, but not expert in formal verification; (2) concise, bearing as much resemblance to the original proof as possible; and (3) checkable entirely automatically. While tools like HOARe^2 do not meet all these criteria exactly, we believe they come close. Our verification of RSM, for instance, involved only coding the utility function (transformed from the pseudocode of the mechanism) and adding annotations to be checked automatically.

At the same time, the range of mechanisms that can be practically verified is less clear. Since we are trying to verify proofs, the main bottleneck is the complexity of the proof, rather than the complexity of the mechanism itself. We do not have a precise characterization of which mechanisms or proof structures can be feasibly verified in systems like HOARe^2, but we are optimistic that these tools are mature enough to capture many of the mechanisms proposed today.

Of course, the process of constructing the proof may be much more difficult than we have shown for RSM; there is an art to encoding a mechanism in the right way, and some mechanisms are easier to verify than others. We can only offer some observations from our experience: Clean proofs where each step reasons about localized parts of the program area are easier to verify; common proof patterns—like universal truthfulness—can also simplify verification.

One interesting challenge for formal verification is handling mechanisms that operate as extensive form games, rather than one-shot games. These mechanisms are commonplace in practice (e.g., the ascending price auction), but there are several obstacles to verification. First, extensive-form mechanisms rarely make sincere bidding a *dominant* strategy, because of the possibility of *threats* from other agents. Hence, we may need to work with more delicate solution concepts, like *ex-post Nash equilibrium*; encoding such properties as program properties may not be trivial. Second, the execution of an ascending price auction involves not just the code describing the auction mechanism, but also the decisions of all the agents across the rounds. While our verification of BIC already models the other agents, the agents in an extensive form game can behave adaptively and are more difficult to model. Here, we hope to borrow ideas from verification of cryptographic systems, which often involves modeling adaptive agents/adversaries.

While there are many other directions for future work in formal verification, let us conclude with implications for mechanism design. Formal verification can manage the increasing complexity of mechanisms by formally proving incentive properties for everyone—mechanism designers, mechanism users, and even mechanism programmers. As we have shown, 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 in the near future, 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 *LNCS*, pages 146–166. Springer, 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. ACM, 2015.

[5] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *IACR International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT), Saint Petersburg, Russia*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426, Heidelberg, 2006. Springer.

[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] T. Groves. Incentives in teams. *Econometrica: Journal of the Econometric Society*, pages 617–631, 1973.

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

[16] 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.

[17] 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.

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

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

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

[21] D. A. Naumann. Theory for software verification. 2009.

[22] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani. *Algorithmic game theory*. Cambridge University Press, 2007.

[23] T. Roughgarden. *Selfish routing and the price of anarchy*, volume 174. MIT press Cambridge, 2005.

[24] 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, 2003.

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

A A note about worst-case complexity

In line with the typical program verification setting, 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, the checking must be easy: agents should be able to take the formal proof and efficiently verify it fully automatically.

While worst-case polynomial time for the entire verification process would be preferable, it is not very realistic as we cannot really 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. Furthermore, relaxing the running time condition when constructing the proof is particularly 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 (an input of fixed size), 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 cornerstone of 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 of this mechanism.

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 produces the* social-welfare *maximizing outcome:*

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

and prices

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

That is, the price 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 incentive compatibility for VCG in HOARe^2. 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 3 presents the full code.

```
def VcgM(values, range) =
  welfare = sumFuns(values);
  outcome = findMax(welfare, range);

  for i = 1...n:
    welfWithout = sumFuns(values_{-i});
    outWithout = findMax(welfWithout, range);
    prices[i] = welfWithout(outWithout) - welfWithout(outcome)
  end

  (outcome, prices)
```

Figure 3: Encoding the VCG mechanism in HOARe^2

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; `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. No matter which agent deviates, we will model her report in the first run as her "true" valuation. Then, we want to give `VcgM` the following annotation:

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

The predicate `vcgTruth` captures the truthfulness, and is similar to the assertion in § 4:

$$\mathsf{vcgTruth} := \forall \mathsf{j} \in [\mathsf{m}].\ (\mathsf{values}_{-j,1} = \mathsf{values}_{-j,2}) \implies$$
$$\mathsf{values[j]}_1(\mathsf{out}_1\mathsf{[j]}) - \mathsf{pays}_1\mathsf{[j]} \geqslant \mathsf{values[j]}_1(\mathsf{out}_2\mathsf{[j]}) - \mathsf{pays}_2\mathsf{[j]}.$$

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 named *relational properties*. Reasoning about relational properties is challenging and subject of active research in programming languages. A 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]. More 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 incurs in a state explosion 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 like the one for Bayesian Incentive Compatibility.

An alternative approach based on *interactive theorem proving* has been explored by Bai, Tadjouddine, Payne, and Guan [2]. Interactive theorem provers permit to specify and formally reason about arbitrary auctions and different truthfulness properties. More in general they can be used to formalize large parts of mathematics Gonthier, Asperti, Avigad, Bertot, Cohen, Garillot, Roux, Mahboubi, O'Connor, Biha, Pasca, Rideau, Solovyev, Tassi, and Théry [13]. Unfortunately, verifying the required properties can require advanced proof formalization skills that only specialized user have. Moreover, the complete formalization of complex auctions can require a huge amount of work also for specialized users. The examples showed by Bai et al. [2] are very simple mechanisms like 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 *LNCS*, pages 23–40. Springer, 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*. ACM, 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 *LNCS*, pages 146–166. Springer, 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. ACM, 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), 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*. IEEE, 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. ACM, 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*, pages 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), 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 *LNCS*, pages 288–297. Springer, 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 *LNCS*, pages 197–214. Springer Berlin Heidelberg, 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.