Constructing a Haskell Framework for Auction Design

Brandon Smart
School of Computer Science
University of Adelaide
a1743623@student.adelaide.edu.au

Dr. Mingyu Guo
School of Computer Science
University of Adelaide
mingyu.guo@adelaide.edu.au

Abstract

In the field of auction theory, auctions are often designed to exhibit important properties such as strategy proofness and individual rationality. However, current methods of proving auction properties are complex and highly sensitive to small changes in the auction design. The automated testing procedures available to software engineers are not available to auction designers, making it unnecessarily difficult to identify simple cases that falsify properties.

We develop a Haskell library that provides a framework for representing arbitrary auction designs, automatically generates test cases likely to disprove properties and identifies simple failing cases to aid designers in the understanding of their system.

1. Introduction and Motivation

Auction design is a complex process as it requires resolving situations where self-interested bidders may act against the public good. This makes forming definitive conclusions about how bidders act challenging, and makes forming conclusions on how to design auctions that incentivise truth-telling similarly difficult. The goals of an auction designer (to maximise profit and maximise social welfare) are often at odds with the goals of bidders (to maximise their own profit, potentially by misrepresenting themselves).

In the field of software engineering, software testing is the primary method used by developers to establish confidence in the correctness of their software. Auction designers want to ensure the 'correctness' of their designs by asserting that their auction has a number of useful properties, and then testing to see if those properties are present. Yet, no current system allows for sufficiently robust testing of such properties. We propose a framework for auction design that facilitates the creation and testing of general auction designs.

Auction design is an important problem as auctions are the primary mechanism used to sell goods of undefined value. Combinatorial auctions for spectrum licenses in the United States raised 40 billion dollars for the Federal Communications Commission from 1994 to 2001, and companies such as Google use a form a second-price auction to sell advertisements on their search pages, generating billions of dollars of income [11]. Aiding the ability of auction designers to test and implement their designs would allow for faster iteration and improvement on proposed changes, and from a technical viewpoint it requires the development of intelligent test generation systems and robust models for auctions.

The aim of the system is to provide three key pieces of functionality. It should provide a generalised method of representing arbitrary auction designs in Haskell, it should be able to generate test cases that are likely to cause a property to be invalidated, and it should allow for the identification of the most simple failing test cases possible, aiding the understanding auction designers have about their systems. We choose to implement our system in Haskell, building off of the Haskell type system and functional programming paradigms.

2. Literature Review

As mentioned, the proposed system would allow for the representation of arbitrary auction designs, facilitate the testing of those designs, and aid in the identification of simple failing test cases. Due to the highly specialised nature of the proposed system, no direct counterpart to it exists in the literature. This is despite wide research into mechanism design (see Section 2.3) as well as research into automated test generation and property validation (see Section 2.4).

However, we can decompose the system into the three core components we have previously mentioned. These areas are much more commonly researched, and the literature provides insight into how these components may be developed. We begin the literature review by discussing the relevance of our system in the wider literature and previous testing suites for auctions, before discussing the literature on how to build these components.

2.1. Auction Design Generation and Need for Testing Software

Many new areas of research into auction design, such as the application of deep learning to auction design tasks by Dütting et al. [4], involve the generation of potentially invalid or non-optimal mechanisms. For these methods to prove viable, and to increase the confidence of auction-runners in the auction design, effective testing mechanisms are needed to show that the generated mechanisms are correct and optimal.

Many other systems for automated mechanism design exist in the literature, such as the systems proposed by Sandholm [16] and Conitzer [2]. These methods are primarily concerned with the maximisation of social welfare in systems, and impose constraints on the set of feasible designs such as strategy-proofness and individual rationality. However, it can be expensive for these systems to impose such constraints on mechanisms. In these applications, a testing mechanism could be used to test if a proposed mechanism has the desired properties without needing to impose them all as constraints.

2.2. Existing Testing Systems in Game Theory and Mechanism Design

Currently in the literature, no general testing frameworks facilitate the testing of arbitrary auction designs. Methods for the automatic assessment of general mechanisms and games exist, but are limited in their capabilities. Systems such as Gambit [9] are designed to allow for the assessment of simple games, but are limited to games with non-continuous strategy spaces and do not provide meaningful systems for reasoning about payment and agent valuations.

Combinatorial auction designs are very well-studied, and specific methods have been developed for assessing their value. Current testing mechanisms for combinatorial auctions such as CATS proposed by Leyton-Brown et al. [8], and the methods cited within are effective for combinatorial auctions, particularly in assessing how optimally they distribute goods, due to the utilisation of test generation techniques specific to combinatorial auctions. However the methods used aren't general enough to cover arbitrary auction designs, nor do they provide methods for simplifying failing test cases.

2.3. Representations of Auctions and Other Game Theory Mechanisms

Auctions are most commonly studied as a mechanism design problem [5]. Throughout the literature a general model for representing mechanisms is described, such as by Harris et al. [5] and Mishra [11]. These models describe auctions (and other games of social choice) as a system where each agent has a private 'type' prescribed by nature from a set 'type space'. Each of the agents in a game can then honestly or dishonestly report their type, with the 'mechanism' deciding on an outcome from the reported type information of each agent.

Others have defined more specialised models for modeling auctions and situations were indivisible goods need to be allocated to agents. Myerson [13] propose methods of representing the set of feasible allocations in a combinatorial setting, and calculating the utility of each agent dependent on their satisfied demands. Many researchers suggest representing an outcome as both the allocation of goods, and the monetary value transfered for each agent [10]. An issue with many of these models is that only apply to a subset of auctions, with many features becoming less important when considered in generality.

These papers present mathematical models for auctions, but do not discuss issues that arise when implementing them in software. In particular, it can be difficult to write software that implements the models discussed due to the large number of agent types and behaviours that may be present. Work has been done to implement negotiation mechanisms where software and human agents can communicate, such as [18] and [19]. However the focus of these systems is on distributed communication processes, and not representing arbitrary auction designs and agent type spaces. These issues form the primary challenges that need to be overcome in a successful model.

2.4. Intelligent Identification of Test Cases

The automatic generation of test cases for particular systems is a well studied but complex problem. Howden showed that there is no effective procedure for always generating a finite set of tests that guarantee correctness for all inputs [6]. Thus, the testing procedures we create cannot provide a guarantee of the properties of a mechanism. Among the papers that examine the problem of intelligent generation of tests, we highlight the work of DeMilli and Offutt [3], which focuses on the generation of test cases that aim to show the presence or absence of specific faults in a system. In our proposed model we have a strong understanding of the properties being tested, but no knowledge of the particular mechanisms being studied. Thus, we expect a "fault-based testing" approach to be

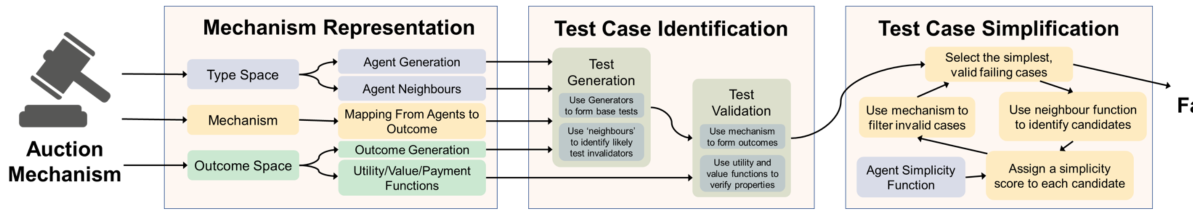

Auction Mechanism
Mechanism Representation
Type Space
Mechanism
Outcome Space
Agent Generation
Agent Neighbours
Mapping From Agents to Outcome
Outcome Generation
Utility/Value/Payment Functions
Test Case Identification
Test Generation
Use Generators to form base tests
Use 'neighbours' to identify likely test invalidators
Test Validation
Use mechanism to form outcomes
Use utility and value functions to verify properties
Test Case Simplification
Select the simplest, valid failing cases
Use mechanism to filter invalid cases
Use neighbour function to identify candidates
Agent Simplicity Function
Assign a simplicity score to each candidate
Simple Failing Test Cases

Figure 1. A high level outline of the representation, testing and simplification process

effective.

There are existing libraries that allow for automatic property-based test generation. QuickCheck is an existing Haskell library that provides many testing procedures for inbuilt Haskell types, and provides functionality that allows users to provide generators for their own types [1]. QuickCheck is the most influential property-based testing library, but alternatives such as SmallCheck for Haskell [15] and Hypothesis for Python [7] exist. The key challenges these libraries face is the possibility of arbitrary types, and the completeness of tests generated.

2.5. Finding Minimum Failing Test Cases

The above methods allow for the identification of failing test cases. However, these test cases may be unnecessarily complicated for human researchers, limiting their ability to understand and gain insight from them. We wish to define a metric that describes the understandability or simplicity of test cases. A common approach in the literature is to define the simplicity of an input test vector as a function of the vectors size, the size of the numeric values, the precision of the floating point numbers and the depth of any data structures used [15].

If we assign this form of numeric 'simplicity score' to each input test vector, then the problem of finding the simplest failing test case is equivalent to finding the maxima in an n-dimensional vector space. This is a very well researched problem in the literature, with common techniques including hillclimbing, simulated annealing, gradient descent and application of genetic algorithms [17] [12] [14]. One of the key challenges with these approaches is finding local maxima rather than the more ideal global maxima. Translating an arbitrary agent type into a numeric value may not be possible in some cases, which may necessitate the use of a 'neighbourhood' search approach like hillclimbing to find maxima.

3. Methodology

We consider the process of identifying simple failing test cases for a mechanism to be a system composed of three core components

- Representation of the Mechanism
- Test Generation and Property Testing
- Simplification of Failing Test Cases

Figure 1 gives a high level overview of the auction representation and testing processes. The decomposition of the problem into these three components can be seen. We describe the methodology used in each component individually.

3.1. Representations of Type Space, Outcome Space and Mechanisms

In order to work with any arbitrary auction design, we decompose auctions into three components: the type space, the outcome space and the core mechanism. These three components align with the usual representation of auctions in the literature (as mentioned in Section 2.3).

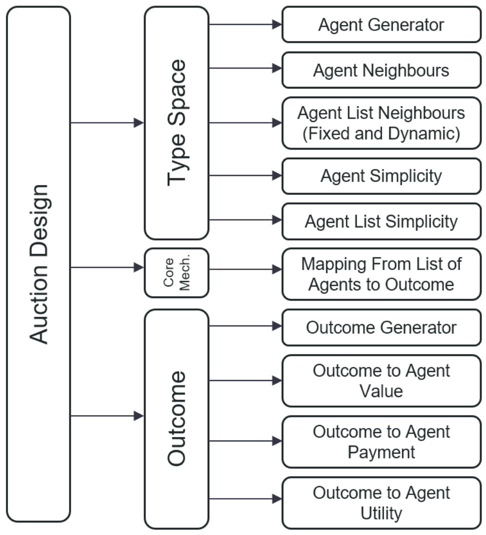


Figure 2. The separation of an auction design into the type space, outcome space and mechanism, and the further decomposition into core functions

The first component is the type space, which is the set of all possible agents that can take part in an auction. We introduce 'MechAgent' as a Haskell type class, asserting that all valid type spaces are represented as instances of MechAgent. Similarly, the outcome space is the set of all possible outcomes to an auction, with valid outcome spaces being represented as instances of MechOutcome (another introduced Haskell type class). The final component is the core mechanism, which is the function that takes the list of reported agents types as an argument, and returns an outcome.

Figure 2 demonstrates the decomposition of an auction design into these three components. The auction design can be successfully run just using just these three components. However to facilitate the testing of properties and the simplification of failing cases, we allow several key functions to be implemented for both the type space and outcome space (described below).

To enable property testing for outcomes, mechanisms need to provide a function that takes in an outcome, the index of an agent, and their true type, and returns that agent's valuation, payment and utility. We intentionally allow users to use any data type/structure they wish to represent their auction in order to facilitate the representation of as many auction types as possible. We restrict our system to interacting with auction outcomes using these three functions.

By using Haskell as the language of this library, we can use Haskell's type inference system to determine which of the functions mentioned in Figure 2 should be used in any given part of our system.

3.2. Properties and the Generation of Test Cases

Our system allows for users to define their own properties for testing their mechanisms. We choose to implement four key properties of interest that are often used to discuss auction designs:

Individual Rationality For any possible set of agents, each agent gets a non-negative utility

Strategy Proofness For any possible set of agents, no agent would get a better utility by reporting themselves as having a different type

Weak Budget Balance For any possible set of agents, the net payment to the auction-runner is non-negative

Efficiency For any possible set of agents, the generated outcome is the one that maximises the net valuation of the agents that took part

These properties are evaluated using the discussed functions for fetching the valuation, payment and utility of agents from a generated outcome.

We build our testing system on top of QuickCheck's property system. Implementations of type spaces and outcome spaces are required to implement 'generators', which are used to generate random agents and outcomes for use in testing. Users can write their own generators in order to facilitate the generation of test cases that are likely to invalidate properties.

For the purposes of this paper, we generate bidder valuations from uniform distributions. However, because custom generators can be introduced for all types, these can be replaced with generators more likely to produce agents of interest.

The properties we have implemented are decomposed into five functions which handle the generation and validation of test cases, simplification of failing test cases, and conversion of a result to a string.

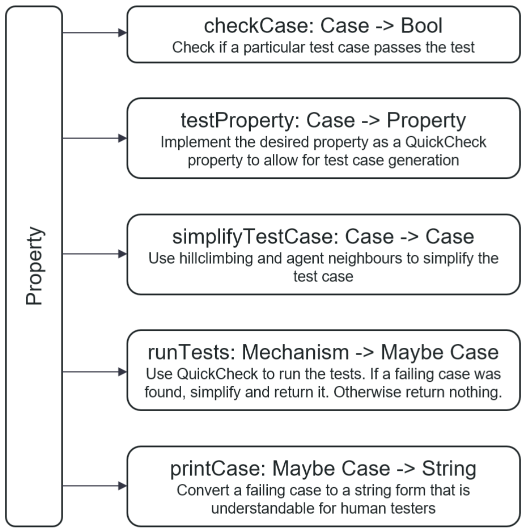

Property
checkCase: Case -> Bool
Check if a particular test case passes the test
testProperty: Case -> Property
Implement the desired property as a QuickCheck property to allow for test case generation
simplifyTestCase: Case -> Case
Use hillclimbing and agent neighbours to simplify the test case
runTests: Mechanism -> Maybe Case
Use QuickCheck to run the tests. If a failing case was found, simplify and return it. Otherwise return nothing.
printCase: Maybe Case -> String
Convert a failing case to a string form that is understandable for human testers

Figure 3. The decomposition of a property into the functions needed for testing, simplification and printing (we use 'Case' and 'Mechanism' as a stand-in for more complex type signatures)

In order to find test cases that are more likely to invalidate properties, we implement a 'neighbouring' function for agents and agent lists (as seen in Figure 2). These functions take in an agent (or a list of agents), and return a slightly modified version of that agent (or list of agents). Because slight deviations often allow properties to be invalidated (such as when an agent attempts to report their type as a slight variation of their true type), we can increase the likelihood that we create a failing case without having to dramatically increase the number of tests.

Below, in Figure 4, is on overview for how the 'runTests' function shown in Figure 3 is implemented using the other property functions. By changing the maximum number of tests we run before termination, we can control the trade-off between time and expected coverage of the test space.

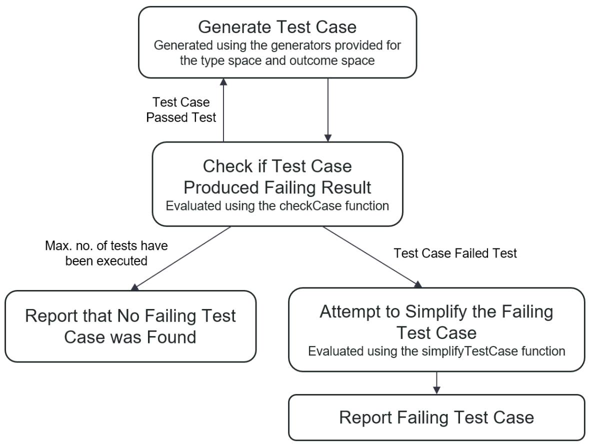

Generate Test Case
Generated using the generators provided for the type space and outcome space
Test Case Passed Test
Check if Test Case Produced Failing Result
Evaluated using the checkCase function
Max. no. of tests have been executed
Test Case Failed Test
Report that No Failing Test Case was Found
Attempt to Simplify the Failing Test Case
Evaluated using the simplifyTestCase function
Report Failing Test Case

Figure 4. The process of identifying a simple failing test case

3.3. Simplification of Test Cases

The final stage of the testing process is the simplification of failing test cases. If no failing case was found, we can simply report that all the tests were passed. However in cases where a failing case is found, it is often the case that it is not easily understandable to human testers. In these cases, we introduce simplicity score functions (as shown in Figure 2) that returns a numerical score representing how simple an agent (or a list of agents) is.

Consider a single item auction, where each agent has a private valuation of the item being sold. Here is a sample graph that might represent how simple that agent is as a function of their valuation.

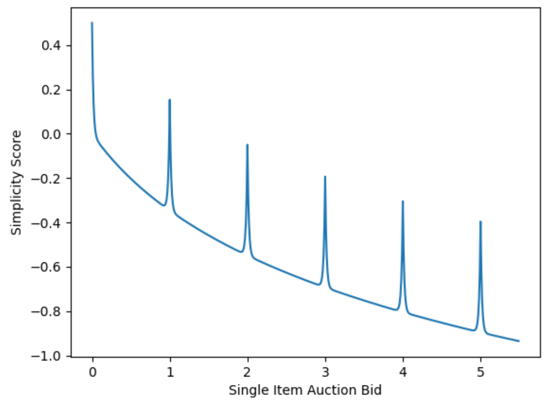


Figure 5. The simplicity score for a single item auction bidder

Here, the simplicity of a bidder is considered a function of how large the bid is, and how close it is to an integer. The reasoning is that human testers will have an easier time understanding a bidder with smaller valuations, but will also have an easier time understanding a bidder with an integer valuation (eg. a bidder with a valuation of 3 is easier to understand than a bidder with valuation 2.783). Agents in more complex auction types (such as combinatorial auctions) will naturally have more complex simplicity functions dependent on how they value each set of items.

Below is an example of the simplicity score of a list of agents. Here, two single item auction bidders are considered.

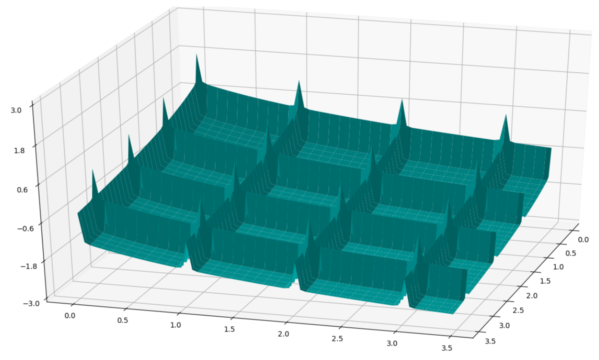


Figure 6. The simplicity score for two single item auction bidders

The goal of simplifying a failing test case is to find the set of agents with the highest simplicity score that is also a failing case for the property being tested. We achieve this by using a hillclimbing algorithm. We start with the failing case identified by the earlier testing steps, which likely has a low simplicity score. We then use the neighbouring functions for agents and sets of agents to permute the test case in a large number of ways (roughly equivalent to moving some distance along the horizontal axes). We then select the new test case with the highest simplicity score that still fails the property test as the new simpler test case. This process is repeated until no more improvements can be made (i.e. a local maxima of the simplicity function has been reached). This process is shown in Figure 7.

Note that both the single agent and multi-agent simplicity functions shown in Figures 5 and 6 have a large number of local maxima aligned with the integer valuations that we do not want the hillclimbing algorithm to settle on. Because the 'steps' of the hillclimbing algorithm are controlled by the generated neighbours for each agent, we can specify how much change the hillclimbing algorithm should attempt in each iteration. For this example, we choose a neighbour function that attempts to substitute the valuation of an agent with:

- Valuations that are ± 0.05 from the original
- Valuations that are ± 1 from the original
- Valuations that are rounded to the nearest integer

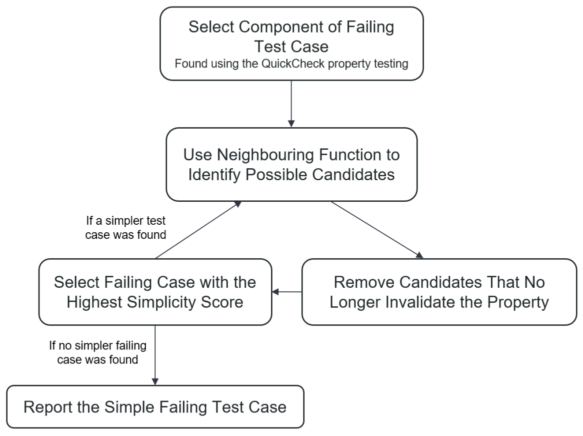

Select Component of Failing Test Case
Found using the QuickCheck property testing
Use Neighbouring Function to Identify Possible Candidates
If a simpler test case was found
Select Failing Case with the Highest Simplicity Score
Remove Candidates That No Longer Invalidate the Property
If no simpler failing case was found
Report the Simple Failing Test Case

Figure 7. The process of hillclimbing to simplify a component of a test case

This prevents the hillclimbing algorithm from becoming stuck at a suboptimal maxima. It should be noted though that this method doesn't guarantee the global maxima is found, particularly for mechanisms/properties with non-continuous failing test case regions. However, because the simplicity scores are a heuristic for human understandability, local maxima are often sufficiently simple for most cases.

For more complex auctions (such as combinatorial auctions), the neighbour and simplicity functions used will be more complex as they have to take into account a larger number of valuations by each agent.

For properties such as strategy proofness, we need to simplify both the set of agents, and the new agent that someone is falsely reporting themself as. To achieve this, we iteratively hillclimb each of these components until a stable configuration has been reached or the maximum number of iterations have been exceeded.

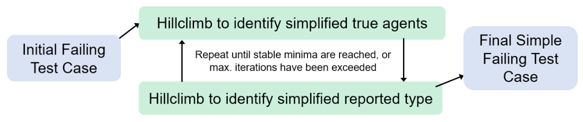

Initial Failing Test Case
Hillclimb to identify simplified true agents
Repeat until stable minima are reached, or max. iterations have been exceeded
Hillclimb to identify simplified reported type
Final Simple Failing Test Case

Figure 8. The process of iterative hillclimbing to simplify the components of a failing case for strategy proofness

4. Results

Here we list the auction designs and properties tested, benchmark our results for accuracy and speed, and discuss challenges and limitations in our design.

4.1. Implemented Auction Designs

For testing our system, we implemented four categories of auction.

The first auction type is the single item auction, where one item is being sold, and each agent reports how much they value that item. We implement three mechanisms for single item auctions:

- First Price Sealed Auction
- Second Price Sealed Auction
- First Agent Always Wins Auction (where the item is always sold the the first agent at their requested price)

The second auction type we implement is the multi-unit auction, where some number of identical items are sold and each agent reports how much they would value owning 'x' copies of that item. We implement three mechanisms for multi-unit auctions:

- Vickrey-Clarke-Groves mechanism (with no free disposal of items)
- Vickrey-Clarke-Groves mechanism (with free disposal of items)
- All to One Auction (where all items are sold to the bidder who is willing to pay the most for the entire set)

The third auction type we implement is the combinatorial auction, where a number of distinct items are being sold and each agent reports how much they would value owning each subset of items. We implement three mechanisms for combinatorial auctions:

- Vickrey-Clarke-Groves mechanism (with no free disposal of items)
- Vickrey-Clarke-Groves mechanism (with free disposal of items)
- Greedy Auction (where the highest valued subset of items is continually sold to the person who values it highest until all items are gone)

The final auction type we implement is the public good auction. There are many different types of public good auctions, each typically having their own unique type and outcome space. We choose to implement a public good auction where agents each have a valuation for how much the public good would be worth to them. We implement two mechanisms for this type of auction:

- Allow All If Funded Auction (if the total valuation of agents is above 80, then charge all agents their valuation and allow access of public good to all agents)

- Rejection Auction (Agents who have a valuation below 20 are not charged, but they do not get access to the public good. If the total valuation of the remaining agents is above 80, then the public good is constructed, and the agents with a valuation above 20 pay their reported valuation and get access to the good)
- Vickrey-Clarke-Groves Auction

All of these auction types were successfully implemented using our system. However the implementation of Vickrey-Clarke-Groves auction designs are non-polynomial (both with respect to time, and the amount of space required to store the bids of each agent). This makes these auction designs slower to run and test.

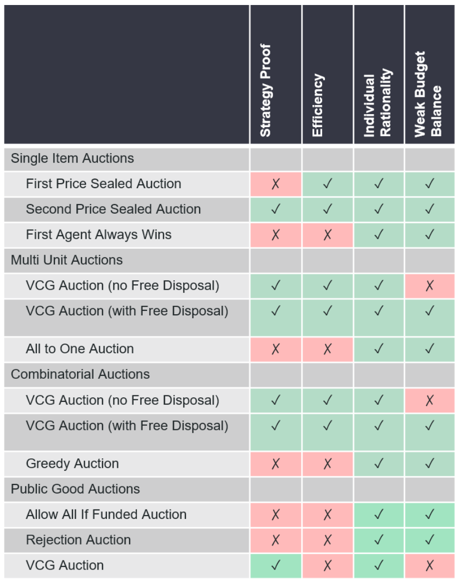
While our system was successful in representing the listed auctions, it can only be used to represent auctions where each agent has a nature-given 'type' that they report to the mechanism, and the mechanism makes all decisions about payment and item assignment. For example, our system cannot be used to represent English/Dutch auction designs, as these auctions have dynamic behaviour (and may involve more complex strategies from agents if the allowable increase in bid is limited).

4.2. Benchmarks for Property Proving

We evaluate the accuracy of our system in identifying the properties of the auctions presented. For each auction/property, we generate 100 tests. Because we are testing for failures, we assume that a mechanism has the desired property if no failing case was found. As such, we expect all failures to be accurate, but our system may falsely identify an auction as having a property it truly does not have, due to a failure to find a counterexample.

Note that we redefine the weak budget balance property for public good auctions to take into account the cost of the public good being built. Also note that we identify building the public good and allowing access to everyone to always be the efficient outcome as it maximises the total valuation of bidders.

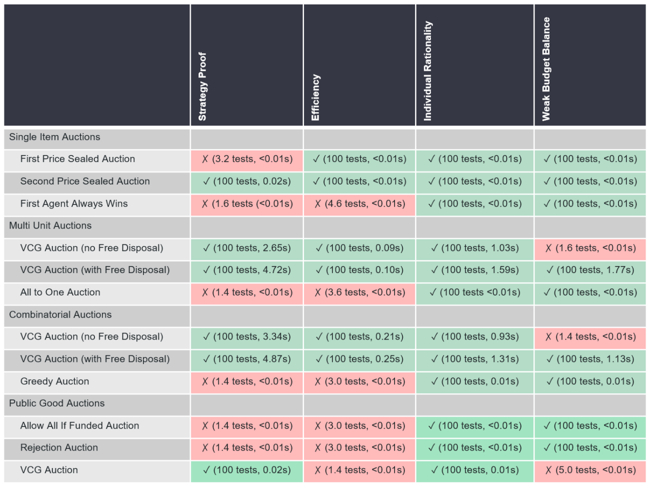
Figure 9 shows our results. Here, we are limiting the number of agents in each auction to between 2 and 8, and we are restricting the number of items in multi-unit auctions to 5 and the number of items in combinatorial auctions to 2. Despite these restrictions, we manage to find the accurate results for all of our tests. This procedure works effectively for these auction designs as failing cases are present in cases with low numbers of bidders and low numbers of items. However, because the size of the test space grows exponentially with the number of bidders for most auction designs, we expect that the accuracy of our system would be challenged by more specialised auctions with rarer failing cases (such as those that only exist when a large number of bidders take part).



	Strategy Proof	Efficiency	Individual Rationality	Weak Budget Balance
Single Item Auctions				
First Price Sealed Auction	✗	✓	✓	✓
Second Price Sealed Auction	✓	✓	✓	✓
First Agent Always Wins	✗	✗	✓	✓
Multi Unit Auctions				
VCG Auction (no Free Disposal)	✓	✓	✓	✗
VCG Auction (with Free Disposal)	✓	✓	✓	✓
All to One Auction	✗	✗	✓	✓
Combinatorial Auctions				
VCG Auction (no Free Disposal)	✓	✓	✓	✗
VCG Auction (with Free Disposal)	✓	✓	✓	✓
Greedy Auction	✗	✗	✓	✓
Public Good Auctions				
Allow All If Funded Auction	✗	✗	✓	✓
Rejection Auction	✗	✗	✓	✓
VCG Auction	✓	✗	✓	✗

Figure 9. The list of implemented auction designs and the results our system assigns them

We also benchmark the average amount of time and number of tests it took to invalidate each property for each mechanism in Figure 10. We imposed the same restrictions on the auction designs here as we did for Figure 9.



	Strategy Proof	Efficiency	Individual Rationality	Weak Budget Balance
Single Item Auctions				
First Price Sealed Auction	✗ (3.2 tests, <0.01s)	✓ (100 tests, <0.01s)	✓ (100 tests, <0.01s)	✓ (100 tests, <0.01s)
Second Price Sealed Auction	✓ (100 tests, 0.02s)	✓ (100 tests, <0.01s)	✓ (100 tests, <0.01s)	✓ (100 tests, <0.01s)
First Agent Always Wins	✗ (1.6 tests (<0.01s)	✗ (4.6 tests, <0.01s)	✓ (100 tests, <0.01s)	✓ (100 tests, <0.01s)
Multi Unit Auctions				
VCG Auction (no Free Disposal)	✓ (100 tests, 2.65s)	✓ (100 tests, 0.09s)	✓ (100 tests, 1.03s)	✗ (1.6 tests, <0.01s)
VCG Auction (with Free Disposal)	✓ (100 tests, 4.72s)	✓ (100 tests, 0.10s)	✓ (100 tests, 1.59s)	✓ (100 tests, 1.77s)
All to One Auction	✗ (1.4 tests, <0.01s)	✗ (3.6 tests, <0.01s)	✓ (100 tests <0.01s)	✓ (100 tests, <0.01s)
Combinatorial Auctions				
VCG Auction (no Free Disposal)	✓ (100 tests, 3.34s)	✓ (100 tests, 0.21s)	✓ (100 tests, 0.93s)	✗ (1.4 tests, <0.01s)
VCG Auction (with Free Disposal)	✓ (100 tests, 4.87s)	✓ (100 tests, 0.25s)	✓ (100 tests, 1.31s)	✓ (100 tests, 1.13s)
Greedy Auction	✗ (1.4 tests, <0.01s)	✗ (3.0 tests, <0.01s)	✓ (100 tests, 0.01s)	✓ (100 tests, 0.01s)
Public Good Auctions				
Allow All If Funded Auction	✗ (1.4 tests, <0.01s)	✗ (3.0 tests, <0.01s)	✓ (100 tests, <0.01s)	✓ (100 tests, <0.01s)
Rejection Auction	✗ (1.4 tests, <0.01s)	✗ (3.0 tests, <0.01s)	✓ (100 tests, <0.01s)	✓ (100 tests, <0.01s)
VCG Auction	✓ (100 tests, 0.02s)	✗ (1.4 tests, <0.01s)	✓ (100 tests, 0.01s)	✗ (5.0 tests, <0.01s)

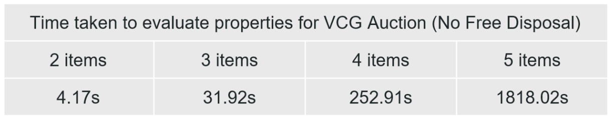
Figure 10. The time and average number of tests it took to disprove each property, or to exhaust 100 tests

Here it can be seen that for simple auction designs with low computational complexity such as single item and public good auctions, the amount of time and the number of tests required to invalidate properties is negligible. However, the amount of time taken for combinatorial auction designs such as Vickrey-Clarke-Groves is much greater, as these are NP-Hard auction designs.

It can be noted that it takes more time for our system to test mechanisms for strategy-proofness, as this requires generating multiple outcomes in order to assess whether any agent benefits by falsely reporting their type. In our results, we also see that efficiency takes less time to test than other properties, which we expect is due to low-level compiler optimisations that aren't being completed for other properties, suggesting that improvements to runtime can be made. Of course, the absolute values for these tests will vary depending on the hardware used to run them.

As mentioned, for combinatorial auctions the size of the test space increases exponentially with respect to both the number of bidders and the number of items being sold. The uniform distribution test generation methods used in the current system for these tasks are ill-equipped for auctions with large numbers of items. Other domain-specific test generation methods (such as those mentioned by Leyton-Brown in CATS [8]) should be used to implement custom generators and focus testing on important parts of the test space. However, despite these drawbacks, the use of neighbouring functions allows us to find accurate results for the auction designs tested.

The table below shows the effect of the number of items being sold in a combinatorial auction on the amount of time it takes our system to test the VCG auction design (with no free disposal of items).



Time taken to evaluate properties for VCG Auction (No Free Disposal)			
2 items	3 items	4 items	5 items
4.17s	31.92s	252.91s	1818.02s

Figure 11. The time it took to assess the properties of a combinatorial auction dependent on the number of items being sold

4.3. Efficacy of Simplification Process

Finally we evaluate the effectiveness of the simplification methods introduced in this system. Simplicity and understandability are subjective to the user, so we highlight some examples of the simplification method working.

We start by showing the simplification process in action in Figure 12, and how a failing case for strategy proofness is simplified to a more human understandable counterexample.

Here, we are extending the neighbouring function for agents in a single item auction to include bidders with valuations of 0 or 1, which reduces the number of iterations required to simplify failing cases.

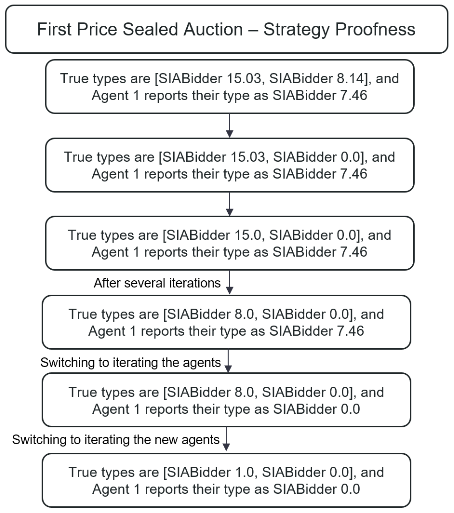

First Price Sealed Auction – Strategy Proofness
True types are [SIABidder 15.03, SIABidder 8.14], and Agent 1 reports their type as SIABidder 7.46
True types are [SIABidder 15.03, SIABidder 0.0], and Agent 1 reports their type as SIABidder 7.46
True types are [SIABidder 15.0, SIABidder 0.0], and Agent 1 reports their type as SIABidder 7.46
After several iterations
True types are [SIABidder 8.0, SIABidder 0.0], and Agent 1 reports their type as SIABidder 7.46
Switching to iterating the agents
True types are [SIABidder 8.0, SIABidder 0.0], and Agent 1 reports their type as SIABidder 0.0
Switching to iterating the new agents
True types are [SIABidder 1.0, SIABidder 0.0], and Agent 1 reports their type as SIABidder 0.0

Figure 12. An example showing the steps that are taken in the process of simplifying a complex failing case for strategy proofness in a first price sealed auction

In Figure 13, we include examples of simplifications made by our system.

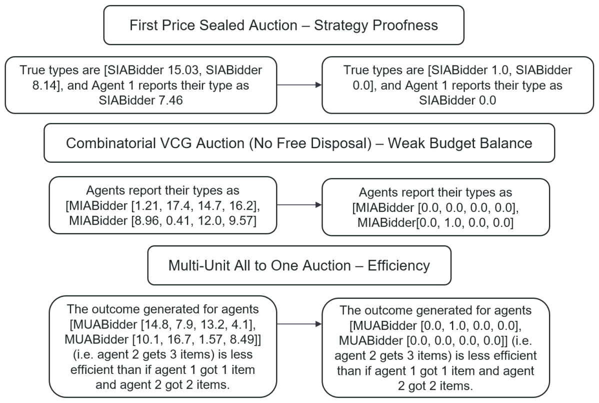

First Price Sealed Auction – Strategy Proofness
True types are [SIABidder 15.03, SIABidder 8.14], and Agent 1 reports their type as SIABidder 7.46
True types are [SIABidder 1.0, SIABidder 0.0], and Agent 1 reports their type as SIABidder 0.0
Combinatorial VCG Auction (No Free Disposal) – Weak Budget Balance
Agents report their types as [MIABidder [1.21, 17.4, 14.7, 16.2], MIABidder [8.96, 0.41, 12.0, 9.57]
Agents report their types as [MIABidder [0.0, 0.0, 0.0, 0.0], MIABidder[0.0, 1.0, 0.0, 0.0]
Multi-Unit All to One Auction – Efficiency
The outcome generated for agents [MUABidder [14.8, 7.9, 13.2, 4.1], MUABidder [10.1, 16.7, 1.57, 8.49]] (i.e. agent 2 gets 3 items) is less efficient than if agent 1 got 1 item and agent 2 got 2 items.
The outcome generated for agents [MUABidder [0.0, 1.0, 0.0, 0.0], MUABidder [0.0, 0.0, 0.0, 0.0]] (i.e. agent 2 gets 3 items) is less efficient than if agent 1 got 1 item and agent 2 got 2 items.

Figure 13. Examples of the simplification process effectively identifying failing test cases that are more easily understandable to human testers

These examples highlight that for these auction types, our simplification processes are effective, as they manage to find the simplest failing test cases available. This is due to the fact that the 'region' of failing test cases is continuous for these auctions, and greedily simplifying failing cases does lead to the global maxima of the simplicity function. In these cases, finding a failing case and hillclimbing to make it simpler is a much more efficient method than doing an exhaustive search of the test space, and selecting the simplest failing case found.

An example of a case where the hillclimbing isn't as effective is in the rejection auction (for a public good) mentioned in Section 4.1. Here, the agents all report their type representing how much they value the public good. Agents who have a valuation below 20 are not charged, but they do not get access to the public good. If the total valuation of the remaining agents is above 80, then the public good is constructed, and the agents with a valuation above 20 pay their reported valuation and get access to the good.

Figure 14 shows an attempt at simplifying a failing test case for this auction.

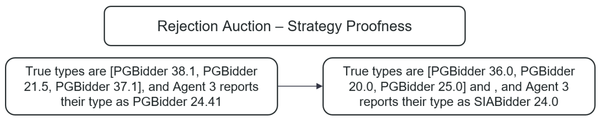

Rejection Auction – Strategy Proofness
True types are [PGBidder 38.1, PGBidder 21.5, PGBidder 37.1], and Agent 3 reports their type as PGBidder 24.41
True types are [PGBidder 36.0, PGBidder 20.0, PGBidder 25.0] and , and Agent 3 reports their type as SIABidder 24.0

Figure 14. An example where hillclimbing fails to simplify a failing case for the rejection auction as much as is desired

Here, the hillclimbing algorithm finds a failing case where no single-agent deviation results in a simpler failing test case. Decreasing the valuation of any agent will result in a passing test case rather than a failing one, and increasing the valuation will increase the complexity of the test case. However, simpler failing test cases could be found by ’transferring’ valuations from one agent to another.

5. Conclusion

The methods introduced here succeed in representing a wide variety of auction mechanisms, and in identifying simple failing test cases.

However, for combinatorial auctions with high dimensionality, our test case generation methods are less efficient, and we would expect reduced accuracy in cases where failing examples require large numbers of items or bidders.

Future research can incorporate existing methods of domain-specific test case generation (such as those introduced by Leyton-Brown [8]). The use of QuickCheck as the foundation of testing proves limiting for efficient testing, and for control over the generation of sets of agents (rather than the generation of single agents), so future systems may wish to be developed using different testing frameworks.

Finally, our system provides no mechanism for evaluating how ’close’ to optimal a system is, as it simply assesses whether or not the system always generates the optimal result. This idea of how close an auction is to the optimal outcome is of key theoretical interest, and future systems may wish to implement this functionality.

References

[1] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[2] V. Conitzer and T. Sandholm. Complexity of mechanism design. In *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence*, pages 103–110. Morgan Kaufmann Publishers Inc., 2002.

[3] R. DeMilli and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.

[4] P. Dütting, Z. Feng, H. Narasimhan, D. C. Parkes, and S. S. Ravindranath. Optimal auctions through deep learning. *arXiv preprint arXiv:1706.03459*, 2017.

[5] M. Harris and A. Raviv. Allocation mechanisms and the design of auctions. *Econometrica: Journal of the Econometric Society*, pages 1477–1499, 1981.

[6] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, (3):208–215, 1976.

[7] HypothesisWorks. Hypothesis. `https://github.com/HypothesisWorks/hypothesis`, 2019.

[8] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76. Citeseer, 2000.

[9] R. D. McKelvey, A. M. McLennan, and T. L. Turocy. Gambit: Software tools for game theory. 2006.

[10] F. M. Menezes and P. K. Monteiro. *An introduction to auction theory*. OUP Oxford, 2005.

[11] D. Mishra. An introduction to mechanism design theory. *The Indian Economic Journal*, 56(2):137–165, 2008.

[12] M. Mitchell, J. H. Holland, and S. Forrest. When will a genetic algorithm outperform hill climbing. In *Advances in neural information processing systems*, pages 51–58, 1994.

[13] R. B. Myerson. Optimal auction design. *Mathematics of operations research*, 6(1):58–73, 1981.

[14] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[15] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Acm sigplan notices*, volume 44, pages 37–48. ACM, 2008.

[16] T. Sandholm. Automated mechanism design: A new application area for search algorithms. In *International Conference on Principles and Practice of Constraint Programming*, pages 19–36. Springer, 2003.

[17] S. S. Skiena. *The algorithm design manual: Text*, volume 1. Springer Science & Business Media, 1998.

[18] M. Wellman, P. R. Wurman, and W. E. Walsh. The michigan internet auctionbot: A configurable auction server for human and software agents. In *Second International Conference on Autonomous Agents (AGENTS)*, pages 301–308, 1998.

[19] P. R. Wurman, M. P. Wellman, W. E. Walsh, and K. A. O’Malley. A control architecture for flexible internet auction servers. In *IBM/IAC Workshop on Internet-Based Negotiation Technology*. Citeseer, 1999.