Rice University

Computer-aided Mechanism Design

by

Ye Fang

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Swarat Chaudhuri, Chair
Assistant Professor of Computer Science

Moshe Vardi
Karen Ostrum George Distinguished
Service Professor in Computational
Engineering

Luay K. Nakhleh
Associate Professor of Computer Science
and Biochemistry and Cell Biology

Christopher Jermaine
Associate Professor of Computer Science

Houston, Texas

May, 2015
ABSTRACT

Computer-aided Mechanism Design

by

Ye Fang

Algorithmic mechanism design, as practised today, is a manual process; however, manual design and reasoning do not scale well with the complexity of design tasks. In this thesis, we study computer-aided mechanism design as an alternative to manual construction and analysis of mechanisms.

In our approach, a mechanism is a program that receives inputs from agents with private preferences, and produces a public output. Rather than programming such a mechanism manually, the human designer writes a high-level partial specification that includes behavioral models of agents and a set of logical correctness requirements (for example, truth-telling) on the desired mechanism. A program synthesis algorithm is now used to automatically search a large space of candidate mechanisms and find one that satisfies the requirements. The algorithm is based on a reduction to automated first-order logic theorem proving — specifically, deciding the satisfiability of quantifier-free formulas in the first-order theory of reals.

We present an implementation of our synthesis approach on top of a Satisfiability Modulo Theories solver. The system is evaluated through several case studies where we automatically synthesize a set of classic mechanisms and their variations, including the Vickrey auction, a multistage auction, a position auction, and a voting mechanism.
Acknowledgments

This thesis would not have been possible without the guidance of my advisors and support from my friends, my family, and fellowship.

I would like to express my sincere gratitude to my advisors Pro. Chaudhuri and Prof. Vardi for their continuous support and invaluable advice during my study and research in Rice. I really appreciate the various times when Pro. Chaudhuri patiently outlined the goals of our research, went through technique difficulties, and revised my writings. Prof. Chaudri and Pro. Vardi have also provided me with many opportunities to go to top conferences and summer schools, getting exposure to the newest ideas.

I want to thank Kuldeep Meel and Dror Fried for their sincere help and inspiring conversations. They have made my research life more enjoyable. I would also like to thank my friend Powei Feng, who was always willing to help edit my writing and presentations. Many thanks to Yuming Zhang, Quiqi Liu, Fred Liu, and Binhang Yuan for their companionship and help during my master study.

Last but not least, I would like to thank my fellowship and family for their support. They have always been there for me, to encourage me, comfort me, trust me, and love me.
Contents

Abstract ii
Acknowledgments iii
List of Illustrations vi

1 Introduction 1
1.1 Motivating Example ............................................. 2
1.2 Previous Approaches ............................................ 3
1.3 Our Contribution ................................................ 6
1.4 Thesis Overview ................................................ 8

2 Background 10
2.1 Preliminaries .................................................... 10
2.2 Background and Motivation .................................... 13
2.3 Related Work .................................................... 17

3 MECHSYNTH Language 22
3.1 MECHSYNTH Language Syntax ............................... 23
3.2 Syntactic Sugar .................................................. 26
3.3 Example Program ............................................... 27

4 Synthesis Algorithm of MECHSYNTH 31
4.1 Synthesis Semantics ............................................. 32
4.2 Decision Procedure ............................................. 36
        4.2.1 Assumptions ............................................. 36
4.2.2 Early termination ........................................... 37
4.2.3 Quantifier Elimination ........................................ 37
4.2.4 Example ......................................................... 39
4.2.5 SMT-solving ..................................................... 40
4.2.6 Correctness ....................................................... 42
4.2.7 Complexity ....................................................... 42

5 Case Studies ........................................................ 44
  5.1 Single-goods auction ........................................... 45
      5.1.1 Variant 1 ................................................... 46
      5.1.2 Variant 2 ................................................... 46
      5.1.3 Variant 3 ................................................... 47
      5.1.4 Variant 4 ................................................... 47
  5.2 Multi-stage single goods auction ............................... 47
  5.3 Multi-units identical goods auction ............................ 49
  5.4 Position auction ................................................ 51
  5.5 Aggregating social choice ...................................... 52

6 Concluding Remarks and Future Work .......................... 54

Bibliography .......................................................... 56
Illustrations

1.1 three bidder single-good sealed-bid auction ............................ 5

3.1 Core language for computer-aided mechanism design. Here, \( A \) is an
agent name; \( \overline{f} \) and \( f \) represent field names; \( \overline{p} \) and \( p \) represent formal
parameters; \( \overline{h} \) and \( h \) represent holes; \( \overline{o} \) represents outputs; and \( m \) and
\( uf \) are function names. ........................................ 24

3.2 Specification for single-goods auction ................................. 28

5.1 Specification for multi-stage single-goods auction ................. 48

5.2 Overall truth-telling ................................................... 48

5.3 Discounted utility ....................................................... 49
Chapter 1

Introduction

Economics mechanism design concerns with the construction of the rules in markets of selfish participants. Market designers aim to achieve some desired outcomes by executing different rules. With the development of Internet, many influential electronic markets have been created, such as Amazon marketplace and keyword auctions (Internet advertisement auctions) [1]. As in real-life markets, participating agents in online economic activities represent different entities with the goal of maximizing their own benefits. However, a straightforward online implementation of real-life markets can be problematic in terms of scale, size, speed, and complexity [2]. These computational issues have led to more complex mechanism design tasks. These challenges brought about the field of algorithmic mechanism design that considers both strategic behavior of agents and computational constraints and efficiency of online implementations. In both real-life and online setting, designers construct rules based on experience and intuitions and verify those rules through formal methods [3]. Though such design methodology has yielded some successful mechanisms that are used in real-life markets and online applications, the design process remains manual and ad-hoc, resulting in low productivity.

Given the increasing demand of online protocols, recent research focuses on developing a more systematic and automated design methodology to improve the productivity of mechanism design.

This thesis makes the first attempt in formalizing mechanisms as programs. Specif-
ically, we provide a high-level specification language for designers to encode problem settings and their insights on the skeleton of mechanisms (a partial mechanism with details to be filled). Then, we apply a program synthesis algorithm to construct a complete mechanism. Our approach exploits the recent improvement in program synthesis techniques [4, 5, 6], which realize the automatic generation of programs from specifications.

In the rest of this chapter, we firstly motivate the mechanism design problem through a simple auction example and then present previous approaches and their solutions to the example auction. Lastly, we formally introduce our approach and our solution to the example problem.

1.1 Motivating Example

In a typical auction, bidders are asked to express their preferences over the auction good through bids. Then, the auctioneer receives those bids and decides on the bid-winner and bid-winning price. Now, consider a programmer tasked with the design of a mechanism for a single good auction [7]. Each agent participating in the auction, i.e. bidder in this case, has a private evaluation of the auctioned good. This private evaluation is the maximum amount that the agent is willing to pay for the auction good. We assume the given system prefers truth-telling, i.e., behavior where each agent bids an amount that is equal to its true evaluation of the good. Concretely, we want to construct auction rules such that agents, each with its private preference, will receive the greatest gain by telling their true preferences no matter how other agents behave.

When analyzing markets in real-life, it is necessary to use game-theoretical approach where agents are assumed to be rational (i.e. their aim is to maximize their
own utilities. For example, in this single good auction, let the agent’s utility function be defined as the amount of gain after the auction. Under this function, a rational agent will bid for a value other than its private evaluation if the agent can gain more utility to do so.

Assuming such strategic behavior of the participating agents, how can we reverse engineer a mechanism where the agents will gain the most utility by revealing their private evaluations?

1.2 Previous Approaches

Both mechanism design and algorithmic mechanism design have traditionally been a manual practice. In a simple problem setting, a designer comes up with several candidate mechanisms and then manually proves if any of them meets the requirement. Alternatively, the designer can formulate both the mechanism design problem and the desired properties mathematically and apply analytical tools to verify candidate solutions [2].

For this single good sealed-bid auction example, the celebrated Vickrey auction [8] was invented where truth-telling is the best strategy. Vickrey auction mechanism chooses the highest bidder as the auction winner and the second highest bid as the price. Given the rational agent model, we can easily prove that truth-telling is the dominant strategy, i.e, the only best strategy.

These manual approaches are actively practiced today, and they have yielded successful applications such as the Vickrey-Clarke-Groves (VCG) mechanism [8, 9, 10] for spectrum allocation [11] and ebay’s proxy bidding. However, this manual practice is not a systematic engineering process since designers use experiences and intuitions to come up with the initial set of candidate solutions. Such design practice results in
that each of these mechanisms is designed only for a small class of settings and for a specific objective. Such trial-and-error production methodology can also be lengthy. Moreover, researchers have no guarantee on the existence of a valid mechanism given a set of formal requirements, which can result in wasted effort.

In [12, 13, 14, 3, 15, 13, 16], Sandhlon and Cortnizer were the first to explore a computer-aided approach to systematically engineer mechanisms, called *automated mechanism design*. This line of work formulates mechanism design as an optimization problem provided with constraints and an objective function. Particularly, a mechanism is a function that maps from the agents’ input space to the outcome space. The designer is assumed to have full knowledge of agents’ input space, and both the input space and output space are assumed to be finite. In addition, game-theoretical properties are formalized mathematically as constraints, and the desired property are represented by an objective function. In the framework as described, the task of mechanism design is to search for a function from input space to output space such that the constraints are satisfied and the objective function is maximized/minimizied.

However, this basic automated design approach cannot help to find a truthful mechanism for the single-good sealed-bid auction given we have no prior knowledge on bidders’ private evaluations of the good, i.e., the input space of the auction mechanism. Moreover, though many automated mechanism design approaches have been studied followed this constraints-based optimization formulation of mechanism design, almost all of them assume prior knowledge of the agents’ private information. Our approach is better because we require no prior information on the mechanism’s input space.

In our motivating example, we would like the mechanism to incentivize bidders to tell their true evaluations regardless of others’ bids. The most promising solution
Maximize: $v_1 - (c_1 b_1 + c_2 b_2 + c_3 b_3)$ (total utility)

Subject to:

For every true evaluation vector $v_1 \geq 0$, $v_2 \geq 0$, $v_3 \geq 0$, and every bid vector $b_1 \geq b_2 \geq b_3 \geq 0$,

$v_1 - (c_1 b_1 + c_2 b_2 + c_3 b_3) \geq 0$ (individual rationality),

$v_1 \geq b_2 \geq b_3 \Rightarrow (v_1 - (c_1 b_1 + c_2 b_2 + c_3 b_3) \leq v_1 - (c_1 b_1 + c_2 b_2 + c_3 b_3)),$

$b_2 \geq v_1 \geq b_3 \Rightarrow (v_1 - (c_1 b_1 + c_2 b_2 + c_3 b_3) \leq v_1 - (c_1 b_2 + c_2 v_1 + c_3 b_3)),$

$b_2 \geq b_3 \geq v_1 \Rightarrow (v_1 - (c_1 b_1 + c_2 b_2 + c_3 b_3) \leq v_1 - (c_1 b_2 + c_2 b_3 + c_3 v_1)),$

$v_2 \geq b_1 \geq b_3 \Rightarrow 0 \leq v_2 - (c_1 v_2 + c_2 b_1 + c_3 b_3),$

$v_3 \geq b_1 \geq b_2 \Rightarrow 0 \leq v_3 - (c_1 v_3 + c_2 b_1 + c_3 b_2)$

Figure 1.1: three bidder single-good sealed-bid auction

in this vein is found in [1]. Guo et al’s method builds on existing mechanisms, and reduces the design space using domain knowledges. One of the mechanisms created in [1] guarantees optimal redistribution in VCG auctions in the worst case, i.e, having no assumption on bidders’ bids. To illustrate the method of Guo et al, let us use a simpler version of the motivating example. Assume the good is allocated to the highest bidder and the price function is restricted to be linear function of bidders’ bids. We also assume that only the winner will be charged. The design problem for a three-player auction can be formulated as in figure 1.1.

Our task is to find the values for $c_1$, $c_2$, $c_3$, but the problem also includes real-valued unknowns $v_1, v_2, v_3, b_1, b_2, b_3$. [1] proved that the redistribution problem can be transformed to a linear programming problem so that the unknown variables can be avoided. However, this proof for redistribution mechanism cannot be applied to
other design problems and cannot be used in automating designs. Therefore, no existing method can help to design a truthful mechanism for the example auction in an automated way.

Though previous automated mechanism design methodologies have shifted some of the burden to computers, they have not realized much automation. Designers need expertise in both optimization and economics in order to translate a mechanism design problem to a constraint-based optimization problem.

Rather than building upon the previous formulation, we approach mechanism design from a different perspective. In this thesis, we regard an economic activity as a game. We model games as programs, and the desired properties as specifications of programs. Explicitly, one round of a game resembles an execution of a program, and a game is said to meet the requirements when all of the outputs from executing the program satisfy predefined specifications. When the mechanism is not given, a game can be encoded as a partial program, i.e., an incomplete program, annotated with specifications [17]. Now, the task of mechanism design is to find an executable piece of code of the missing mechanism function such that the resulting program satisfies predefined specifications.

1.3 Our Contribution

This thesis studies computer-aided mechanism design where the task of constructing a mechanism that meets requirements is regarded as a form of program synthesis [4, 5, 6]: the automatic generation of programs from declarative specifications. Concretely, our synthesis-based approach has two main components: (1) a high-level language for specifying models of agents and requirements on mechanisms; and (2) a synthesis algorithm that generates a concrete mechanism from such a specification.
A game formalized in our language consists of following elements:

- Models of agents that participate in the market, written in a flexible imperative programming language. The dynamics of an agent consists of updates to its local state, as well as invocations of the mechanism in synchrony with other agents.

- A “template” that captures high-level insights about a correct mechanism’s structure and restricts the search for correct mechanisms.

- A “driver” that simulates a game/market by executing the participating agents along with the mechanism and pays off the agents.

- A set of logical requirements — for example, truth-telling or optimality of resource allocation — that constrain the payments made by the mechanism to the agents.

Applying this model to our example, the single-goods auction game can be specified by:

- an agent model that includes a real-valued field for private evaluation, a real-valued field for actual bidding price, and an utility function,

- a template of mechanism function,

- the property of truth-telling as dominant strategy which is annotated as the specification for the output of mechanism function,

- a “driver” which defines how a game is conducted. In this case, the “driver” firstly creates multiple agents, then calls the mechanism function with a set of bids collected from these agents, and finally calculates the winner and price.
Now, the task of the synthesis algorithm, given such a specification, is to find values for the unknown parameters in the mechanism template such that the resulting mechanism satisfies the requirements. This task is NP-hard, but like most recent approaches to automated program analysis, our method circumvents this worst-case hardness by using state-of-the-art techniques for automated logical reasoning (cf. [18]).

Specifically, using a series of logical transformations, our algorithm reduces the search for missing parameters in the mechanism to deciding the satisfiability of a quantifier-free formula in the first-order theory of reals. This task is then solved using a Satisfiability Modulo Theories (SMT) solver [19]. An SMT-solver is a theorem prover for quantifier-free theories of first-order logic that places specialized decision procedures for first-order theories within a larger SAT-solving framework based on DPLL search [20]. The scalability of these solvers has improved dramatically in the last few years and is critical to our approach.

The language and algorithm are implemented in a prototype called MechSynth. Our system is evaluated through several case studies including the Vickrey auction, a multistage auction, a position auction, and a voting mechanism. We have successfully reconstructed the classical mechanisms for these games, and also synthesized new mechanisms for a variety of auctions.

1.4 Thesis Overview

In Chapter 2, we describe the preliminaries needed for further discussion, including the market model and several game-theoretic concepts. We also provide an overview of the manual mechanism design methodology and automated mechanism design.

In Chapter 3, we formally present the syntax of our specification language and show an example program for the single good sealed-bid auction mentioned in this
In Chapter 4, we introduce our synthesis algorithm that includes the synthesis semantics of our language and illustrate a decision procedure created as a part of this thesis.

In Chapter 5, we present several famous mechanisms reconstructed by our system, including second-price auction rule, generalized second-price auction rule, position auction rule used by GoogleAds auction, and linear voting mechanisms. We also explore several variations of the single-good sealed-bids auction by using different agent utility function and equilibrium properties.

Finally, we summarize our contributions and outline future directions in Chapter 6.
Chapter 2

Background

2.1 Preliminaries

Given the generality of preference aggregation market model, this thesis focuses on designing mechanisms for markets of this model, such as auctions and voting. In a preference aggregation setting, a group of agents must jointly make a decision based on their preferences [21]. Preference is a quantified value of how much an agent prefers one candidate. This value is assumed to be private, i.e., only known by agent itself. However, in order to make a final decision to achieve desired properties, decision maker(s) need some protocols to elicit individual agents’ preferences. For example, a good voting rule gives voters incentive to express their true preferences over candidates, and chooses the most socially preferred candidate.

Game Setting

In a computational setting, a preference aggregation game generally includes following elements [3]:

- A mechanism (rule) $M$ defines the follows:
  - a set of feasible actions $X$ that agents can take, say, the kind of messages that agents can sent to the mechanism or among each other.
  - a set of possible outcomes $O$. 
- a mechanism function $M : x_1 \times x_2 \times \cdots \times x_n \to o$, where $x_i \in X_i$ and $o \in O$.

- A set of agents $A = \{1, 2, ..., n\}$. Each agent $i$ has:

  - its private preference, termed as type, $\Theta_i$ with a probability distribution $\gamma_i$ over $\theta_i \in \Theta_i$. In a prior free setting, we assume no pre-knowledge on the agent’s type.
  
  - a set of feasible actions $X_i$.
  
  - a utility function $f : o \times \theta_i \to \mathbb{R}$, where $o \in O$, and $\theta_i \in \Theta_i$.
  
  - a strategy function: $s_i : \theta_i \to x_i$, where $\theta_i \in \Theta_i$ and $x_i \in X_i$. Since a rational agent acts to maximize its utility, the strategy can be characterized as a function.

For example, in a single-goods auction, agent $i$’s type, $\theta_i$, is how much agent $i$ is willing to pay for the auction good; the strategy function calculates how much the agent actually bid based on its type; the outcome $o \in O$ includes the winner of the auction and the price to be charged. Let us assume that agents share a common utility function.

As discussed, a rational agent chooses the strategy that maximizes its expected utility based on its belief of others’ strategies and the knowledge of the rule. So, the mechanism function can be rewritten as:

$$M : s_1(\theta_1) \times s_2(\theta_2) \times \cdots \times s_n(\theta_n) \to o.$$ 

The utility for each agent $a_i$ can also be rewritten as:

$$u_i(M(s_1(\theta_1), s_2(\theta_2), \ldots, s_n(\theta_n), \theta_i) \to \mathbb{R},$$
which indicates the outcome and the amount of utility that each agent receives depend on its type.

**Solution Concept**

The problem arises as given a preference aggregation game, how can we find the best strategy(ies) of participating agents? In *game theory*, *solution concept* is a general principle to predict what strategies will agents use in a game [22]. The best strategies can be viewed as the agent’s solution of a game. For instance, a game is said to implement the solution concept of *dominant strategy*, if there exists only one best strategy for each agent such that:

\[ u_i(M(s_1(\theta_1), \ldots, s_i^*(\theta_i), \ldots, s_n(\theta_n)), \theta_i) \geq u_i(M(s_1(\theta_1), \ldots, s_i(\theta_i), \ldots, s_n(\theta_n)), \theta_i). \]

This solution \( S^* \) is said to be an *equilibrium point* of the game, meaning that no agent \( i \) has incentive to deviate from its solution (best strategy) of the game, \( s_i^* \), no matter what other agents’ actions are. \( S^* \) is also called *equilibrium strategy profile*. \( S_{-i} \) is constructed by excluding agent \( i \)'s strategies from \( S \). The set \( S^* \) here is the solution profile of rational agents found under the concept of *dominant strategy*. Other solution concepts include *Nash equilibrium*, *Bayesian Nash equilibrium*, *Sub-game perfect equilibrium*, and etc.

**The Problem**

In a mechanism design problem, designers are given a game setting, a solution concept, desired properties of the outcome, and, possibly an objective function \( g \). The task of mechanism design is to find the mapping \( m \) from all possible agents’ collective strategies to the outcome space such that when implementing the mechanism function \( m \),
1) we can find solutions (equilibrium strategy profile) of the game when applying provided solution concept,

2) the solutions meet the desired properties (e.g. dominant strategy, equilibrium solution),

3) the outputs maximize or minimize the objective value (if an objective function is provided).

### 2.2 Background and Motivation

**Economic Mechanism Design**

The introduced mechanism design problem was originally studied in economics. The field of mechanism design considers how to construct rules to achieve desired properties for markets involving interactions of rational agents [2]. The most studied properties include optimal seller’s revenue and social welfare (efficiency). Economists combined *game theory* and *social choice theory* to study strategic behavior of rational agents and the decision making process among these agents. Game theory provides a framework to reason the outcomes of a game of rational agents. Explicitly, game theorists aim to find the best strategies of agents who know the rule and have partial (*Bayesian setting*) or complete information of other players’ strategies. The profile of agents’ best strategies is called the equilibrium solution or outcome of a game, i.e., the stable point where no agent has incentive to deviate from its best strategy.

Meanwhile, the field of social choice theory provides us many useful theoretical results that address how preferences from a group of individuals can be aggregated to achieve a collective decision of desired property. To summarize, the essence of mechanism design is to construct such rules that the strategic behaviors of rational players
will reach a desirable equilibrium point. During the last several decades, handful of economic mechanisms are invented, such as the remarkable Vickery auction [8] and Vickrey-Clarke-Groves (VCG) auction [8, 9, 10]. They have brought much influence on real life practices like large privatizations and spectrum allocations [11].

**Algorithmic Mechanism Design**

Since Internet is invented in 90’s, economic activities, such as buying and selling, and auctions, have been largely moved to this online platform [2]. Internet has also given birth to new economic activities, for instance, the keyword auction (Internet advertisement auctions), such as Google Adwords and Yahoo sponsored search [1]. Other notable examples, which involve the allocation/redistribution of resources, include p2p resource sharing network and grid computing.

Contrast to computers in traditional centralized systems, computers connected over Internet are owned by different entities with their own goals. Shoham and Leyton-Brown coined “multiagent system” in [22] to characterize such highly autonomous and distributed systems. The networked computers are called computerized agents because their self-interested behavior resembles that of real-life rational agents. In such systems, we can no longer assume that computerized agents will follow predefined protocols unless the protocols provide agents incentives to do so.

Nisan and Ronen [23] firstly introduced the framework of *mechanism design theory* to study protocols for multiagent systems. In their framework, an mechanism design setting for computerized agents generally includes:

- $N$ computerized agents (e.g. online proxy bidders), each of whom has private information $t^i \in T^i$. 
• an output function \( \text{alloc}() \) which maps a set of collective agents’ type, \( t = \langle t^1, \ldots, t^n \rangle \), to a set of outputs \( o \in O \).

• the utility function \( u_i \) for each agent \( a_i \) (in an anonymous setting, agents are assumed to have a common utility function). The utility function measures the preferences of agents over the possible output of the algorithm.

Moreover, payment is used as part of the mechanism to create incentives. Now, let \( A_i \) denotes the set of strategies available to agent \( i \), then the outcome is \( o = \text{alloc}(a_1, \ldots, a_n) \), and the payment for agent \( i \) is \( p_i = \text{pay}(a_1, \ldots, a_n) \). If for any agent \( i, a_i = t_i \), we say agent \( i \) tells the truth. Usually an objective function \( g \) is provided, which maps the outcome \( o \) and agent type vector \( t \) to a real value. Then, the designer will choose the candidate mechanism that maximizes the value of \( g \). An example of the goal function can be the total value of individual’s utility after an auction. A mechanism function is said to achieve the most economic efficiency when it maximizes the collective utility.

However, problems quickly arose as mechanisms hold nice game-theoretical properties are not always computationally tractable. For example, [24] mentions that in the combinatorial auction where goods are bid and sold in bundles, the value of each bundle of goods for each agent is, most likely, a nonlinear function of the number of goods in the bundle. In the worst case, the space required to express bidders’ valuation is exponential in terms of the total number of goods. So the computational implementation of VCG mechanism has to address the problem of space efficiency.

Therefore, the field of algorithmic mechanism design emerged. Researchers started to exploit algorithmic tools, such as randomization, approximation, and worst/average-case analysis, to construct computationally implementable mechanisms. Alterna-
tively, assumptions are made to trade the generality of mechanisms for computational efficiency. Examples of these assumptions include [2]: whether money transfer is included in the mechanism; what kind of solution concept is used; Is the mechanism to achieve efficient allocation of resource or maximum revenue of individual.

The new algorithms and protocols yield from the field of algorithmic mechanism design have been vastly implemented in online applications. For example, Ebay uses social learning techniques to set the reserve price for single item auction to maximize seller’s expected revenue. This mechanism is based on Myerson’s single item auction, while remedies the assumed pre-knowledge of agents’ private information by learning from history data of similar items.

However, algorithmic mechanism design and analysis remains a manual process. Several downsides of the manual design can be easily identified:

- The most famous classic mechanisms, (for example, VCG and dAGVA), only promise to render certain property, for example truth-telling (veracity), optimal resource allocation, and privacy. However, in practice, a designer may want to assert variations or relaxations of these properties, or achieve several properties at the same time.

- Most of the algorithm still use side payments as part of the mechanism to achieve truth-revealing, while money transfer is not implementable in many real settings such as barter-based electronic marketplaces [3].

- Manual reasoning can only accommodate mathematically clean models of agent behavior while agents in real-world markets may not abide by demands of mathematical simplicity. Frameworks, which provides the flexibility for agent model modeling, are needed to study more complicated system behaviors.
• Manual design method cannot circumvent the situation where mechanism does not exist for certain properties. This will lead to unnecessary investment in research. For example, Arrow proves in [25] that there does not exist any “fair” voting mechanism satisfying the three criterions defined in the paper. Arrow’s impossibility theorem is explained in chapter 5.

2.3 Related Work

Automated Mechanism Design

Sandholm and Conitzer [12, 13, 14, 3, 15, 13, 16] introduces the first computational model for automated mechanism design, which is described in Section 2.1. In general, they formulate the problem as to search for a function $f$ that maps the collective agents’ type space to the outcome space. The function needs to satisfy predefined constraints, such as incentive compatibility, i.e., truth-telling is the dominate strategy, while optimizes the specified objective value (e.g. social welfare, expected revenue).

In [13], Sandholm and Conitzer apply this technique to several small instances of problems, like divorce-settlement and combinatorial auctions. A comprehensive study of computational issues in automated preference-aggregation mechanism design can be found in [26]. This framework has been applied to automate the design process of several other game settings. Guo gives an overview of the applications in his dissertation [1].

As promised, this original automated mechanism design methodology has shifted some of the burden from human designers to computers, and rendered a clean formulation and solution of the problem. However, it cannot be applied to larger game settings (e.g. multiple players, larger type spaces, etc). The reason is that the number
of parameters in the corresponding linear programming instances grows exponentially with the increasing number of agents, agents’ types and possible outcomes. Moreover, this approach requires the agents type space and the outcome space to be discrete, which restricts the application of this method to small number of game settings.

This automated design method has also been applied to finding specific families of parameterized mechanisms, which are defined by a set of parameters. In the case of designing parameterized mechanisms, the original problem of finding a function that optimizes the objective value reduces to the problem of searching for optimal values of the set of parameters that define the mechanism. The spirit of this approach, reducing the mechanism design to the search of a small set of parameters, is very similar to the idea of the methodology in this thesis. We ask designers to write a high-level specification, i.e, a partially written program, aiming to reduce the search space for the synthesizer. Examples of the parameterized mechanisms includes the family of Affine Maximizer Auction and Virtual Valuations Combinatorial Auctions [27, 28], efficiency-maximizing mechanisms for multi-unit auctions subjects to a minimal revenue constraints [29], the family of shared-good auctions with different objective functions [30], and a collection of VCG based redistribution mechanisms summarized in [1].

Other automated mechanism design methods include heuristic mechanism design [31] and incremental mechanism design [32]. They share the similar idea: the algorithm starts with a naive mechanism and improves the solution iteratively.

To summarize, compared to manual design, automated mechanism design approaches have realized reverse engineering of mechanisms since the mechanism search process is done by computers. However, it requires much expertise in both optimization and economics to formulate the original problem to the corresponding optimiza-
tion problem.

This thesis approaches the mechanism design problem from the perspective of computer-aided programming. Observed that algorithms and protocols are eventually encoded as real programs, we propose to directly represent mechanisms and games as programs. Specifically, a domain-specific language is provided for designers to encode the high-level specification of a mechanism design problem. Then, a program synthesis algorithm is applied to automatically generate mechanisms from specifications.

**Program Synthesis**

The goal of program synthesis is to automatically discover an executable piece of code provided with user defined specifications, which can be logic formulas, input-output examples, templates, etc. Program synthesis has been explored in many domains ranging from spreadsheet programming [18] to robotics [33]. Its applications have reached various classes of users such as non-expertise programmers, students and teachers, software developers, and algorithm designers. With the recent improvements in formal reasoning tools based on SAT/SMT solver and various advanced search engines, program synthesis has begun to render its promises in efficiently synthesizing small programs.

This thesis has drawn specific inspiration from the following works in designing MechSynth.

In [17], Armando introduces the Sketch language, which is a giant step in bridging the gap between general programming practice and inductive program synthesis systems. Sketch system provides an imperative programming language model, which allows programmers to write partial programs, i.e., programs with holes, to express their high-level implementation strategies. Instead of generating the entire program,
the synthesis problem in Sketch system reduces to the problem of searching for assignments to the holes. This search problem can be further framed as a constraint satisfaction problem. The beauty of this work lies in its “Synergy” between the programmer and the synthesizer. Specifically, the language developed in this thesis enables the synthesis process to be guided by the programmer’s expertise in specific domain. We applied this “Synergy” when designing the language used in MECHSYNTH. The domain specific language invented in this thesis allows economists to encode their high-level intuitions on the structure of mechanisms through partial programs.

Later, [34] formalizes the concept of synthesizing programs with constraint solvers and provides the implementation guideline. They introduced the three components for a system which can automatically convert a program searching problem to a constraint-based synthesis problem. These components include: 1) a language in which the programmer can encode the template of candidate programs P, termed partial program, as well as the desired properties φ, 2) an interpreter which translates the program to a first-order logic formula of the form $\exists p, \forall x, \phi(x, P(x))$, where $P$ is the program, and $x$ are the set of variables in the missing implementations, and 3) a backend program which implements the constrain solver based search algorithm. The program will search for the $x$ to complete the program $P$. We have followed this recipe in building MECHSYNTH. The detailed design and implementation are introduced in Chapter 3 and 4.

Compared with previous automated mechanism design methodologies, this computer-aided programming based approach has eased designers from manual formulation of mechanism design problems. Designers can easily explore different design spaces by making minor changes to already coded programs. Moreover, this program synthe-
sis based framework allows us to exploit symbolic reasoning, which can handle cases where prior knowledge on mechanism inputs is not available.
Chapter 3

MECHSYNTH Language

In this chapter, we formalize the computer-aided mechanism design problem using a core specification language. The Python-like syntax in Fig. 3.2 can be mapped to this language using standard techniques.

Particularly, there are two aspects that need special considerations when designing such specification language: 1) how to abstract a game and map its components into a program, 2) how to incorporate synthesis support into this language.

As mentioned in former chapters, this thesis mainly focuses on the set of preference aggregation problems where a collective decision has to be made based on agents’ reported preference [22]. The types of preference aggregation games studied here include auctions and voting. Observed that the identities of individual agents in this type of game should not affect final decision making, we represent individual agents by their types. Precisely, agents of same type can be captured by the same set of parameters or functions. Each agent is an instance of the agent type it belongs to. This abstraction resembles the Class construct in Object-Oriented language paradigm. Furthermore, the mechanism function is encoded as a simple procedural function. Finally, we use a “driver” which defines the execution of a program. The function of “driver” is similar to main() function in C programming language.

In order to address the second aspect, we adopt the Sketch programming language model [17], which provides a successful example in incorporating synthesis support into existing language. The implementing language of Sketch model is simply a pro-
cidental language extended with a basic integer hole. This new construct, the integer
hole, has given programmers a robust mechanism to express their insights on the
overall shape of the result code while leaving unspecified the low level implementa-
tion details. Applying the same idea, the MECHSYNTH language also includes the
construct of holes, represented by symbol $??$, for mechanism designers to encode
mechanism template, which results in a partial program.

With these two design decisions in mind, we now formally present the MECH-
SYNTH language.

\section{MECHSYNTH Language Syntax}

Let us assume disjoint universes of names for holes, fields in agents, functions, formal
parameters for functions, outputs of the market, and agents. Also, let $x$ denote a
nonempty tuple of names $\langle x_1, \ldots, x_n \rangle$, and for any nonterminal $N$, let $[N]^+$ represent
one or more occurrences of the nonterminal. Then, the grammar for a specification
in the core language can be defined as in Fig. 3.1.

A specification consists of a game (nonterminal Prog) and a requirement. A game
consists of:

- collection of agents

Each agent is of an agent type whose template is defined as nonterminal Agent
(e.g. Line 1-7 in figure 2.1). The template includes:

- a tuple $\bar{f}$ of fields

- a collection of utility functions $Utility$ A utility function $uf$ takes in a set
  of parameters $\bar{p}$, and computes a piecewise linear expression $UB$ over $\bar{p}$
  and the agent’s fields.
\[
\text{Prog} ::= [\text{agent } A : \text{Agent}]^+ : \text{Mech} : \text{output } \overline{o} = \langle D_1, \ldots, D_k \rangle
\]

\[
\text{Agent} ::= \text{field } \overline{f} : [\text{Utility}]^+
\]

\[
\text{Utility} ::= \text{uf}(\overline{p}) : UB
\]

\[
\text{Mech} ::= \text{m}(\overline{p}) : \text{hole } \overline{h} : \langle MB_1, \ldots, MB_k \rangle
\]

\[
D ::= n \mid D_1 + D_2 \mid n \ast A.f \mid \text{if } (D_1 \geq 0) \text{ then } D_2 \text{ else } D_3 \mid
\]

\[
(m(D_1, \ldots, D_k)(i)) \mid A.uf(D_1, \ldots, D_k)
\]

\[
UB ::= n \mid UB_1 + UB_2 \mid n \ast p \mid n \ast p \mid \text{if } (UB_1 \geq 0) \text{ then } UB_2 \text{ else } UB_3
\]

\[
MB ::= n \mid MB_1 + MB_2 \mid n \ast p \mid n \ast h \ast p \mid \text{if } (MB_1 \geq 0) \text{ then } MB_2 \text{ else } MB_3
\]

Figure 3.1: Core language for computer-aided mechanism design. Here, A is an agent name; \(\overline{f}\) and \(f\) represent field names; \(\overline{p}\) and \(p\) represent formal parameters; \(\overline{h}\) and \(h\) represent holes; \(\overline{o}\) represents outputs; and \(m\) and \(uf\) are function names.
• a mechanism template (nonterminal $Mech$)

A mechanism template $m$ takes in parameters $\overline{p}$ and computes an expression $MB$ over $\overline{p}$ and holes $\overline{h}$. This expression allows holes to be multiplied with parameters (as at the end, holes are unknown coefficients), but does not allow other nonlinear operations as discussed. We explain the reason for imposing this requirement in chapter 4.

• a driver ($\overline{\sigma} = \langle D_1, \ldots, D_k \rangle$)

A driver computes a tuple of outputs $\langle o_1, \ldots, o_k \rangle$, where $o_i$ evaluates to an expression $D_i$. The expressions $D_i$ can reference fields of arbitrary agents, invoke the mechanism, read the $i$-th component of the result, and also perform linear arithmetic.

Notice that the holes $\overline{h}$ and the agent fields $\overline{f}$, while declared, are not initialized to concrete values. The agent fields — types and actions of agents — can be viewed as inputs to the program. However, the partial program does not have deterministic execution because the fields are only symbols and, together with the holes, are not initialized.

An operational flavor of the system can be attained by instantiating the holes with numerical constants. In other words, the partial mechanism (the mechanism to synthesis) are turned into a complete executable function. Moreover, by assigning values to the fields of various agents, we can define a concrete market scenario, where the mechanism and the driver code produce a concrete output. Such an operational semantics is useful for simulating markets, and system MechSynth supports this facility.
3.2 Syntactic Sugar

The language defined so far only includes basic expressions and is relatively low-level. From researching the set of preference aggregation games, we observe that several more complex operations have been frequently used. Moreover, the encoding of these game-related operations can be tedious. To ease designer’s programming process, MechSynth offers a set of higher level constructs which are implemented as syntactic sugar over the set of basic expressions.

We summarized four frequently used operations in encoding game settings as follows:

1. **dot(D1, D2)**

   where D1 and D2 are any arithmetic expressions as defined in Fig. 3.1. In a preference aggregation setting, vector is used as a container of agents’ reported preferences. As we see in Fig. 3.2, arithmetic operations over these vectors are heavily used throughout the program. The dot( ) operation is therefore defined to represent the low level vector multiplications.

2. **swap(V, e1, e2)**

   where V is a vector, e1 is an current element in V, e2 is an element of same type as e1. swap() operations replace e1 in vector V with e2. This operation is frequently used in describing equilibrium properties for a game. For example, as described in chapter 2, a game is said to implement the concept of dominant strategy, if there exists only one best strategy for each agent such that: ∀s ∈ S_{−i},

   \[ u_i(M(s_1(\theta_1), \ldots, s_i^*(\theta_i), \ldots, s_n(\theta_n)), \theta_i) \geq u_i(M(s_1(\theta_1), \ldots, s_i(\theta_i), \ldots, s_n(\theta_n)), \theta_i). \]

   Let \( x_i \) stands for \( s_i(\theta_i) \), i.e, the action of agent \( i \). Notice that set \([x_i^*, X_{−i}]\) can
be constructed from \([x_i, X_i] \) using only one swap operation, namely \(\text{swap}(X, x_i, x_i^*)\). As demonstrated, \(\text{swap}()\) function makes it much easier to express equilibrium properties.

3. \(\text{sorted}(V)\)

where \(V\) is a vector. Often times, sortedness is a desire input property for mechanism function because when agents report their preference, one thing the decision maker usually does is to sort the preference, and then applies the mechanism. So, rather than encode the inequalities over each pair of elements in the input vector, we include the \(\text{sorted}()\) logic construct in the language.

4. \(V.\text{add}(e)\)

where \(V\) is a vector, and \(e\) is an element of the same type as the elements in the vector. This \(\text{add}\) operator resembles the preference collecting process in real life.

More high level constructs can be incorporate to the existing language as we continue modeling other type of games.

With this game modeling language in hand, the next challenge is how to exploit the intuition from the mechanism template, and synthesize the mechanism function that meets the specification.

### 3.3 Example Program

Now let us apply \textsc{MechSynth} language to write a declarative specification for the example single-goods auction, which is depicted in Fig. 3.2. This specification consists of the following components:
```python
def agent ag (id):
    field v, b

def uf(result):
    if (b == result.winningbid):
        then ut = v - result.winningprice
    else ut = 0
    return ut

def mechanism m(env):
    @pre: sorted(env)
    bidh = ??
    priceh = ??
    return {winningbid: dot(bidh, env), winningprice: dot(priceh, env)}

bids = emptyset
for i = 1 to N:
    agents[i] = new ag(i)
    bids.add(agent[i].b)
result = m(bids)
@post: forall i:
    let result’ = m(swap(bids, agents[i].b, agents[i].v)):
    agents[i].uf(result) <= agents[i].uf(result’)
```

Figure 3.2: Specification for single-goods auction
• The declaration of an “agent template” \( \text{ag} \) (Lines 1–7).

An agent of this template has two real-valued fields: a private valuation \( v \) of the object being auctioned, and the amount \( b \) that the agent actually bids. Agent template definitions also include some function definitions: the type \( \text{ag} \) has one such function \( \text{uf} \), which happens to be the common utility function for agents in auctions. The designer could define \( \text{uf} \) in other ways, or even define multiple functions within \( \text{ag} \) that capture different dimensions of agents’ utility functions. More generally, the system could allow for multiple agent templates.

• A “template” of a mechanism \( \text{m} \) (Lines 9–13).

The template specifies that: (1) the mechanism takes in a multiset \( \text{env} \) of bids that is sorted in descending order (the sortedness of the multiset is specified as a precondition); (2) the mechanism returns a record that contains the bid of the agent that won the auction (the field \( \text{winningbid} \)) and the price the winner has to pay (\( \text{winningprice} \)). Naturally, the winning bid and price depend on the content of \( \text{env} \). The programmer, however, does not know these expressions precisely; her only guess is that they are linear. we model this partial knowledge by declaring two coefficient vectors \( \text{bidh} \) and \( \text{priceh} \), and letting the winning bid and price respectively be the dot products of these vectors with \( \text{env} \). we set these coefficient vectors to unknown constants (known as \text{holes}) represented by the symbol \( ? ? \) (the holes) in the code.

• Some “driver” code (Lines 15–19) to “execute” the auction.

In the present example, this code constructs a fixed number \( N \) agents using agent template \( \text{ag} \) along with a multiset that collects their bids. Then it invokes the mechanism \( \text{m} \) and records the result. Note that this code does not
instantiate the bids or valuations of agents, but leaves them as symbolic entities.

• A statement that truth-telling is the dominant strategy, in the form of a post-
  condition written in first-order logic (Lines 20–22).

In this syntax, \( \text{swap}(S, b, v) \) refers to the multiset \( S \setminus \{b\} \cup \{v\} \). In general, the program can have multiple assertions at different points of the code.

The partial mechanism \( m \) corresponds to a set of (concrete) mechanisms obtained by instantiating the holes with arithmetic expressions. Now, the goal of synthesis here is to automatically find a correct set of assignments for the holes — i.e., satisfies all assertions in the code for all values of the fields \( v \) and \( b \) in the various agents.
Chapter 4

Synthesis Algorithm of MechSynth

The computer-aided mechanism design approach studied in this thesis resolves the problem of automated mechanism generation by formulating it as program synthesis problem. The program synthesis problem is further converted to a constraint satisfaction problem solved by Satisfiability Modulo Theories (SMT) solver. The synthesis algorithm, therefore, includes two parts: 1) the semantics for translating high-level specification of a problem to a first-order logic formula, and 2) a decision procedure for solving the generated formulas.

The first half of this chapter gives a precise definition of the semantics of the MechSynth language that performs the translation of a game into a collection of arithmetic expressions. For example, the game’s outputs can be represented in terms of its holes and inputs. Furthermore, the problem of reasoning about a game can be reduced to a problem of first-order theorem-proving using the semantics. Specifically, the MechSynth system reduces mechanism synthesis to checking the validity of a quantified formula in the first-order theory of bilinear real arithmetic. The second half of this chapter explains the decision procedure developed as part of this thesis, which is tailored for this family of formulas.
4.1 Synthesis Semantics

Formally, let $S$ represent a game, $A_1, \ldots, A_n$ be the agents; $m$ be a mechanism template. Let us assume, without loss of generality, that each agent has $l$ number of fields as $f_1, \ldots, f_1$. Also, let the holes in $m$ be $h_1, \ldots, h_k$. Let a fresh logical variable $h_i$ be introduced for each hole $h_i$, together with a fresh logical “input” variable $f_{i,j}$ for each expression $A_i.f_j$, and consider the set $\Phi$ of real arithmetic expressions over these variables. Now we show how to give the semantics of $S$ as a tuple of such expressions.

Let an environment $\sigma$ be a partial function that maps variable names to expressions in $\Phi$. Intuitively, an environment assigns a meaning to variables that are free within a given scope. We start by inductively defining a semantics $\llbracket D_i \rrbracket_\sigma$ of the driver expressions $D_i$ of $S$ under the environment $\sigma$ (and similarly, the bodies of mechanisms and utility functions).

- The semantics of arithmetic operations are the usual ones — for example, $\llbracket D_1 + D_2 \rrbracket_\sigma = \llbracket D_1 \rrbracket_\sigma + \llbracket D_2 \rrbracket_\sigma$. The remaining rules in this category are omitted here.

- Field and variable lookups:
  $$\llbracket A_i.f_j \rrbracket_\sigma = f_{i,j} \quad \llbracket x \rrbracket_\sigma = \sigma(x).$$

- Invoking utility functions:
  $$\llbracket A_i.\text{uf}(D_1, \ldots, D_k) \rrbracket_\sigma = \llbracket E \rrbracket_\sigma[\text{uf} \mapsto (f_{i,1}, \ldots, f_{i,k})], \ p \mapsto ([D_1]_\sigma, \ldots, [D_k]_\sigma),$$
  where $\text{uf}$ has body $E$ and formal parameters $p$.

- Invoking the mechanism and reading the $i$-th component of the tuple that it returns:
  $$\llbracket (m(D_1, \ldots, D_k))(i) \rrbracket_\sigma = \llbracket E_i \rrbracket_\sigma[\text{m} \mapsto (h_1, \ldots, h_k)], \ p \mapsto ([D_1]_\sigma, \ldots, [D_k]_\sigma)]$$
where \( \mathbf{m} \) has a body \( E_1, \ldots, E_k \)

Finally, let the outputs of one round of game \( S \) be \( o_1, \ldots, o_k \), bound respectively to expressions \( D_1, \ldots, D_k \). Then we define the semantics of \( S \) to be \( \llbracket S \rrbracket = \langle \llbracket D_1 \rrbracket \epsilon, \ldots, \llbracket D_k \rrbracket \epsilon \rangle \), where \( \epsilon \) is the empty map. The \( i \)-th component of this tuple is denoted by \( \llbracket S \rrbracket (i) \).

A requirement \( \mathcal{R} \) for the above game \( S \) is a pair \((\varphi_1, \varphi_2)\), where \( \varphi_1 \) is a pre-condition over the input variables \( f_{i,j} \) and the holes \( h_i \), and \( \varphi_2 \) is a post-condition over a set of output variables \( o_i \) as well as \( h_i \) and \( f_{i,j} \). Both \( \varphi_1 \) and \( \varphi_2 \) are formulas in quantifier-free real arithmetic that permit multiplication of variables \( h_i \) and \( f_{i,j} \), but no further nonlinear operation. This linearity is guaranteed by the set of mechanism functions supported by current MECHSYNTH language. We discuss the reason for this requirement in Chapter 4.2.1.

Now, mechanism synthesis amounts to finding an instantiation of the holes \( \overline{h} \) of \( S \) such that the resulting games satisfy the requirement \( \mathcal{R} \). Let \( \overline{h} = \langle h_1, \ldots, h_k \rangle \) and \( \overline{f} = \langle f_{1,1}, \ldots, f_{n,k} \rangle \). It is easy to see that mechanism synthesis amounts to:

**Problem 1 (Mechanism synthesis)** Find an instantiation of the variables \( \overline{h} \) such that the following formula (the synthesis condition) is valid:

\[
\Psi_{S, \mathcal{R}}(\overline{h}, \overline{f}) \equiv \varphi_1 \Rightarrow \left( \bigwedge_i (o_i = \llbracket S \rrbracket (i) \Rightarrow \varphi_2) \right).
\]

**Example**

Now, let us consider the specification for Vickrey auction in Fig. 3.2) with three agents. The variables are defined as follows:

- \( b_0, b_1 \) and \( b_2 \) represent the bids for three agents, respectively;
- \( v_0, v_1, \) and \( v_2 \) represent the agents’ valuation of the goods;
\( c_0, c_1, c_2 \) are the missing coefficients (holes) in the mechanism template.

The agent’s utility function is \( uf(\ldots) \) for \( A.uf \). Here, we assume the agents have common utility function. Also, for simpler notation, rather than expanding the body of \( uf \) and \( m \) each time we use them, we leave them as interpreted functions.

Finally, let the precondition of the mechanism \( m \) be:

\[
F \equiv b_0 \geq b_1 \geq b_2.
\]

Let the postcondition of the game (after unrolling the \texttt{for}-loop) be:

\[
uf(m(result)) \leq uf(m(v_0, b_1, b_2)) \quad \land \quad uf(m(result)) \leq uf(m(b_0, v_1, b_2)) \quad \land
uf(m(result)) \leq uf(m(b_0, b_1, v_2)).
\]

All the variables other than the number of agents are logical variables in the domain of real numbers.

By applying the semantics, the \texttt{MechSynth} system computes a synthesis condition \( \Psi \) that is the conjunction of three formulas \( \psi_0 \), \( \psi_1 \) and \( \psi_2 \), each generated from the requirements on one agent. Each \( \psi_i \) is a conjunction of formulas \( \psi_{i,j} \) that corresponds to an ordering relationship between the \( v_i \)'s and the \( b_i \)'s. In more detail:

(1) \( \psi_0 \equiv \psi_{0,0} \land \psi_{0,1} \land \psi_{0,2} \), where:

- \textit{Case} \( v_0 \geq b_1 \geq b_2 \):

\[
\psi_{0,0} \equiv (b_1 \leq v_0) \land (b_2 \leq b_1) \Rightarrow (v_0 - c_0 \cdot b_0 - c_1 \cdot b_1 - c_2 \cdot b_2 \leq v_0 - c_0 \cdot v_0 - c_1 \cdot b_1 - c_2 \cdot b_2)
\]

- \textit{Case} \( b_1 \geq v_0 \geq b_2 \):

\[
\psi_{0,1} \equiv (v_0 \leq b_1) \land (b_2 \leq b_1) \Rightarrow (v_0 - c_0 \cdot b_0 - c_1 \cdot b_1 - c_2 \cdot b_2 \leq 0)
\]
By applying the semantics, the original mechanism design problem is now automatically translated to the synthesis problem $\Psi$:

$$\Psi(b_0, b_1, b_2, v_0, v_1, v_2) \equiv (b_0 \geq b_1 \geq b_2) \Rightarrow (\psi_0 \land \psi_1 \land \psi_2).$$

The goal is to find values for $c_0, c_1, c_2$ such that $\Psi$ is valid.
4.2 Decision Procedure

4.2.1 Assumptions

Theoretically, by assuming the specifications only manipulate real-valued data and have no unbounded loops, the problem of synthesizing values for unknown (constant) holes can be posed as a decision question in the first-order theory of reals [35], and is thus decidable. Therefore, Problem 1 is also decidable. However, because the problem is obviously NP-hard, good heuristics are essential for solving the problem effectively on practical instances.

This thesis focuses on a restricted class of mechanism functions, which allows for designing more efficient synthesis algorithms. Specifically, the synthesis algorithm requires:

(1) the expressions computed by a mechanism to be piecewise linear functions (with unknown coefficients) of the mechanism’s inputs,

(2) no quantification over unbounded sets in the pre- and post-conditions.

Though this decision choice has imposing strong restrictions on the classes of mechanisms, we found there are many practical mechanisms design settings satisfying these requirements.

Given the assumed linearity of mechanism function, MechSynth uses a quantifier elimination technique to reduce mechanism synthesis to checking the satisfiability of a quantifier-free formula in nonlinear real arithmetic. This problem is then solved using a SMT solver [19].
4.2.2 Early termination

In Problem 1, the goal is to instantiate variables $\overline{h}$ so that the synthesis condition $\Psi$ is valid. The problem is equivalent to finding $\overline{h}$ such that $\neg \Psi$ is unsatisfiable.

we solve this problem by converting $\neg \Psi$ into the conjunctive normal form (CNF). Let $C_{Clauses}$ denote the set of clauses in this CNF formula. we observe that $\neg \Psi$ is unsatisfiable if any subset of $C_{Clauses}$ is unsatisfiable. In other words, the synthesis algorithm can terminate early if it finds assignments to $\overline{h}$ that makes a subset of $C_{Clauses}$ unsatisfiable.

Using this observation, MECHSYNTH generates subsets $C$ of $C_{Clauses}$ in the order of increasing cardinality $i$. For each $C$, we create a separate problem instance that searches for $\overline{h}$ such that $C(\overline{h}, \overline{f})$ is unsatisfiable. MECHSYNTH terminates if any of these searches is successful. (In all the case studies, MECHSYNTH is able to terminate for a small value of $i$.)

4.2.3 Quantifier Elimination

Now we show how to find $\overline{h}$ such that $C(\overline{h}, \overline{f})$ is unsatisfiable. MECHSYNTH reduces this problem to the problem of satisfiability checking in quantifier-free nonlinear arithmetic, using a technique originally introduced in the context of program verification by [36].

Because the mechanism functions are restricted to the class of piecewise linear function of the mechanism’s inputs, each of them is, precisely, a bilinear function of the unknown coefficients and the mechanism’s inputs. Given the fact that the formula $C$ will only involve bilinear arithmetic, we can use Motzkin’s transposition theorem [37]: 
Theorem 1 ([37])

For matrices $A$, $B$ and vectors $b$, $c$, the following statements are equivalent:

- The system $Ax \leq b, Bx < c$ has a solution.

- For all vectors $y \geq 0, z \geq 0$, we have:

$$A^T y + B^T z = 0 \Rightarrow b^T y + c^T z \geq 0 \text{ and } A^T y + B^T z = 0 \land z \neq 0 \Rightarrow b^T y + c^T z > 0.$$ 

By negating the two statements in Motzkin’s theorem, we conclude that the following two statements are equivalent:

- The system $Ax \leq b, Bx < c$ has no solution

- There are vectors $y, z$, such that

$$y \geq 0 \land z \geq 0 \land A^T y + B^T z = 0 \land b^T y + c^T z < 0 \text{ or } y \geq 0 \land z > 0 \land A^T y + B^T z = 0 \land b^T y + c^T z \leq 0.$$ 

Now suppose we are asked to prove the unsatisfiability of a conjunctive system of linear constraints. By the above observation, this problem (“Show that a linear system has no solution”) can be reduced to checking the satisfiability of a dual system of constraints (“Find $y, z$ such that a property holds”). Note that the dual system does not have any reference to $x$, the “inputs” for the original system. In a mechanism design context, this reduction gives a prior-free mechanism.

Consider again the problem of finding $\overline{h}$ such that $C(\overline{h}, \overline{f})$ is unsatisfiable. To apply Motzkin’s theorem, we first convert $C$ into the disjunctive normal form (DNF). Let $D\text{Clauses}$ denote the set of clauses in the DNF of $C$: each clause $D_i(\overline{h}, \overline{f}) \in$
**DClauses** is a linear system where coefficients are either constants, or products of constants and hole variables. The formula $C$ is unsatisfiable iff each $D_i$ is unsatisfiable.

For each $D_i$, we generate the dual formula $D'_i(\overline{h}, \overline{y}, \overline{z})$. Note that $D'_i$ is a quantifier-free formula, and also that it does not refer to the input variables $\overline{f}$ (corresponding to the vector $x$ in Theorem 1) that were universally quantified in $C$. The synthesis problem now amounts to finding a satisfying assignment of

$$\Theta_C = \bigwedge_i D'_i(\overline{h}, \overline{y}, \overline{z})$$

### 4.2.4 Example

Consider the following constraint generated by MechSynth from the Vickrey Auction with three players. Here, $c_0, c_1, c_2$ are the holes, and $b_0, b_1, b_2, v_0, v_1, v_2$ are the inputs representing bids and private valuations of the three players:

$$(b_2 \leq v_0) \land (v_0 \leq b_1) \land (v_0 - c_0 \cdot b_0 - c_1 \cdot b_1 - c_2 \cdot b_2 \leq 0).$$

we rewrite it as a linear system $Mx \leq n$, where

$$M = \begin{bmatrix} 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ -c_0 & -c_1 & -c_2 & 1 & 0 & 0 \end{bmatrix} \quad x = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ v_0 \\ v_1 \\ v_2 \end{bmatrix} \quad n = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

We can instantiate $c_0, c_1, c_2$ to render this system unsatisfiable iff the constraint
(y \geq 0) \land (M^T y = 0) \land (n^T y < 0) \text{ is satisfiable, where:}

\[
M^T = \begin{bmatrix}
0 & 0 & -c_0 \\
0 & -1 & -c_1 \\
1 & 0 & -c_2 \\
-1 & 1 & 1 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
\]

n^T = \begin{bmatrix}
0 \\
0 \\
0
\end{bmatrix}

However, it is easy to see that the dual constraint is not satisfiable.

\subsection{SMT-solving}

We check the satisfiability of each $\Theta_C$ using an SMT-solver, a generic engine for quantifier-free first-order theories. SMT-solvers combine the capabilities of the DPLL procedure for boolean satisfiability and specialized decision procedures for quantifier-free first-order theories [38]. The last decade has seen a revolution in SMT-solving and numerous applications of such solvers in software verification and synthesis [39]. Specifically, we use the Z3 SMT-solver [19], which implements a highly optimized satisfiability procedure for nonlinear real arithmetic.

In case $\Theta_C$ has a satisfying assignment, MECHSYNTH will return the assignment to $\overline{h}$ as a solution to the synthesis problem. If $\Theta_C$ is unsatisfiable, then as explained in Section 4.2.3, the algorithm moves on different subset of clauses in the CNF of $\Psi$. The algorithm terminates reporting failure if $\Theta_C$ is unsatisfiable for all subsets $C$ of clauses in the CNF of $\Psi$. 
\textbf{input}: first-order formula $\Psi_{S,R}(\overline{h}, \overline{f})$

\textbf{output}: satisfying assignments of $\overline{h}$ or UNSAT

1 Negate $\Psi$;

2 Convert $\neg \Psi$ to conjunctive normal form $CNF(\neg \Psi)$;

3 Let $C\text{Clauses}$ denotes the set of conjunctive clauses of $CNF(\neg \Psi)$;

4 Generate all subsets of $C\text{Clauses}$ in order of increasing cardinality;

5 $status \leftarrow UNSAT$;

6 foreach subset $C$ do

7 Let $c$ be the conjunction of the clauses in $C$;

8 Convert $c$ in disjunctive normal form $DNF(c)$;

9 Let $D\text{Clauses}$ be the set of clauses in $DNF(c)$;

10 $\Theta_c \leftarrow TRUE$;

11 for $i = 0$ to $|D\text{Clauses}|$ do

12 Introduce new variables $\overline{y}_i, \overline{z}_i$;

13 Apply Motzkin’s theorem to transform $D_i(\overline{h}, \overline{f})$ to $D'_i(\overline{h}, \overline{y}, \overline{z})$;

14 $\Theta_c = \Theta_c \land D'_i(\overline{h}, \overline{y}, \overline{z})$

15 end

16 $status = SMT(\Theta_c)$;

17 if $status == SAT$ then

18 return assignments of $\overline{h}$

19 end

20 end

21 return UNSAT

\begin{algorithm}
\caption{the algorithm of SYNTH}
\end{algorithm}
4.2.6 Correctness

Let $C$ denote a subset of $CClauses$, and the algorithm consisting of the above phases (as summarized in Algorithm 1) be called SYNTH. It is easy to see that:

Theorem 2

Every solution returned by SYNTH is a correct solution to Problem 1. Moreover, if Problem 1 has a solution, then SYNTH returns a solution; otherwise it reports failure.

4.2.7 Complexity

Theorem 3

The SMT-formula generated from Problem 1 is of size $n \times 2^{O(n)}$ where $n$ is the number of agents.

Proof The formula in Problem 1, $\Psi_{S,R}(\vec{h}, \vec{f})$, is a first-order formula involving only bilinear arithmetic. Let the inequalities be the literals. The Well-formed formula of $\Psi_{S,R}(\vec{h}, \vec{f})$ has $O(n)$ number of clauses.

Converting $\neg \Psi$ in conjunctive normal form produces formula $CNF(\neg \Psi)$, which has worst-case length of $2^{O(n)}$ conjunctive clauses (line 2 in Algorithm 1). Then the number of iterations of the outer for-loop (line 6-20) is $2^{2^{O(n)}}$ in worst case, $|CClauses| = 2^{2^{O(n)}}$.

For each subproblem in $CClauses$, the conjunction of all clauses in the subproblem, $c$, has worst-case length of $2^{O(n)}$ (when all conjunctive clauses are included). Let $k$ be the average length of each conjunctive clause of the subproblem. $DNF(c)$ contains $k^{2^{O(n)}}$ disjunctive clauses. $|DClauses| = k^{2^{O(n)}}$.

Each disjunctive clause $D_i(\vec{h}, \vec{f})$ in $DClauses$ of $DNF(c)$ is a linear system with $\vec{h}$ being the coefficients and $\vec{f}$ being the variables. Applying Motzkin’s theorem to $D_i(\vec{h}, \vec{f})$ results in a new linear system $D'_i(\vec{h}, \vec{y}, \vec{z})$. The total size of $\vec{h}$, $\vec{y}$ and $\vec{z}$ is...
linear to the total size of $\overline{h}$ and $\overline{f}$. $|D'_i(\overline{h}, \overline{y}, \overline{z})| = O(n)$.

Therefore, the formula for each subproblem in $CClauses$, which is to be solved by SMT solver, is of size $|DClauses| * O(n) = k^{2^{O(n)}} * n$.

Then, the worst-case complexity (when the satisfiability of all subproblems in $CClauses$ are checked) of Problem 1 is $2^{2^{O(n)}} * k^{2^{O(n)}} * n = n * 2^{2^{O(n)}}$. 
Chapter 5

Case Studies

In this chapter, we report on several case studies where MechSynth is applied to a set of classical preference aggregation games as well as variations on them. These mechanisms fall into two categories: auctions and mechanisms for aggregating social choice.

Given the language implemented so far and the requirements posed by the synthesis algorithm, we mainly focus on synthesizing mechanisms satisfying the following criterions:

- the input variables have no probabilistic information. In other words, my system excludes the set of probabilistic properties such as Bayesian Nash equilibrium and optimal expected revenue.

- the mechanism function satisfies the requirements discussed in Chapter 4.

- the mechanism function implements dominant strategy.

- regarding games involving resource redistribution such as auction, payment is used as part of the mechanism.

Furthermore, for each auction, we first synthesize a truth-telling mechanism. If the auction has a truth-telling mechanism, we continue to explore other possible mechanisms by changing the properties of the outcome at Nash equilibrium. This
is based on the revelation principle, which states that if a game of incomplete information has a Bayesian Nash equilibrium, there will always exist a payoff-equivalent revelation mechanism that has an equilibrium where the players truthfully report their types [40]. Namely, if there exists other mechanisms result in a different equilibrium profile, there must exist a mechanism implements truth-telling property; if there does not exist a truth-telling mechanism, there will not exist any mechanism which achieves equilibrium profile.

In particular, we studied three types of auctions and their variations, including a single goods auction, a multi-unit identical goods auction, and a one-shot position auction. In all cases, we assumed that 1) the agents have their own private valuation of the auctioned goods, that 2) we do not have prior knowledge about the bids and valuations, and that 3) bids and valuations are real-valued.

Among social-choice mechanisms, we studied a general voting game, and reconstructed an interesting “fair” voting mechanism that results in dictatorship.

### 5.1 Single-goods auction

We synthesized the truth-revealing mechanism for the single-goods auction described in Chapter 1.1. The mechanism template and requirements for this game are as described before; in particular, the bids received by the mechanism are sorted in descending order (and numbered from 0 through \((N - 1)\), and \(bid_h\) and \(price_h\) are respectively the unknown coefficients in the expressions for the winning bid and price.

For a game of \(N\) agents, the mechanism function synthesized by MechSynth on this example has the following form:

\[
bid_h[i](B) = \begin{cases} 
1 & \text{for } i = 0 \\
0 & \text{for } i \neq 0
\end{cases}
\]

and

\[
price_h[i](B) = \begin{cases} 
1 & \text{for } i = 1 \\
0 & \text{for } i \neq 1
\end{cases}
\]
Given the sortedness of the bids, this means that the highest bidder wins the auction, and is asked to pay the second highest bid. MECHSYNTH can synthesize mechanisms for single-goods auctions with up to five players, and produce a result in less than 30 seconds.

5.1.1 Variant 1

We also explored several variations on the basic single-goods auction in Fig. 3.2. The first variation was the “almost truthful” auction, where we allow dominant strategies in which the $i$-th agent can bid within a factor $(1 + \alpha_i)$ off the true value.

Whenever the constants $\alpha_i$ were different across agents, MECHSYNTH terminated reporting that there was no mechanism that met the requirements. These empirical observations turned out to be special cases of a general property: there does not exist a linear “almost truthful” mechanism where bidding $\alpha_i$ off the true value is the dominant strategy, unless $\alpha_i$ is the same for all agents.

5.1.2 Variant 2

Next we considered a modification of the above “almost-truthfulness” property where $\alpha_i$ is the same constant $\alpha$ for all agents. In this case, MECHSYNTH finds a satisfactory mechanism of the following form:

$$bidh[i](B) = \begin{cases} 
1 & \text{for } i = 0 \\
0 & \text{for } i \neq 0 
\end{cases} \quad \text{and} \quad price[i](B) = \begin{cases} 
\frac{1}{1+\alpha} & \text{for } i = 1 \\
0 & \text{for } i \neq 1 
\end{cases}$$

Since the dominant strategy for agent $i$ is $b_i = (1 + \alpha) \cdot v_i$, revenue generated by this mechanism is $\frac{1}{1+\alpha} \cdot (1 + \alpha) \cdot v_1 = v_1$. Therefore, the designer cannot increase revenue simply by providing agents incentives to overbid.
5.1.3 Variant 3

Next we considered a variation on the game where the utility function is:

\[
uf(result) = \begin{cases} 
  v - result.winningprice + \alpha & \text{for } b = result.winningbid \\
  0 & \text{for } b \neq result.winningbid
\end{cases}
\]

MECHSYNTH terminates reporting failure in this case, meaning that there is no linear truthful mechanism under this assumption.

5.1.4 Variant 4

Finally, we considered an alternative agent model where utility function is:

\[
uf(r) = \begin{cases} 
  v - r.winningprice & \text{for } b = r.winningbid \land r.winningprice \leq (1 - \alpha) \cdot v \\
  \alpha \cdot v & \text{for } b = r.winningbid \land r.winningprice > \alpha \cdot v \\
  0 & \text{for } b \neq r.winningbid
\end{cases}
\]

where \(r\) is short for \(result\).

Once again, the system established that this agent model does not permit a linear truthful mechanism.

5.2 Multi-stage single goods auction

Next we considered a multi-stage single-goods auction [41] where each stage applies the same rule, and there are \(R\) stages. Fig. 5.1 shows part of the specification of this auction that is different from the one for single-goods auctions (Fig. 3.2). Here, line 2 declares an array of bids of size \(R\). Line 12 - 15 constructs the array of inputs \(bids\) for \(R\) rounds of auction. This process is run \(R\) times (Line 17-18).

Other than the property of truth-telling, we can relax the property to “overall” truth-telling, which means the total value of bids from round 1 to \(R\), \(b_1 + b_2 + ... + b_R\), is equal to \(R \cdot v\). This property can be coded as in Fig. 5.2.
def agent ag (id):
    field v, b[R]
def uf(result):
    ...
    return ut...

for j = 1 to R:
    bids[j] = emptyset

for i = 1 to N:
    agents[i] = new ag(i)
    for j = 1 to R
        bids[j].add(agent[i].b[j])

for j = 1 to R:
    result[j] = m(bids[j])

Figure 5.1: Specification for multi-stage single-goods auction

@post: forall i:
    for all j:
        let result' [j] = m(swap(bids[j], agents[i].b[j], agents[i].v):
        agents[i].uf(result[1]) + ... + agents[i].uf(result[R])
    <=
        agents[i].uf(result'[1]) + ... + agents[i].uf(result'[R])

Figure 5.2: Overall truth-telling
forall i:
for all j:
    let result'[j] = m(swap(bids[j], agents[i].b[j], agents[i].v):
        (1+n*0.05)* agents[i].uf(result[1]) + ... + 1*agents[i].uf(result[R])
    <=
        (1+n*0.05)*agents[i].uf(result'[1]) + ... + 1*agents[i].uf(result'[R])

Figure 5.3 : Discounted utility

we can also model cases where the agent has discounted utility. The property in this case can be coded as in Fig. 5.3.

In each of these cases, the system generates the second price mechanism as a solution.

5.3 Multi-units identical goods auction

We extended our modeling of the single-goods auction to an auction of multiple units of homogenous goods. In my model of this auction:

1. There are n agents and k units of identical goods.

2. Agent models are the same as in single-goods auction.

3. A mechanism template for this setting consists of templates for an allocation function and a payments function.

   • Allocation function: Let us assume that the bids are sorted in decreasing
order. The matrix of unknown coefficients for the allocation function is

\[
bidh = \begin{bmatrix}
c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\
c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\
\vdots & \vdots & \ddots & \vdots \\
c_{n,1} & c_{n,2} & \cdots & c_{n,n}
\end{bmatrix}
\]

For each row of \( bidh \), the \( i \)-th agent wins a unit of goods only when \( bidh[i] \cdot env > 0 \), where \( env \) is a vector of bids sorted in descending order.

- The matrix of unknown coefficients for the price function is:

\[
priceh = \begin{bmatrix}
d_{1,1} & d_{1,2} & \cdots & d_{1,n} \\
d_{2,1} & d_{2,2} & \cdots & d_{2,n} \\
\vdots & \vdots & \ddots & \vdots \\
d_{n,1} & d_{n,2} & \cdots & d_{n,n}
\end{bmatrix}
\]

For each row of \( priceh \), the price that the \( i \)-th agent needs to pay can be calculated as \( priceh[i] \cdot env \).

4. The requirement asserts that truth-telling is the dominant strategy for all agents.

\textsc{MechSynth} computed the following mechanism from this specification:

\[
bidh[i][j](B) = \begin{cases} 
1 & \text{for } i \leq k \land j = i \\
0 & \text{for } i > k \lor j \neq i
\end{cases}
\]

and

\[
priceh[i][j](B) = \begin{cases} 
1 & \text{for } i \leq k \land j = k + 1 \\
0 & \text{otherwise}
\end{cases}
\]

The result is special case of a more general theorem proved by [42].
5.4 Position auction

We used MECHSYNTH to automatically synthesize a position auction [43] of keywords in a web search. We modeled a one-shot position auction for an arbitrary keyword as follows:

1. There are $k$ positions related to the keyword. The click-through rate of the $i$-th position is $ctr_i$ and $ctr_1 > ... > ctr_k$. There are $n$ players, and we have $k < n$.

2. Each agent has fields for its bid $b$ and its true valuation $v$. Utility functions of agents are as follows:

$$uf(result) = \begin{cases} 
ctr_i \cdot (v - result.winningprice[i]) & \text{for } b = result.winningbid[i] \\
0 & \text{for } b \neq result.winningbid[i]
\end{cases}$$

3. The allocation and payment functions are the same as those in the previously described multi-units identical goods auction.

4. The requirement states that truth-telling is the dominant strategy.

In my experiments, we synthesized a mechanism for position auction of four players and two slots with $ctr_1 = 5$ and $ctr_2 = 2$. The values of $bidh$ and $priceh$ in the resulting mechanism were as follows:

$$bidh = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad priceh = \begin{bmatrix} 0 & \frac{3}{5} & \frac{2}{5} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

This result is confirmed by [44], who establish that assuming the $k$-th player pays the $(k+1)$-th player’s bid, the relationship between payments of $i$th player and
(i + 1)-th player is given by:

$$\text{payment}_i - \text{payment}_{i+1} = \frac{(\text{ctr}_i - \text{ctr}_{i+1})}{\text{ctr}_i} \cdot (\text{bid}_i - \text{bid}_{i+1}).$$

### 5.5 Aggregating social choice

The objective of social-choice mechanisms is to pool the preferences of different agents so as to best reflect the wishes of the population as a whole. Voting is perhaps the most well-known example of such a mechanism. Most such mechanisms have the following components: (a) n voters; (b) a set A of m candidates to vote for; (c) a preference profile that captures the voting preferences of the each voter. A preference profile $L_i$ for the $i$-th voter can be defined as a real-valued utility function $u_i : A \to \mathbb{R}$, where a candidate gets higher utility when $i$-th voter thinks that she is more preferred to win in the election.

In addition, we invented a linear social welfare function that aggregates individual utility into an aggregate utility for the society. The coefficient of each input variable, namely, individual’s preference, quantifies how much effect is the person’s preference on the final decision. Let $up_i$ denotes the set of utilities over $A$ evaluated by $u_i$, which is the utility function of $i$-th voter. The social welfare function can be represented as $f : UP \to up_s$, where $UP = \bigcup up_i$, and $up_s$ means the set of aggregated social utilities over candidates $A$.

Recall the three basic criterion for a voting mechanism listed in [25]:

1. **Pareto efficiency (Unanimity):** If for a pair of alternatives $(x, y)$, $\forall i, u_i(x) > u_i(y)$, then $u_s(x) > u_s(y)$.

2. **Independence of irrelevant alternatives:** If $U$ and $\tilde{U}$ have same preference profiles over an arbitrary pair of alternatives $(x, y)$, then $u_s$ and $\tilde{u}_s$ has same preference
order of \((x, y)\).

3. **Non-dictatorship:** There is no individual \(i\) whose preference always prevails. In other words,

\[
\exists i \forall (x, y), x \in A, y \in A, x \neq y : u_x(x) > u_y(y) \iff u_i(x) > u_i(y).
\]

Arrow’s impossibility Theorem says if a social welfare function satisfies two of the above three properties, then the third cannot hold [25].

We used MECHSYNTH in the synthesis of a voting mechanism with 4 voters and 5 alternatives. We modeled social welfare functions as linear functions over the preferences of voters with unknown coefficients \(c_1, \ldots, c_k\). We synthesized a social choice function where the coefficients for the preferences of the four voters were given by the vector

\[
\begin{bmatrix}
c_0 & c_1 & c_2 & c_3
\end{bmatrix} = \begin{bmatrix}
0 & 0 & \frac{1}{8} & 0
\end{bmatrix}.
\]

After taking the products of this coefficient vector \(c\) with \(UP\) (the union of the utility profiles), we can see that the set of social utilities over \(A\) depends only on the utility profile of the third agent. Thus, this mechanism satisfies the first two properties in the above list, but is a dictatorship where the third voter determines the final outcome.
Chapter 6

Concluding Remarks and Future Work

In this thesis, we present a new approach to automating mechanism design. Our system is the first attempt at designing mechanisms with program synthesis techniques. Our approach allows the programmer to specify partial information of a game through a specification language. The problem of mechanism design is then translated to the problem of satisfiability checking in quantifier-free bilinear real arithmetic. This translation is implemented through the MechanSynth system and is built on top of the Z3 SMT solver. We evaluate the system by synthesizing several classic mechanisms and their variations.

Currently, MechanSynth supports the design of prior-free mechanisms. However, priors of mechanism inputs are often available especially in online markets (i.e. using the known bid distribution of one product on similar or related products). Bayesian mechanism design [45] assumes prior knowledge of the mechanism inputs and aims to find the mechanisms that maximize the expected objective value, and much of the mechanism design literature is focused on this form of optimization. With recent advances in probabilistic programming, we look to extend the specification language to allow for the encoding of distributional information on mechanism inputs and incorporate such information in the design process.

We also plan to apply other techniques from formal methods to investigate mechanism behaviors. One promising direction is to synthesize mechanisms that satisfy certain property for an arbitrarily large number of players. Another is to permit
unbounded loops to capture repeated interactions between agents and the mechanism. Such extensions may render the synthesis problem undecidable. In that case, the challenges will be identifying interesting decidable subsets of the problem and designing heuristics that behave well on a large number of practical examples.
Bibliography


