Sound Auction Specification and Implementation

MARCO B. CAMINATI, University of Birmingham, UK
MANFRED KERBER, University of Birmingham, UK
CHRISTOPH LANGE, Fraunhofer IAIS, University of Bonn, Germany; University of Birmingham, UK
COLIN ROWAT, University of Birmingham, UK

We introduce 'formal methods' of mechanized reasoning from computer science to address two problems in auction design and practice: is a given auction design soundly specified, possessing its intended properties; and, is the design faithfully implemented when actually run? Failure on either front can be hugely costly in large auctions. In the familiar setting of the combinatorial Vickrey auction, we use a mechanized reasoner, Isabelle, to first ensure that the auction has a set of desired properties (e.g. allocating all items at non-negative prices), and to then generate verified executable code directly from the specified design. Having established the expected results in a known context, we intend next to use formal methods to verify new auction designs.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.2.4 [**Programming Techniques**]: Software/Program Verification; F.3.1 [**Logics and meanings of programs**]: Specifying and Verifying and Reasoning about Programs; F.4.1 [**Mathematical logic and formal languages**]: Mathematical logic; I.2.3 [**Artificial intelligence**]: Deduction and theorem proving; J.4 [**Social and behavioral sciences**]

Additional Key Words and Phrases: formal proof, mechanized reasoning, auction theory

1. INTRODUCTION

This paper introduces a novel, integrated approach to two questions in auction design and practice. First, is the auction design *soundly specified*, in the sense of possessing the properties that its designers wish it to have? Second, does the actual auction *faithfully implement* the specification? Failure on either front can be hugely costly – both financially and reputationally – especially in high-stakes, one-off auctions.

In some simple cases, these issues do not seem pressing. For these, theoretical results may exist (e.g. the well-known revenue equivalence results, or Vickrey's theorem). Further, leading auction theorists do not doubt the soundness of the major results in auction theory, nor are there any well-known examples of costly errors being discovered in manually-derived theorems in auction theory.

Typically, auction designers' greater concern is that the auction must operate in less restrictive conditions than theory typically assumes (e.g. risk-neutrality, or common knowledge of the support of bidders' valuations). Thus, auction designs and their software implementations are typically extensively tested before actual use, in experimental labs, perhaps on test data such as that generated by CATS [Leyton-Brown and Shoham 2006], or in 'dry runs' with the intended bidders. While necessary, such tests

This work has been supported by EPSRC grant EP/J007498/1. See http://www.cs.bham.ac.uk/research/projects/formare/ for the ForMaRE project homepage.
Authors' addresses: Marco B. Caminati and Manfred Kerber, School of Computer Science, University of Birmingham; email: {m.b.caminati,m.kerber}@cs.bham.ac.uk; Christoph Lange, Fraunhofer IAIS and University of Bonn, Germany; email: math.semantic.web@gmail.com; Colin Rowat, Department of Economics, University of Birmingham; email: c.rowat@bham.ac.uk.

EC'15, June 15–19, 2015, Portland, OR, USA. ACM 978-1-4503-3410-5/15/06.
http://dx.doi.org/10.1145/2764468.2764511

can only demonstrate that no anomalies have been discovered yet on the finite set of test cases to which the auction has been exposed.

Dijkstra, recipient of the 1972 Turing Award, famously remarked that "testing shows the presence, not the absence of bugs" [Buxton and Randell 1970]. NASDAQ's Facebook IPO provided a recent example of this, allowing "a race between new orders and the print of the opening trade, an infinite loop that . . . hundreds of hours of testing had missed" [Kirilenko and Lo 2013].

Using tools from mechanized reasoning, we formally prove that an auction design has certain properties and extract verified executable code to run it, providing a higher level of confidence both in its sound specification and its faithful implementation. Figure 1 outlines our approach, which uses Isabelle, a powerful and flexible mechanized theorem prover. We first supply definitions specifying an auction and theorems stating its properties in Isabelle's formal language. For the most part, we also supply formal proofs of the theorems to Isabelle (an example of *interactive theorem proving*); at some points, we use Isabelle's *automated theorem proving* tools to avoid the more tedious, fine-grained steps. Isabelle then checks our proofs and extracts executable Scala code directly from the designs whose properties we have mechanically verified.[1]

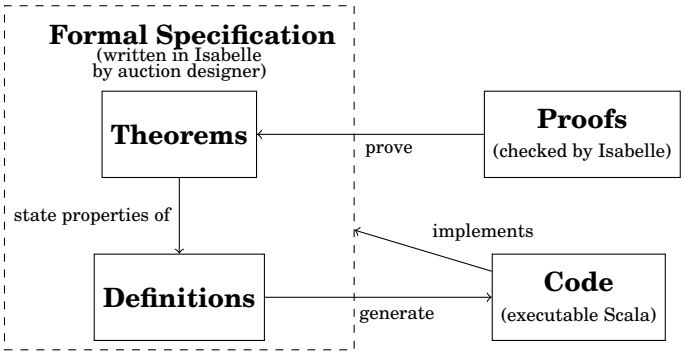

Figure 1: High-level outline of our approach

Mechanized reasoners do not eliminate all possibilities of error. First, an auction designer must still correctly formalize the definitions and theorems, since a reasoner cannot know what a definition is meant to specify, or whether a specification is in line with common usage. Second, we replace trust in our manual proofs with trust in mechanical tools.

To address the first concern, the task of checking statements' correctness may be easier in a single mechanized reasoner than under the standard, manual approach. Manually, different notation may be used for the same objects, or the same notation to denote slightly different objects. Manual statements are also more prone to ambiguity and under-specification. Mechanized reasoners do not allow any logically necessary step to be ignored, nor do they paper over inconsistencies in definitions.[2] Mechanized

[1]Besides the Java-based Scala, Isabelle can also generate executable code in three more functional languages: SML, OCaml, and Haskell.

[2]For example, Aygün and Sönmez [2013] recently identified a hidden assumption in Hatfield and Milgrom [2005] – which they view as "widely considered to be one of the most important advances of the last two decades in matching theory" – without which many of their results fail to hold. The oversight arose from "an ambiguity in setting the primitives of the model", which would produce an error in a mechanized reasoner.

reasoners' advantage in this respect comes to the fore when, for example, changing the specification of an object: the reasoner itself quickly checks whether the new statement yields any inconsistencies, a laborious task to perform by hand. (The final section of this paper returns to this point with an example.)

As to the second concern, theorem provers like Isabelle are designed to satisfy the so-called de Bruijn criterion [Barendregt and Barendsen 1997], requiring that their logical cores be small enough to be checked by hand. Indeed, the core for higher-order logics need not exceed a page or two (q.v., for example, [Paulson 2014, Section 2.2]). If these cores are open to public scrutiny, as is Isabelle's, then – once they have been investigated by a large community – one has greater confidence that any result produced by the core is sound.[3] This is much more efficient than requiring a community to investigate, for example, each proof.[4] The highest level of a widely used software security specification is Evaluation Assurance Level 7 (EAL7), whereby properties of the software's design have been formally verified and tested. Even software written to much lower levels of security is routinely trusted: compilers for high-level programming languages are much lengthier than the logical cores of theorem provers, but questions about their reliability rarely arise; when human experts checked the computations behind the proof of the four-color map theorem, they either confirmed the computer's output, or – when their calculations were in conflict with the computer's – discovered errors in their own [Wilson 2005].

We formally specify the combinatorial Vickrey's auction, or – equivalently – the combinatorial Vickrey-Clarke-Groves (VCG) mechanism. This is, of course, well understood (q.v. [Ausubel and Milgrom 2006]): there is no doubt that it is well defined, nor that it has been faithfully implemented. We work with it for this reason, introducing the novel techniques of mechanized reasoning and code generation in a simple, familiar environment, thus removing doubts about the results themselves. Future work will implement computationally efficient algorithms, and apply these techniques to auction designs whose properties are not yet established, and whose implementations are not yet verified. This should ensure a higher level of reliability of such auctions, as well as easing alterations to them. These techniques are not, of course, restricted to auctions. Indeed, a larger application domain may be finance, which relies upon algorithms at almost every step [The Government Office for Science 2012].[5]

Section 2 introduces formal proofs and mechanized reasoning. Section 3 provides a 'paper' definition of Vickrey's auction. Section 4 translates this definition into Isabelle, and describes the ensuing Isabelle theorems, along with their proofs. Section 5 discusses code extraction in detail. Section 6 introduces the Auction Theory Toolbox containing the code generated for this paper, allowing it to be reused and extended. Section 7 discusses related work. Section 8 concludes and notes how easily our definitions and proofs may be modified to specify combinatorial first-price auctions.

2. BRIEF OVERVIEW OF MECHANIZED REASONING

The idea of mechanizing reasoning dates back at least to Leibniz [1686], who envisaged a machine which could compute the validity of arguments and the truth of mathematical statements. The development of formal logic from 1850 to 1930, the advent of the computer, and the inception of *artificial intelligence* (AI) as a research field at the

[3]There may, of course, be inconsistencies in the logical core; however, since 1901 when Russell discovered antinomies in Frege's system, no more have been discovered.

[4]Harrison [2006b] takes this one step further, using HOL Light – another theorem prover – to formally investigate its own core. Of course, if HOL Light is inconsistent, then it may not detect its own inconsistency.

[5]See Muhit [2014] for a recent commentary on the use of functional programming languages – the basis for many theorem proving systems – in finance.

1956 Dartmouth Conference all paved the way for the first mechanized reasoners in the 1950s and 1960s, most notably de Bruijn's Automath. Computer programs realizing this old idea are called mechanized reasoners, proof checkers, or proof assistants, according to the varying functionalities they provide.

At their heart, all seek to certify whether a mathematical statement is provable, by performing the complete sequence of simpler logical steps, many of which are often omitted in a standard paper proof. Doing so requires first defining a mathematical foundation (the axioms) and a set of basic inference rules (which specify the admissible "simpler logical steps"). Together, these form a calculus.

In pure mathematics, mechanized support has been most famously provided in three cases. First, over 100 years after the four-color map theorem was posed (four colors suffice to color any planar map without any adjacent regions sharing a color), it was proved by a combination of mathematical text and a purpose-built computer program [q.v. Appel and Haken 1977; Appel et al. 1977]. Despite concerns about relying on such 'black box' code, the proof has been grudgingly accepted, and has subsequently been confirmed by Coq, a general-purpose theorem prover [Gonthier 2008].

Second, Robbins' conjecture – that one of the axioms in Huntington's basis for Boolean algebras is equivalent to one of Robbins' axioms – was resolved after 60 years when a solver generated (after running for eight days) a 17-step proof that humans could check manually; fine-tuning found an eight-step proof [McCune 1997].

Third, Kepler's conjecture about optimal sphere packing was reduced, by the mid-20th century, to an exploration of about $5{,}000$ possible packings. Hales originally solved that by minimizing a 150-variable function on each packing, generating some $100{,}000$ linear programming problems [Hales 2005]. As 12 human referees could only – after a five-year review – report that they were '99% certain' the proof was correct, Hales set out to use the HOL Light general-purpose theorem prover in the same way that Coq had been used to prove the four-color map theorem. The Flyspeck project [Hales et al. 2015] was successfully completed in 2014.

However, mechanized reasoning has not yet been as widely adopted in pure mathematics as some of its advocates initially hoped. One reason for this seems to be the quantity and diversity of mathematical knowledge that must be encoded to support modern, research-level mathematics. Once encoded, however, the libraries are available for use in other applications. Thus, the better developed theorem provers have good support for real and complex analysis, Euclidean geometry, combinatorics, and basic algebraic structures (including monoids and groups).

Outside pure, research-level mathematics, mechanized reasoners have had clearer successes in software and hardware verification, which both require more routine mathematics, and whose costs of failures can easily justify the additional effort required to reach formal levels of verification. Hardware verification views computer chips as sets of Boolean statements – including, for example, AND, OR, XOR gates. A chip may then be viewed as defining a logical universe within which theorems may be proved [Harrison 2006a]. Most famously, Intel has used theorem provers since the mid-1990s, following an embarrassing and costly recall resulting from the discovery that one of its chips did not correctly implement the IEEE floating point division standard.

Similarly, any computer program defines a logical universe in which theorem provers may assess the truth of certain statements: in code controlling automated commuter rail systems, e.g., the theorem that no two trains occupy the same location at the same time has been proven [Woodcock et al. 2009]. With Facebook's 2013 acquisition of Monoidics, a start-up firm applying theorem proving to software code analysis, these techniques may be gaining greater public attention.

We believe that the complexity and scope of many modern auctions is such that the additional assurances provided by mechanized reasoning justify their additional costs.

3. COMBINATORIAL VCG AUCTIONS

This section presents a standard specification of VCG auctions (q.v. Ausubel and Milgrom [2006]). The next section uses this to formally specify a VCG auction in Isabelle. Let the set of *agents* be $\{0, \ldots, N\}$, with agent 0 denoted as the *seller* and the rest as *bidders*. The seller's *endowment* is the set $\Omega \neq \emptyset$ of indivisible goods. An *allocation* partitions the endowment. Denote by X_n agent n's share of the allocation so that $\bigcup_{n=0}^{N} X_n = \Omega$. If $X_n = \emptyset$ then agent n is allocated nothing.[6]

Any given bidder n privately values each subset of the endowment $X \subseteq \Omega$ at $v_n(X)$; we assume $v_n(\emptyset) = 0$. Bidders simultaneously submit *bids* to the seller, $b_n(X), \forall X \subseteq \Omega$. For the purposes of this paper, it suffices to take the bids as primitives.

We assume that the seller seeks to maximize value, as proxied by bids. The seller therefore solves for the *value-maximizing allocation*:

$$X^* \in \underset{X_1,\ldots,X_N}{\arg\max} \sum_{n=1}^{N} b_n(X_n) \text{ s.t. } \bigcup_{n=1}^{N} X_n \subseteq \Omega \text{ and } X_n \cap X_{n'} = \emptyset \text{ for } n \neq n' \tag{1}$$

at prices

$$p_n \equiv \alpha_n - \sum_{m \neq n} b_m(X_m^*) \tag{2}$$

where

$$\alpha_n \equiv \max_{\substack{X_m \\ m=1,\ldots,N, m \neq n}} \left\{ \sum_{m \neq n} b_m(X_m) \;\middle|\; \bigcup_{m \neq n} X_m \subseteq \Omega \text{ and } X_m \cap X_{m'} = \emptyset \text{ for } m \neq m' \right\} \tag{3}$$

is the value generated by the value maximization problem when solved without n's bids. Thus, p_n is the opportunity cost of the items won by bidder n: the maximal sum of the goods' value to all bidders other than n, α_n, less the reported value of the items n did not win. Expression (1) is the *winner determination problem* (WDP). When there are multiple solutions X^* we denote the set of them by $\mathcal{X}^*$; a tie-breaking rule must then be used to adjudicate. A typical solution involves assigning random numbers to each possible bundle, denoted by $b'_n(X)$, including the empty set, $b'_n(\emptyset)$. The WDP is then run once more over $\mathcal{X}^*$ to maximize the sum of the random numbers:

$$X^{**} \equiv \underset{X^* \in \mathcal{X}^*}{\arg\max} \sum_{n=1}^{N} b'_n(X_n^*). \tag{4}$$

We conclude by fixing these concepts in the context of a standard example.

Example 3.1 (*[Ausubel and Milgrom 2006, p. 23]*). Assume $N = 3$ bidders submit the following bids for two items, A and B:

$$b_1(AB) = b_2(A) = b_2(B) = b_3(A) = b_3(B) = 2;$$

and $b_n(X) = 0$ for all other $n \in \{1, 2, 3\}$ and item sets $X \subseteq \{A, B\}$. This yields two equivalent value-maximizing allocations; we focus on $X^{**} = (\emptyset, A, B)$, omitting the

[6]Typically, in mathematics $X_n = \emptyset$ would not be an element of a partition on X, whereas in computer science it would.

seller.[7] Then

$$\alpha_1 = \max_{X_2, X_3} \left\{ b_2 (X_2) + b_3 (X_3) \;\middle|\; X_2 + X_3 \subseteq \Omega \right\} = b_2 (A) + b_3 (B) = 2 + 2 = 4;$$

and

$$\alpha_2 = \max_{X_1, X_3} \left\{ b_1 (X_1) + b_3 (X_3) \;\middle|\; X_1 + X_3 \subseteq \Omega \right\} = 2;$$

where α_2 may be determined by either $b_1 (0) + b_3 (B) = 2$ or $b_1 (AB) + b_3 (0) = 2$. Finally, by symmetry, $\alpha_3 = \alpha_2 = 2$. Prices are then

$$p_1 = 4 - [b_2 (X_2^{**}) + b_3 (X_3^{**})] = 4 - [2 + 2] = 0;$$
$$p_2 = 2 - [b_1 (X_1^{**}) + b_3 (X_3^{**})] = 2 - [0 + 2] = 0;$$

and, by symmetry, $p_3 = p_2 = 0$.

In the special case of a single good, the combinatorial Vickrey's auction awards the good to the highest bidder, who pays a price equal to the highest remaining bid; no other bidder pays anything.

4. FORMALLY SPECIFYING A VCG AUCTION

The previous section defined a VCG auction in a way familiar to economists. We now do so in the Isabelle proof assistant which is based on *higher-order logic* (HOL) – a classical logic whose expressions are close to those of paper mathematics. Particular properties of the formal specification can be formally proved to be correct. Here, we prove that our VCG specification defines a function mapping from inputs to unique outcomes, allocates goods no more than once, and assigns non-negative prices to bundles. We cannot, however, formally prove that our formalization faithfully implements the informal definitions: here, human scrutiny must be used.[8]

Translating equations (2) and (4) into Isabelle is conceptually simple: as higher-order logic recognizes functions, anything that needs to be computed can be expressed as a function. Our VCG specification consists of two functions: $vcga$, yielding equation (4)'s final allocation, X^{**}; and $vcgp$, yielding equation (2)'s prices, p_n.

Each of these functions take four arguments: a set of bidders (corresponding to N in (1)); a list of goods (corresponding to Ω in (1)); a bid vector (corresponding to $\boldsymbol{b}$ in (1)); and a random number r (used to obtain the randomized bid vector $\boldsymbol{b}'$ occurring in (4)).

To construct $vcga$ and $vcgp$ in Isabelle, we first split equations (1) and (2) into simpler parts, which we also write as functions.

For example, $\arg\max_{X_1, \ldots, X_N} \sum_{n=1}^{N} b_n (X_n)$ found in (1) is decomposed into the functions $\arg\max$ and $\sum_{n=1}^{N} b_n (X_n)$, each of which has immediate counterparts in the Isabelle library. Denoting their composition by f (written as $argmax \circ setsum$ in Isabelle), running a VCG auction consists of applying f twice to a set of allocations, given some vectors of bids (the actual bids, $\boldsymbol{b}$, initially, then the random $\boldsymbol{b}'$):

$$\mathcal{X} \overset{f(\boldsymbol{b})}{\mapsto} \underset{\subseteq \mathcal{X}}{\mathcal{X}^*} \overset{f(\boldsymbol{b}')}{\mapsto} \underset{\subseteq \mathcal{X}^*}{\{X^{**}\}} \tag{5}$$

In the next subsection, we present a proof that appropriate choice of $\boldsymbol{b}'$, the randomized bid vector, can guarantee that $\{X^{**}\}$ is a singleton. We apply f twice, even when $\mathcal{X}^*$ is a singleton, to avoid reasoning with case analyses.

[7] We use $(\emptyset, A, B)$ as a shorthand notation for $\boldsymbol{x}^* = \{\emptyset, \{A\}, \{B\}\}$.

[8] In our case, in addition to our formalization having the properties that we prove in this section, we can prove Vickrey's theorem on it, and replicate known examples.

It remains to specify $\mathcal{X}$ and $\boldsymbol{b}'$. The former is the set of feasible allocations, or the admissible tuples $(X_1, \ldots, X_N)$ satisfying the "such that" requirements of expression (1). It, therefore, depends on N and Ω alone. We implement it by the function $allAllocations$.

To obtain $\mathcal{X}^*$, we compute $allAllocations$ on the set of participants $\{0, \ldots, N\}$.[9]

abbreviation "$vcgas\ N\ \Omega\ b\ r\ ==\ Outside\{seller\}$‘
$((argmax \circ setsum)\ (randomBids\ N\ \Omega\ b\ r)$
$((argmax \circ setsum)\ b\ (allAllocations\ (\{seller\} \cup N)\ (set\ \Omega))))$" (6)

The first term in the abbreviation defines a function, $vcgas$ which takes N, Ω, b and r as arguments; this notation is common to *functional programming* languages, such as Isabelle. The function is computed from the innermost term, the third line in this case, which applies the f operator mentioned above to return the set of value-maximizing allocations, given bid vector $\boldsymbol{b}$. The function next computes the second line. This applies the f operator a second time: now, the set of feasible allocations is that produced in the third line, while the bid vector is a random vector, $\boldsymbol{b}'$, generated by the $randomBids$ function from a random natural number r. Its construction, which we now explain, guarantees that this second WDP yields exactly one solution.[10]

Let the natural number r be drawn from a sufficiently large domain that each realization represents a unique permutation of the set of bidders (σ_N) and of the powerset of goods (σ_Ω). Then, $b'_n(X) \equiv 2^{|\Omega| \sigma_N(n)} \sigma_\Omega(X)$. Thus, sums of $b'_n(X_n)$ over allocations yield $(N+1)$-digit numbers in base $2^{|\Omega|}$, with the n^{th} digit uniquely indexing the powerset of goods. Such sums therefore uniquely encode allocations, ensuring a unique outcome to the WDP when $\boldsymbol{b}'$ is constructed from r in this way.

Finally, the first line of the $vcgas$ function removes the seller's allocation from the calculations, to correspond to paper definitions (1) and (3), whose expressions are in $\bigcup_{n=1}^{N} X_n \subseteq \Omega$ rather than $\bigcup_{n=0}^{N} X_n = \Omega$. This is done by means of the $Outside\{seller\}$‘ operator, which restricts the domain of its argument to the complement of the singleton $\{seller\}$; the ‘ infix symbol applies this restriction to each element of the argument set.

Thus, $vcgas$ returns the set $\{\mathcal{X}^{**}\}$, from which $vcga$ extracts its unique element, $\mathcal{X}^{**}$:

abbreviation "$vcga\ N\ \Omega\ b\ r == the\_elem(vcgas\ N\ \Omega\ b\ r)$" . (7)

This unique element is extracted by means of the Isabelle function, the_elem.

The VCG pricing function, $vcgp$, has a more direct definition, a straightforward Isabelle translation of (2):

abbreviation "$vcgp\ N\ \Omega\ b\ r\ n\ ==$
$Max\ (setsum\ b\ ‘\ (soldAllocations\ (N - \{n\})\ (set\ \Omega))) - (setsum\ b\ (vcga\ N\ \Omega\ b\ r\ --\ n))$". (8)

The notation $N - \{n\}$ denotes set subtraction; the $--$ removes a single element from the domain of a function or relation. Thus, $vcga\ N\ \Omega\ b\ r\ --\ n$ is the winning allocation without considering participant n.

The **term** command allows us to confirm that the definitions $vcga$ and $vcgp$ return the expected type of object:

term "$vcga\ (N :: participant\ set)\ (\Omega :: good\ list)(b :: bidvector)\ (random :: nat)$"

Isabelle's output is :: "$(nat \times nat\ set)\ set$", which is the right type for an allocation: a function whose argument is of type a (concretely nat) which yields a value of type b

[9] If we had defined allocations to exclude the seller, it would be harder to use mathematical theorems about partitions.

[10] See Theorem $vcgaDefiniteness$, below.

(concretely $nat\ set$) has type $(a\ \times\ b)\ set$; an allocation is a function associating to each participant (type nat) a subset of the goods (type $nat\ set$, a set of natural numbers).

Establishing more sophisticated properties of defined auctions typically requires more effort. The remainder of this section demonstrates how to establish three such properties in Isabelle.

4.1. VCG auctions are functions

One basic desideratum of an auction is that it associates a unique output to each feasible input:

THEOREM 4.1. *Consider a combinatorial VCG auction. Given any set of goods, any set of feasible bid vectors,* $\{b_n\,(X)\}_{n=0}^{N}$*, and a random number* r*, then there is exactly one solution to the winner determination problem* (4) *at prices* p_n *as defined in equation* (2)*.*

Failure of this theorem to hold would clearly be problematic, implying either that admissible bidding did not allow the auction's successful completion, or that different runs of the auction might – given some fixed input – allocate the same goods to different bidders at different prices.

In Isabelle – as in any functional language – attempting to define a function (e.g. $vcga$) causes the system to check that the object can be confirmed to be a function; the definition is accepted only if this is so. Beyond obvious misspecifications (e.g. syntax errors), a definition will not be accepted if it fails to terminate (e.g. provides an endless recursion), or if it fails to associate a uniquely determined value to an admissible input. As $vcgas$ is a total function, it maps from every element of its domain, and hence always computes a unique result. the_elem is meaningful only if its argument is a singleton; when it is not, Isabelle cannot determine the function's value and hence cannot reason about it.[11] That is, in order to show that the definition (7) defines a total function it is necessary to prove that $vcgas$ is a singleton.

In Isabelle, we state this condition as:

theorem vcgaDefiniteness :
  assumes "$distinct\ \Omega$" **and** "$set\ \Omega \neq \{\}$" **and** "$finite\ N$"
  shows "$card\ (vcgas\ N\ \Omega\ b\ r) = 1$"

The "$distinct$" keyword states that the list representing the set of goods, Ω, contains each good exactly once: this allows it to contain multiple versions of the same good, but grants each version a unique identifier.

This is easily proven:

```
1 proof−
2   have "card ((argmax ∘ setsum) (randomBids N Ω b r)
3     ((argmax ∘ setsum) b (allAllocations (N ∪ seller) (set Ω)))) = 1"
4     (is "card ?X = _") using assms lm08 by blast
5   moreover have "(Outside'{seller}) ` ?X = vcgas N Ω b r" by blast
6   ultimately show ?thesis using cardOneImageCardOne by blast
7 qed
```

The proof's structure is typical of many in Isabelle. The **proof** keyword begins the proof. Invoked alone, Isabelle would automatically select inference rules to apply; **proof**– performs manual inference. The **have** ... **using** ... **by** statements structure the ensuing proof: **have** asserts the expressions to be proved; **using** introduces the facts to be used in discharging the proof obligation; **by** invokes a specified proof method.

[11] By contrast, the "$card$" function – introduced below – is defined to return the cardinality of finite sets, but returns 0 for empty and infinite sets.

Lines 2 and 3 claim that the cardinality of the set of solutions to the second WDP (prior to removing the seller's allocation) is one (q.v. the second and third lines of abbreviation (6), above). Line 4 introduces an abbreviation labeled by $?X$ and establishes the statement by applying a proof method called $blast$ to the assumptions (called $assms$) of the theorem and a lemma ($lm08$) previously proved. From this cardinality result, in line 5 an intermediate result – for any function g, $g`A$ is a singleton whenever A is; with g corresponding to $Outside\{seller\}$ – is established by again using the $blast$ method. The **ultimately** keyword in line 6 refers to results established in the previous lines prefixed by **moreover**; the **show** keyword informs Isabelle that we next seek to establish $?thesis$, the proof obligation at the proof's current level of reasoning.

The $blast$ tool manipulates 'simple' objects in higher-order logic [Nipkow 2013]. Here, Lemma $cardOneImageCardOne$ quantifies over all functions g and sets A. While such quantification is inherently higher-order, the proof of the present theorem only requires manipulation of a specific g and a specific A – which can be expressed and manipulated in first-order logic (FOL). FOL is less expressive than HOL but has a *complete* calculus: if an FOL formula follows from premises, then that formula can be derived from those premises. This, in principle, eases automation such as that performed by $blast$. However, FOL's calculus is not *decidable*, but only *semi-decidable*.[12] Thus, in practice, $blast$ either succeeds, fails, or runs until the user cancels it.

This establishes that our formalization returns a unique allocation given bids and a natural number. A similar theorem, labeled $vcgpDefiniteness$, establishes the same result for payments. We now demonstrate that it only allocates the available goods.

4.2. VCG allocations are pairwise disjoint

The second property that we establish is that the outcome allocates all goods at most once, and allocates nothing else:

THEOREM 4.2. *Consider a combinatorial VCG auction. Then the sets $X_1^{**}, \ldots, X_N^{**}$ (as defined at the start of Section 3) are pairwise disjoint.*

THEOREM 4.3. *Consider a combinatorial VCG auction. Then $g \in X_m^{**}$ implies $g \in \Omega$.*

The Isabelle formalization of Theorem 4.2 is:

theorem PairwiseDisjointAllocations :
assumes "$distinct\ \Omega$" **and** "$set\ \Omega \neq \{\}$" **and** "$finite\ N$" **and** "$n1 \neq n2$"
shows "$(vcga'\ N\ \Omega\ b\ r),,n1 \cap (vcga'\ N\ \Omega\ b\ r),,n2 = \{\}$"

This formalization explicitly makes four assumptions on N and Ω which are tacitly assumed in the paper version (Theorem 4.2). First, is the "$distinct$" keyword, which has already been explained above. Second, Ω contains at least one good. Third, the number of participants is finite. The fourth assumption means that the bidders are not the same. Under these assumptions the theorem states that the intersection of the sets of goods allocated to the two participants is empty.

The Isabelle proof of Theorem 4.2 reads:

proof−
have "$vcga'\ N\ \Omega\ b\ r\ \in allocationsUniverse$"
using $vcgaIsAllocationAllocatingGoodsOnly\ assms$ **by** $blast$
then show $?thesis$ **using** $allocationDisjoint\ assms$ **by** $fast$
qed

[12] A calculus is called *decidable* if a procedure exists to decide whether an arbitrary formula follows from arbitrary premises. It is called *semi-decidable* if it does so only if the formula actually follows from the premises, but may not be able to show that it does not if the formula actually does not follow.

The proof's first step uses an existing lemma, *vcgaIsAllocationAllocatingGoodsOnly*, to certify that *vcga′ N Ω b r* belongs to *allocations Universe*, the set of feasible allocations.[13]

Second, the proof invokes another lemma, *allocationDisjoint*, which states that *any* allocation has the pairwise-disjoint property. This step is proven by the *fast* proof tool. As their names suggest, *blast* and *fast* are related: the former, intended as an improvement on the latter, handles a wider range of problems and is generally faster; the latter, however, is simpler, so does occasionally perform better [Paulson 1999].

Finally, Theorem 4.3 becomes:

theorem OnlyGoodsAllocated :
assumes "*distinct* Ω" **and** "*set* Ω ≠ {}" **and** "*finite N*" **and** "*g* ∈ ((*vcgaN* Ω *b r*), , , *m*)"
shows "*g* ∈ *set* Ω"

The proof uses a more general lemma stating that any allocation belonging to *soldAllocations X Y* has a range that is a subset of the powerset of *Y*. Since *vcga N Ω b r* belongs to *soldAllocations N* (*set Ω*), this implies that the union of its range is a subset of *Ω*. Since, by assumption, *g* is in an element of that range, this yields the thesis.

4.3. Prices in VCG auctions are non-negative

The final property that we establish is that VCG prices are non-negative:

THEOREM 4.4. *For a combinatorial VCG auction as defined above,* $\forall n \in N,\ p_n \geq 0$ *(as defined in expression* (2)*).*

The translation into Isabelle is again straightforward:

theorem *NonnegPrices* :
assumes "*distinct* Ω" **and** "*set* Ω ≠ {}" **and** "*finite N*"
shows "*vcgp N* Ω *b r n* ≥ (0 :: *price*)"

The theorem's last line defines 0 to have type *price* in order to prevent Isabelle from mistakenly inferring it to have a different, but still syntactically correct type (e.g. the least element of a bounded lattice). Presently, *price* is an alias for *real*; changing the definition of *price* would allow us to, for example, restrict prices to be natural numbers.

Intuitively, the proof is straightforward. By equation (2), we want to show that the minuend, α_n, is not smaller than the subtrahend, $\sum_{m \neq n} b_m\left(X^{**}_m\right)$. As α_n is the maximum of a sum of bids over a set of possible allocations, and $\{X^{**}_m\}_{m \neq n}$ belongs to that set, we are done.

In Isabelle, this becomes:

```
1 proof −
2   let ?a = "vcga N Ω b r" let ?A = soldAllocations let ?f = "setsum b"
3   have "?a ∈ ?A N (set Ω)" using assms by (rule onlyGoodsAreAllocated)
4   then have "?a −− n ∈ ?A (N − {n}) (set Ω)" by (rule lm099)
5   moreover have "finite (?A (N − {n}) (set Ω))"
      using assms(3) soldAllocationsFinite finite_set finite_Diff by blast
6   ultimately have "Max (?f`(?A (N − {n}) (set Ω))) ≥ ?f (?a −− n)"
      by (rule maxLemma)
7   then show ?thesis by linarith
8 qed
```

[13] Some of our reasoning about allocations does not depend on N and Ω; in these cases, we may work with the generic *allocationsUniverse*, the set of all injective functions which have a partition as their range. Our Isabelle code contains other such generic definitions, identified by the suffix *Universe*.

Line 2 of the proof defines abbreviations, which are introduced by the identifier **let**. Line 3 applies Lemma *onlyGoodsAreAllocated* to establish that the result of $vcga$ is an allocation. Line 4 states that restricting the allocation to $N \setminus \{n\}$ or, in Isabelle parlance, $?a - -n$, yields an allocation on the remaining bidders of the same set of goods.

Line 5 then states that the set of all allocations restricted to the remaining bidders is finite. We already have *soldAllocationsFinite*, stating that the set obtained by applying the function *soldAllocations* to any pair of arguments is finite as soon as the arguments themselves are finite sets. Hence, to use *soldAllocationsFinite*, we must in turn establish that $N - \{n\}$ and $set\ \Omega$ are both finite. $assms(3)$ is the third assumption in the current theorem's statement (N is finite), while *finite_Diff* states that the difference between a finite set and any set is finite: together, they yield that $N - \{n\}$ is finite. The finiteness of $set\ \Omega$ is obtained directly from *finite_set*, which says that the function set applied to any list (in our case, Ω), is finite. These facts are passed to $blast$ by listing them after the **using** keyword; the order in which they are listed does not matter.

In line 4 we established that the set $?A\ (N - \{n\})\ (set\ \Omega)$ includes $?a - -n$. Hence, the maximum of a real valued function (such as $?f$) over this set will be at least as big as the function applied to an arbitrary element of the set, such as line 6's $?a - -n$. Line 6's **ultimately** keyword is similar to **then**, in that it allows to use the preceding line's statement to justify the current step. However, while **then** can refer only to the single preceding line, **ultimately** combines those preceding lines prefixed by the **moreover** keyword, together with the line immediately preceding the first **moreover**. Thus, here, the results of lines 4 and 5 are used as assumptions for line 6's $maxLemma$, which establishes that the maximum of a function (in our case, $?f$) over a set (in our case, $?A\ (N - \{n\})\ (set\ \Omega)$) is not less than the value of that function taken in any element of that set (in our case, $?a - -n$). As the price is the difference between the two terms in line 6, line 7 produces the result by simple arithmetic.

5. EXTRACTING VERIFIED EXECUTABLE CODE

This section explains how executable code can be automatically extracted from the Isabelle specification on which the preceding theorems have been proved.

Traditionally, the design and the implementation of auctions have been done in two separate stages: an auction designer develops the auction principles and performs preliminary tests on them before handing them on to a programmer for implementation. This creates the possibility of translation errors: the programmer may misunderstand the design, may make decisions in the case of an underspecification which contravene the designer's intentions, or may introduce programming bugs. These problems are essentially eliminated by our approach: an automatic procedure translates from Isabelle into a target executable language. Insofar as the code performing this is general, and has been applied to a range of problems, it is more trustworthy than a one-off translation.[14] Against this, Isabelle's code generation features are newer than its HOL core, and have – therefore – been less thoroughly scrutinized, but confidence in them increases with each application.

To support the extraction of executable code, the original Isabelle specification must be written in a *constructive*, rather than a classical, logic. To illustrate the difference between classical and constructive proofs, consider the theorem that there are two irrational numbers a and b such that a^b is rational. A classical proof goes as follows: if $\sqrt{2}^{\sqrt{2}}$ is rational, the theorem holds as $a = b = \sqrt{2}$, which are irrational; if, on the other hand, $\sqrt{2}^{\sqrt{2}}$ is irrational, then the theorem also holds as $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$ are

[14] See Haftmann and Nipkow [2010] for a more detailed description, which includes proofs of partial correctness (to certify that, if the executable code terminates, it faithfully implements the Isabelle definition).

irrational, and $a^b = 2$ is rational. This classical proof establishes the existence of satisfactory numbers a and b, but does not provide a means – or algorithm – of constructing them. By contrast, a constructive proof does not allow proofs by contradiction and the law of the excluded middle; thus, it would require explicit construction of a and b.

In our case, *vcga* and *vcgp* cannot be used to extract code, as they both use the non-constructive *allAllocations*. (The other objects used, such as *argmax*, *setsum*, *the_elem*, and *Max*, are constructive.) We therefore develop two, parallel, definitions for *allAllocations*: the classical definition eases the previous section's theorem proving; the constructive one allows code extraction.

The classical definition of *allAllocations* relies, in turn, on two non-computable terms, namely *all_partitions* and *injections*:

"$allAllocations\ N\ \Omega =$
$Union\{\{a^{-1}|a.\ a \in injections\ Y\ N\}|Y.\ Y \in all\_partitions\ \Omega\}$".

For expositional clarity's sake, we illustrate the process of developing parallel definitions for *injections*, rather than the slightly more complex *all_partitions*. The set $injections\ Y\ N$ contains all the injections (one-to-one functions) defined on the whole of Y which yield values in N.

In turn, the classical definition of the set of injections from a set X to a set Y is

"$injections\ X\ Y = \{R.\ DomainR = X\ \wedge\ Range\ R \subseteq Y\ \wedge\ runiq\ R\ \wedge\ runiq(R^{-1})\}$".

This defines a set of objects, R, with the following properties: R is right unique ($runiq\ R$), so that any element in the domain of R is associated with at most one element in its range; further, the inverse operation is also right unique ($runiq\ (R^{-1})$); finally, R is defined on the whole of X and yields values in Y. The definition is not constructive: it merely states properties of injections between X and Y.

The function, fun *injections_alg*, constructively defines the same entity for finite X and Y:[15]

fun *injections_alg*
where "$injections\_alg\ []\ Y = [\{\}]$" |
"$injections\_alg\ (x\#xs)\ Y = concat\ [[R + *\{(x, y)\}.$
$y \leftarrow sorted\_list\_of\_set(Y - Range\ R)].\ R \leftarrow injections\_alg\ xs\ Y]$"

This works by recursion on X, distinguishing between the base case (empty X) and the step case (non-empty X). As sets' elements are not ordered, we identify the first element in the non-empty X – to be added by the recursive step – by representing X as a list; concrete lists are denoted in Isabelle by square brackets. The two cases are separated by "|".

The base case addresses $[]\ Y$, the set of all injections from the empty list to Y, which is defined to be a single-element list containing only the empty function, "$[\{\}]$". In the step case, X is "$x\#xs$", obtained by prepending an element x to a given list xs. The recursive step uses the set of all injections defined on xs – assumed to have already been computed – to build that on the enhanced list. Its final term, "$R \leftarrow injections\_alg\ xs\ Y$" iteratively passes to R each injective function from the list of all injections from xs to Y. Next, these functions are extended by using the $+*$ operator to add pairs (x, y) to R to obtain functions from X to Y. In order to compute all injective functions, this addition is subject to (denoted by ".") the requirement that we consider all possible $y \in Y$, but which are not in the range of R (denoted $Y - Range\ R$). The *sorted_list_of_set* term converts the (unordered) set to a list to allow computation.

[15]This provides a second motive for maintaining classical definitions: in addition to their more explicit statements of the desired properties of objects, they are not restricted to finite sets.

Maintaining dual definitions requires that we supply Isabelle with *bridging theorems* proving that the classical and the constructive definitions are equivalent when restricted to the finite case. We require two, one for injections and the other for partitions.[16] We discuss in detail the bridging theorem for injections.

After restricting sets to be finite, we seek to prove that

"$set\ (injections\_alg\ X\ Y) =\ injections\ (set\ X)\ Y$".

The left hand side of the thesis presents the computable definition, which is responsible for generating the list of possible injections from X to Y; having done so, the set function converts the list so generated to a set. The right hand side presents the classical definition, which is defined on sets. As we do not generally require that Y is a list, we define it as a set; X, defined as a list, is converted to a set for conformity to the classical definition. In general, lists are better for computing as their elements are inherently ordered, making them easier to parse. Sets, on the other hand, are often better for reasoning and theorem proving as they are closer to paper mathematics, and more general (e.g. they can be infinite).

The proof works by structural induction on the first argument, X.[17] This means that we want to apply the general lemma:

lemma *structInduct* : **assumes** "$P\ []$" **and** "$\forall x\ xs.\ P(xs)\ \longrightarrow\ P(x\#xs)$"
shows "$P\ L$".

The lemma establishes that if a predicate P holds for the empty list (the base case), and if it holds for a list extended by a single element, $x\#xs$, when it holds for the original list, xs, (the induction step), then P holds for any list, L. Hence, the task is now to identify a suitable P. We choose:

"$P\ L\ \longleftrightarrow\ (distinct\ L\ \longrightarrow\ (set\ (injections\_alg\ L\ Y) = injections\ (set\ L)\ Y))$".

This reads: given a list, L, define a predicate P such that, iff the list L has distinct elements, then the sets produced by the two definitions are identical. The property $distinct\ L$ makes sure that the list representing the set contains each element of the set only once.

Hence, our bridging theorem for injections, establishing the equivalence of the classical and constructive definitions in the finite case, is:

theorem *injections_equiv* : **assumes** "$finite\ Y$" **and** "$distinct\ X$"
shows "$set\ (injections\_alg\ X\ Y) =\ injections\ (set\ X)\ Y$"

The Isabelle proof is:

```
1 proof−
2   let ?P = "λL. (distinct L ⟶ (set (injections_alg L Y) = injections (set L) Y))"
3   have "?P []" using injectionsFromEmptyAreEmpty list.set(1) lm099 by metis
4   moreover have "∀x xs. ?P xs ⟶ ?P (x#xs)"
      using assms(1) lm101 by (metis distinct.simps(2) list.simps(15))
5   ultimately have "?P X" by (rule structInduct)
6   then show ?thesis using assms(2) by presburger
7 qed
```

Intuitively, the main step in the proof establishes that $injections\_alg\ (x\#xs)\ Y$ contains all the new injections from the set $x\#xs$ into Y which are added when

[16] The prevalence of these objects in mathematics increases the likelihood that our parallel definitions and their corresponding bridging theorems will be reused by other Isabelle coders.

[17] The more familiar induction occurs over natural numbers. In this case, that would restrict induction to range over the length of the list, rather than over the list itself.

the original set, xs, is augmented by x to become $x\#xs$. In Isabelle, the proof's first step defines the right recursion predicate (line 2). The second step (line 3) proves the base case which, looking at the definition of $?P$, reduces to showing that $set\ (injections\_alg\ []\ Y) = injections\ (set\ [])\ Y$. We employed Isabelle's Sledgehammer tool [Blanchette and Paulson 2014] to suggest methods for doing so. It suggested chaining together the equalities "$set\ (injections\_alg\ []\ Z) = \{\{\}\}$" (established in Lemma *injectionsFromEmptyAreEmpty*) and "$injections\ \{\}\ Y = \{\{\}\}$" (established in Lemma *lm099*), which yields the desired result, once one recognizes that $set\ [] = \{\}$; this last fact is provided by the lemma $list.set(1)$[18], also suggested by Sledgehammer. Sledgehammer's job is over as soon as it suggests the strategy for a particular step: hence, in the final proofs, it is not possible to discern whether the code has been input by the user or generated by Sledgehammer.

The third step proves the induction step (line 4). We apply an auxiliary lemma, *lm101*, which proves the induction step passing from xs to $x\#xs$. Here, Sledgehammer was not only able to suggest proof tools, but also suggested auxiliary lemmas needed to apply *lm101* to this particular step. Here, we know (from the definition of $?P$) that the components of $x\#xs$ are distinct, while to trigger *lm101* we require that x is different from any element of xs and that the components of xs are distinct. As these two conditions are trivially equivalent, Sledgehammer recommended using $distinct.simps(2)$ (the list $x\#xs$ is distinct if and only if xs is distinct and does not include x) and $list.simps(15)$ (the elements of the list $x\#xs$ are the elements of xs, plus x) as an alternative to manually supplying the details.

Now, line 5 applies *structInduct*, establishing that our predicate is characterized by the implication that a list with distinct items implies the equality between the two definitions of injections. Thus, it remains for line 6 to apply *modus ponens* (if p implies q and if p is true, then so is q), to strip out the implication and leave the equality. We initially used the **try0** command, which – like *sledgehammer* – tries a range of methods to find the most appropriate. In this case, it suggested *presburger*, which performs operations in Presburger arithmetic (a simpler arithmetic than Peano's, insofar as it lacks multiplication). While *modus ponens* is not an operation in Presburger arithmetic, it was placed in the *presburger* tool in order to simplify arithmetic expressions.

By replacing the classical definitions of *injections* and *all_partitions* with their constructive counterparts within *vcga* and *vcgp*, we obtain Isabelle functions that constructively define a VCG auction, allowing extraction of Scala code from them; the bridging theorems ensure that the executable code faithfully implements the original sound specification.[19]

6. AUCTION THEORY TOOLBOX

An advantage of mechanized reasoning is that the objects defined may be used with equal facility in other applications. We have therefore initiated an auction theory toolbox (ATT) to assist in the formalization and verification of other auction designs. The ATT code base is freely available at https://github.com/formare. It currently contains the formalizations for the present paper, as well as specifications of a single-good Vickrey auction, and formalizations of Vickrey's theorem developed for Kerber et al. [2014].

[18] Any Isabelle label can be prefixed by a dot and the name of the object or file it refers to. Additionally, some Isabelle theorems can be grouped together, in which case a number in brackets will be appended to the label to refer to a particular theorem in a group. The labels $list.set(1)$, and $distinct.simps(2)$ and $list.simps(15)$ employ both these features.

[19] See https://github.com/formare/auctions/blob/master/scala/VCG.scala for the code.

All proofs can be checked within seconds on a standard laptop. The toolbox also contains the Scala code generated by our constructive definitions.[20]

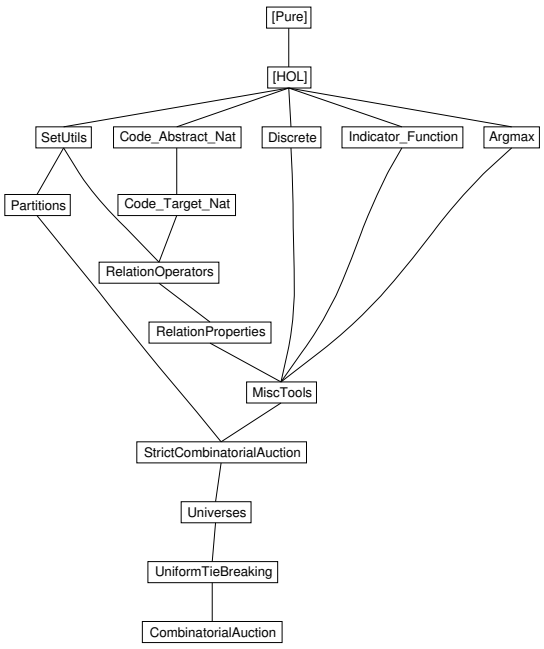

Figure 2 presents the structure of our Isabelle code. The theory files `[Pure]`, `[HOL]`, `Code_Abstract_Nat`, `Discrete`, `Indicator_Function`, and `Code_Target_Nat` come with Isabelle. `SetUtils` (size 6kB) contains some results in set theory (e.g. theorems about sets with at most one element and injective functions). `Argmax` (8kB) defines the maximal argument of a function mapping to a linearly ordered set. The theories `Partitions` (38kB), `RelationOperators` (8kB), and `RelationProperties` (10kB) are self-explanatory. `MiscTools` (42kB) contains theorems which combine entities from the parent theories shown feeding into it. An example is lemma $lm23$, stating that for two finite sets X and Y with "$card(X \cap Y) = card(X)$" it follows that "$X \subseteq Y$".

Figure 2: Theory graph generated by Isabelle

The final four theory files are auction specific, and use the set theoretical, list based, and number theoretic tools formalized in the above. `StrictCombinatorialAuction` (5kB) contains combinatorial auctions, including classical (set theoretical) and computational properties (such as winner and price determination, excluding tie breaking). General properties of the set of all possible allocations are proved in the theory `Universes` (52kB). `UniformTieBreaking` (43kB) introduces and proves properties of `randomBids` as described in Section 4. Finally, `CombinatorialAuction` (40kB) proves the main theorems described in Section 4.

7. RELATED WORK

The three categories of work most closely related to that in this paper are: mechanized theorem proving on auctions; verifying executable auction code; and confirming that an auction design is well specified by model checking.

As regards theorem proving on auctions, Lange et al. [2013] presented a comparative study of proving Vickrey's theorem for single-good auctions in four different systems. [Kerber et al. 2014] is a complementary paper, presenting the Isabelle implementation of Vickrey's theorem in greater depth.

[20]Interested readers can interact with the Scala code at http://www.cs.bham.ac.uk/research/projects/formare/vcg.html.

As to verifying executable code, the consequences of an auction's failure have been known for some time. As yet, however, only limited steps have been taken towards formally verifying executable auction code. For example, the Smith Institute for industrial mathematics and system engineering has "assessed for correctness the software implementations of [. . .] algorithms" used in UK spectrum auctions.[21] The results have not been published, but this work seems to involve traditional methods of running performance tests on test cases. The Smith Institute's Robert Leese has called for auction software to be added to the Verified Software Repository [Woodcock et al. 2009].

The third category, model checking, involves automated, exhaustive searches over finite state spaces; thus, it requires less expert human guidance [Dennis et al. 2012]. Tadjouddine et al. [2009] used SPIN, a widely-used commercial model checker to verify Vickrey auctions' strategy-proofness property.[22] As bids are real numbers, the authors discretized bid values (e.g. i's bid exceeds its valuation, $b_i > v_i$) to obtain finite state spaces. They manually proved the soundness of this reduction. They are thereby able to verify strategy-proofness for any number of agents in a Vickrey auction in a quarter of a second on a desktop computer.

Dennis et al. [2012] developed open source model checking tools, using small auctions as their primary test cases. In a simple auction in which between three and five bidders submit sealed bids to an auctioneer, who awards a good to the highest bidder, they formally verified that the bidder with the highest bid will eventually believe that it has won the auction [Webster et al. 2009]. With just four bidders, each restricted to making one of three possible bids, this property took over an hour to verify.

8. CONCLUSIONS

We have presented a step towards applying mechanized reasoning to the design and practice of auctions. Specifically, we confirmed a number of soundness properties in the context of combinatorial Vickrey auctions: they are functions from their input (bids) to their outcome (allocation and transfers); they allocate all the goods exactly once, and nothing else; prices are non-negative. We have also generated verified executable code that faithfully implements a combinatorial Vickrey auction with the above properties.

Specifying and implementing the auction from within a single mechanized reasoning system both reduces the risk of error and eases extension of the results to other auction designs. For example, our VCG formalizations are easily modified to formalize a first price auction: the WDP is again specified by expression (1); the prices are not determined by equation (2) but by

$$p_n \equiv b_n \left(X_n^* \right). \tag{9}$$

Correspondingly, we replace $vcgp$ with:

abbreviation "$firstPriceP \;\; N\ \Omega\ b\ r\ n == b\ (n, winningAllocationAlg\ N\ \Omega\ r\ b, , n)$"

The theorems in Sections 4.1 and 4.2 carry through directly. The price non-negativity theorem in section 4.3 is now so trivial that Isabelle finds the proof by itself:

lemma assumes "$\forall X.\ b\ (n, X) \geq 0$" **shows**
"$firstPriceP\ N\ \Omega\ b\ r\ n\ \geq\ 0$" **using** $assms$ **by** $blast$.

In this case, the simplicity of the proof implies that it does not need intermediate steps: hence, there is no need of the keywords **proof** and **qed** to enclose them.

[21] q.v. http://www.smithinst.co.uk/case-study/verifying-algorithms/

[22] SPIN is based on a linear temporal logic (LTL), which models states in a linear fashion, thus excluding the consideration of multiple possible future states. Kamp's theorem established the equivalence of LTL to a first-order logic.

In closing, we note two extensions of the present work. First, the present work is intended as a proof of concept, rather than as operationalizable. Thus, our algorithms' computational efficiency must be improved before our code can be used in real auctions: on a standard laptop, it handles two-good examples in less than two seconds, and three-good in 10 seconds, but takes over 12 minutes to handle a four-good example. Improving these times will require both standard software engineering techniques, as well as awareness of methods for efficiently addressing the winner-determination problem, which is NP-hard [q.v. Cramton et al. 2006, Part III]. Second, dynamic features must be added to the ATT, allowing bids (or feasible sets of bids) to be updated over time, and feedback received by bidders. This work is already underway.

Modern auctions are becoming increasingly complex: the FCC's forthcoming 'incentive auctions' are widely regarded as unprecedented in their complexity Federal Communications Commission [2014, pp. 158–167]. Thus, the challenge of ensuring that auctions are soundly specified and implemented is more pressing than ever. We hope that, as the tools presented here develop, they will assist in doing so, both for auctions, and in economics and finance more generally.

ACKNOWLEDGMENTS

We are grateful to Peter Cramton, the Isabelle user community (in particular Tobias Nipkow, Makarius Wenzel, and Jasmin Christian Blanchette), Rob Arthan, Elizabeth Baldwin, and Paul Klemperer for their insights and guidance; Rowat thanks Birkbeck for its hospitality.

References

APPEL, K. AND HAKEN, W. 1977. Every planar map is four colorable part I: Discharging. *Illinois Journal of Mathematics 21,* 3, 429–490.

APPEL, K., HAKEN, W., AND KOCH, J. 1977. Every planar map is four colorable part II: Reducibility. *Illinois Journal of Mathematics 21,* 3, 491–567.

AUSUBEL, L. M. AND MILGROM, P. 2006. The lovely but lonely Vickrey auction. In *Combinatorial auctions*, P. Cramton, Y. Shoham, and R. Steinberg, Eds. MIT Press, chapter 1, 17–40.

AYGÜN, O. AND SÖNMEZ, T. 2013. Matching with contracts: comment. *American Economic Review 103,* 5, 2050–2051.

BARENDREGT, H. AND BARENDSEN, E. 1997. Autarkic computations in formal proofs. *Journal of Automated Reasoning 28*, 2002.

BLANCHETTE, J. C. AND PAULSON, L. C. 2014. *Hammering Away*.

BUXTON, J. N. AND RANDELL, B. 1970. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. NATO Science Committee; available from Scientific Affairs Division, NATO.

CRAMTON, P., SHOHAM, Y., AND STEINBERG, R., Eds. 2006. *Combinatorial auctions*. MIT Press.

DENNIS, L. A., FISHER, M., WEBSTER, M. P., AND BORDINI, R. H. 2012. Model checking agent programming languages. *Automated software engineering 19,* 1, 5–63.

FEDERAL COMMUNICATIONS COMMISSION. 2014. Comment sought on competitive bidding procedures for broadcast incentive auction 1000, including auctions 1001 and 1002. Public Notice FCC 14-191, Washington, D.C. December.

GONTHIER, G. 2008. Formal proof – the four color theorem. *Notices of the AMS 55,* 11, 1382–1393.

HAFTMANN, F. AND NIPKOW, T. 2010. Code generation via higher-order rewrite systems. In *Functional and Logic Programming*. Springer, 103–117.

HALES, T. ET AL. 2015. A formal proof of the Kepler conjecture. *arXiv preprint arXiv:1501.02155*.

HALES, T. C. 2005. A proof of the Kepler conjecture. *Annals of Mathematics 162,* 3, 1063–1185.

HARRISON, J. 2006a. Floating-point verification using theorem proving. In *Formal Methods for Hardware Verification*, M. Bernardo and A. Cimatti, Eds. Number 3965 in Lecture Notes in Computer Science. Springer Verlag, 211–242.

HARRISON, J. 2006b. Towards self-verification of HOL Light. In *Proceedings of the third International Joint Conference, IJCAR 2006*, U. Furbach and N. Shankar, Eds. Lecture Notes in Computer Science Series, vol. 4130. Springer-Verlag, Seattle, WA, 177–191.

HATFIELD, J. W. AND MILGROM, P. R. 2005. Matching with contracts. *American Economic Review 95,* 4, 913–935.

KERBER, M., LANGE, C., AND ROWAT, C. 2014. An introduction to mechanized reasoning. unpublished.

KIRILENKO, A. A. AND LO, A. W. 2013. Moore's law versus Murphy's law: Algorithmic trading and its discontents. *Journal of Economic Perspectives 27,* 2, 51–72.

LANGE, C., CAMINATI, M. B., KERBER, M., MOSSAKOWSKI, T., ROWAT, C., WENZEL, M., AND WINDSTEIGER, W. 2013. A qualitative comparison of the suitability of four theorem provers for basic auction theory. In *Intelligent Computer Mathematics*, J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger, Eds. Number 7961 in Lecture Notes in Computer Science. Springer, 200–215.

LEIBNIZ, G. W. 1686. Projet et essais pour arriver à quelque certitude pour finir une bonne partie des disputes et pour avancer l'art d'inventer. In *Logik-Texte: Kommentierte Auswahl zur Geschichte der modernen Logik*, K. Berka and L. Kreisler, Eds. Akademie-Verlag, Berlin, Deutschland, Chapter I.1, 15–17.

LEYTON-BROWN, K. AND SHOHAM, Y. 2006. A test suite for combinatorial auctions. In *Combinatorial auctions*, P. Cramton, Y. Shoham, and R. Steinberg, Eds. MIT Press, chapter 18, 451–478.

MCCUNE, W. 1997. Solution of the Robbins problem. *Journal of Automated Reasoning 19,* 3, 263–276.

MUHIT, A. 2014. The slow adoption of functional programming in banks. http://oxfordknight.co.uk/blog/oxford-knight-articles/slow-adoption-functional-programming-banks/.

NIPKOW, T. 2013. *Programming and Proving in Isabelle/HOL*.

PAULSON, L. C. 1999. A generic tableau prover and its integration with Isabelle. Tech. rep., Computer Laboratory, University of Cambridge, England.

PAULSON, L. C. 2014. *Isabelle's Logics*.

TADJOUDDINE, E. M., GUERIN, F., AND VASCONCELOS, W. 2009. Abstracting and verifying strategy-proofness for auction mechanisms. In *Declarative Agent Languages and Technologies VI*, M. Baldoni, T. C. Son, M. B. van Riemsdijk, and M. Winikoff, Eds. Number 5397. Springer Verlag, 197–214.

THE GOVERNMENT OFFICE FOR SCIENCE. 2012. Foresight: The future of computer trading in financial markets. Final project report, London.

WEBSTER, M. P., DENNIS, L., AND FISHER, M. 2009. Model-checking auctions, coalitions and trust. Technical Report ULCS-09-004, Department of Computer Science, University of Liverpool.

WILSON, R. 2005. *Four colors suffice: how the map problem was solved*. Princeton University Press.

WOODCOCK, J., LARSEN, P. G., BICARREGUI, J., AND FITZGERALD, J. 2009. Formal method: practice and experience. *ACM Computing Surveys 41,* 4, 1–40.