Automating Mechanism Design with Program Synthesis

Sai Kiran Narayanaswami
UT Austin
Austin, Texas, USA
nskiran@cs.utexas.edu

Swarat Chaudhuri
UT Austin
Austin, Texas, USA
swarat@cs.utexas.edu

Moshe Vardi
Rice University
Houston, Texas, USA
vardi@cs.rice.edu

Peter Stone
UT Austin and Sony AI
Austin, Texas, USA
pstone@cs.utexas.edu

ABSTRACT

This paper presents a new approach to the automated design of *mechanisms* that incentivize self-interested agents to maximize a global objective (such as revenue or social welfare) in equilibrium. Prior work on automated design has either been restricted to relatively simple mechanisms, or represented mechanisms as neural networks that are hard to interpret and cannot easily incorporate prior knowledge. In this paper, we propose program synthesis as a way around these issues. Concretely, we formalize the problem of designing mechanisms in the form of multiagent environments whose transition and reward functions are programs in a domain-specific language (DSL), in order to maximize an outcome such as revenue or social welfare under given assumptions on how agents act in these environments. We present an initial algorithm, based on a combination of stochastic search over programs and Bayesian optimization, for this problem. We empirically evaluate the algorithm in two domains with different characteristics. Our experiments suggest that the approach can synthesize programmatic mechanisms that are human-interpretable and also perform well.

KEYWORDS

Mechanism Design, Program Synthesis, Multiagent Systems

1 INTRODUCTION

Mechanism design is the study of procedures that incentivize self-interested actors to maximize global objectives such as social welfare. In recent years, the design of *algorithmic* mechanisms has become a prominent area of computer science. Research in this area has found applications in a wide range of domains, including policymaking[19], traffic management[8], online advertising[36], and even epidemiology[5].

In the traditional process of algorithmic mechanism design, a designer manually comes up with a specific mechanism with problem-specific guarantees. Such manual design can be brittle and laborious. Also, to ensure guarantees, these classical approaches tend to idealize various aspects of the agents and the mechanism [4, 12, 23, 25, 28, 34]. In particular, agents are assumed to have simple state spaces and follow mathematically simple behavioral models such as perfect rationality. The interaction between the mechanism and the agents is typically one-shot, rather than repeated.. The objectives that the mechanism is designed to enforce — for example, truth telling in auctions or optimality of resource allocation — are also idealized. Collectively, these assumptions significantly limit the applicability of these approaches.

Automated Mechanism Design [9] was proposed to alleviate some of these challenges. Here, one starts with a specification of the finite set of agents and an objective function. The discovery of a desirable mechanism is now phrased as an optimization problem that is solved automatically. While this approach was an important step in reducing the complexity of mechanism design, scalability was a challenge. As a result, the method was only applied to restricted settings in which the agent had small state spaces and repeated interaction between the agents and the mechanism was not permitted.

More recent work has proposed deep learning as a way to scale up automated mechanism design [4, 12, 19, 23, 25, 28, 34, 39]. Perhaps the most prominent example of this approach is the recent *AI Economist* [39], which uses deep neural networks to model both the mechanism (the policymaker) and the agents, and uses Deep Reinforcement Learning to learn a mechanism from simulations of agent behavior. The method was used to learn taxation mechanisms that maximize welfare in a sophisticated setting involving long-term reasoning and resource management by the agents.

A basic issue, however, with such methods is that the mechanisms that they learn are not *human interpretable*. Because applications of mechanism design often have human stakeholders, it is especially important that one be able to inspect and analyze mechanisms, especially ones produced by an automated process[7]. This is, however, impossible for mechanisms represented as neural networks.

In this paper, we propose *program synthesis* [15, 21] as an alternative approach to the automated mechanism design. Concretely, our mechanisms are represented as *programs* in a high-level domain-specific language (DSL). We model repeated interactions between the mechanism and the agents through a general Markov game[20]; the agents are assumed to use reinforcement learning to discover strategies in this game. In general, We assume a user-defined model of *behaviors* (which we discuss in Sec 3.2) that can result under each possible mechanism. Mechanism design is now phrased as the problem of synthesizing a programmatic mechanism such that all behaviors possible under this model achieve a globally desirable outcome (as defined by a user-defined objective).

Our method offers two key advantages over deep-learning approaches such as AI Economist. First, as programs in a high-level DSL, our mechanisms are human interpretable. Second, a user of

Proc.of the Adaptive and Learning Agents Workshop (ALA 2022), Cruz, Hayes, Silva, Santos (eds.), May 9-10, 2022, Online, https://ala2022.github.io/. 2022.

our method can use the DSL to incorporate prior knowledge about the structure and behavior of mechanisms. The use of such priors is difficult in deep-learning approaches. On the other hand, our DSLs can express significantly more complex mechanisms than those permitted in classical methods for automated mechanism design[9, 29].

An essential challenge with program synthesis is that it is a combinatorially hard problem. To address that challenge, we give an initial algorithm, based on a combination of stochastic search over program structures and Bayesian optimization of numerical parameters, for our version of this problem. We present two experimental domains with significantly different characteristics of the components described in Sec 3, and evaluate the algorithm (along with a variant and an ablation) on them. We find that the method returns sensible mechanisms with good performance, which suggests that it is indeed possible to design human interpretable mechanisms in programmatic form.

In summary, the contributions of this work are as follows[1]:

(1) We offer the first program-synthesis formulation of automated mechanism design. This formulation has the benefit of being more general than classical approaches to the problem. At the same time, it produces mechanisms that, unlike those in recent deep-learning approaches, are human interpretable and consistent with human-held prior knowledge.
(2) We present an initial algorithm, based on a combination of stochastic search and Bayesian optimization, for solving our program-synthesis problem.
(3) We offer a promising experimental evaluation over two domains with significantly different characteristics.

2 BACKGROUND

2.1 Multiagent Environments

We use Markov/Stochastic Games[20] as the formalism for describing multi-agent environments (from this point on, we use the terms Markov Games and Multi-Agent Environments interchangeably). While we assume fully observable environments in our description, our work easily applies to partially observable ones as well. A Markov Game M comprises k agents interacting with an environment with the following components:

- A joint state space S, and an initial state distribution ρ_0.
- A collection of action sets $A_1, A_2, \ldots, A_k$, one for each agent.
- A transition function $T : S \times A_1 \cup A_2 \cup \ldots \cup A_k \rightarrow PD(S)$ that takes a state and a given set of actions for each agent and outputs a distribution over the next state.
- A reward function for each agent $\mathcal{R}_i : S \times A_1 \cup A_2 \cup \ldots \cup A_k \rightarrow$ Real, that takes a state and a given set of actions for each agent and outputs a real valued reward.

A *strategy* $\pi_i : S \rightarrow PD(A_i)$ for the i-th agent is a (probabilistic) decision rule that, for a given state, outputs a distribution over the actions of the agent. A *strategy profile* $\boldsymbol{\pi} = (\pi_1, \pi_2 \ldots \pi_k)$ is a tuple of strategies, one for each agent.

A *trajectory* τ is a sequence of states, actions and rewards resulting from the execution of a strategy profile. We define the distribution of trajectories that could result from the execution of a given strategy profile $\boldsymbol{\pi}$ in the natural way, and denote this distribution by $M(\boldsymbol{\pi})$.

We provide a discussion on Multiagent Reinforcement Learning (MARL) based on the above in the supplementary material.

2.2 Program Synthesis

Program synthesis[15, 21] is the automated generation of programs according to some criterion such as performance or the fulfillment of a declarative specification. Programs are expressed, usually in small, tailor-made programming languages called Domain Specific Languages (DSLs), that define the space of possible programs that need to be searched, and their semantics. Symbolic techniques and others based on Satisfiability Modulo Theories solvers have seen successes in applications such as spreadsheet processing and distributed computing[16, 32, 35]. These approaches are concerned with formal constraints on the program and its behavior, including exactly matching the outputs from a given dataset of examples for the corresponding inputs. In quantitative synthesis[3], programs are synthesized to optimize an objective. Recently, there has been tremendous progress in using Machine Learning for quantitative synthesis, in supervised[11], unsupervised[13, 22] as well as Reinforcement learning[37] settings.

3 PROGRAMMATIC MECHANISM DESIGN

In this section, we present the formulation of the programmatic mechanism design problem we have motivated, and elucidate its various aspects using a small-scale traffic management domain. The goal here is to find a suitable mechanism from a search space of programmatically defined multiagent environments that, under certain assumptions about agent behavior, lead to outcomes that maximize a certain objective. The formal statement of the problem has three components, which we now describe.

3.1 Search Space

The search space $\mathcal{F}$ is a subset of all possible multiagent environments whose elements correspond to programs from a Domain Specific Language (DSL). We denote elements from this search space by $f = (f_T, f_R)$, with f_T and f_R being the transition and reward functions of f respectively. Let Π_f denote the space of (not necessarily Markovian) strategy profiles for environment f, and $\Pi_{\mathcal{F}} = \cup_{f \in \mathcal{F}} \Pi_f$, the set of all possible policies over all environments in $\mathcal{F}$. Note that different environments could have different state and action spaces.

Typically, we are interested in synthesizing only parts of the transition and reward functions, while the rest of the environment behaves in a fixed manner. These parts could be directly involved in determining the response (next state and rewards) of the environment, or could be an intermediate step in its computation. We might also wish to synthesize subroutines that can be reused, or even disparate parts of the environment's behavior. Also note that the programs being synthesized and their results could be shared between transition and reward functions (as is often the case with simulators). Many of these points are demonstrated in our illustrative example, as well as later in the experiments.

[1] Supplementary material is available at this link.

3.2 Agent Behavior Generator

An Agent Behavior Generator (henceforth simply Behavior Generator) encodes the assumptions about agent behavior that will result for any given environment. It is a function $\mathcal{B} : \mathcal{F} \rightarrow PD(\Pi_{\mathcal{F}})$ that maps a given environment to a distribution over policies from that environment representing the possible behaviors agents can exhibit in that environment. Note that the distribution must be over only that environment's policy space, even though the codomain of $\mathcal{B}$ can involve other policies as well. That is, $\forall f \in \mathcal{F},\ \mathcal{B}(f) \in PD(\Pi_f)$.

There are various forms a behavior generator could take. One set of possibilities are the Game Theoretic solution concepts that are traditionally studied in Multiagent Systems. For instance, it could be defined as an equiprobable distribution over all Nash Equilibria. A more practically relevant form in domains where finding Nash Equilibria is infeasible, is the result of executing a multiagent learning or planning algorithm, with the random choices (such as initializations) made by the algorithms inducing a distribution over possible agent behaviors. Such a definition can also impose computational constraints on the agent (bounded rationality) such as the amount of memory available, or representation capacity, as well as the extent to which it can interact with the environment (e.g how many episodes it is allowed for learning).

3.3 Objective Function

For a given environment from the search space, the objective function $\mathcal{J} : \cup_f(f, PD(\Pi_f)) \rightarrow$ Real, assigns a score to the environment and distribution over possible agent behaviors for that environment. Usually, the objective function involves criteria that are based on the resulting agent behavior. However, it could also involve properties of the environment itself, e.g the size of the program, or more generally, a cost based on the contents of the program.

With the above definitions in hand, we can formally state the Programmatic Mechanism Design problem as the following optimization problem:

$$\max_{f \in \mathcal{F}} \mathcal{J}\ (f, \mathcal{B}(f))$$

In other words, we would like to find a mechanism f from the search space of programs $\mathcal{F}$ that maximizes the objective function $\mathcal{J}$ under behavior arising from behavior generator $\mathcal{B}$. We summarize this problem statement in Figure 1.

3.4 Example: Traffic Domain

In this section, we present a problem involving a tabular domain (i.e discrete states and actions) as a concrete instantiation of the above concepts. There are two agents whose goal is to cross an intersection to get to their respective destinations. Two sets of traffic signals are present at the intersection, one for each agent's road. The goal is to design the operation of a set of traffic signals and the accompanying traffic rules to be followed by agents to allow for safe, fair and just passage for both agents (which we will shortly define more rigorously). We now present the details of the multiagent environment described (for an illustration, please see the supplementary material).

At each timestep, the agents can either move forward one square or stay at their current position. They are not allowed to turn from one road onto another at the intersection. Each agent gets a reward of +1 upon reaching the goal. If both agents arrive at the intersection at the same time, there is a 2.5% chance of collision, upon which they receive a -1 reward. Note that it is possible for the agents to meet at the intersection without colliding (in fact, that is more likely), and they can continue onward after the collision. There are possible penalties for the agents as described below, which lead to a reward of -1. Otherwise, the reward is zero. The initial signal states are chosen at random, though they are set to opposite values.

The components of the mechanism design problem are specified as follows:

3.4.1 Search Space. Each of the two traffic signals can be in one of two states: `True` and `False`. For the purpose of intuition, we represent `True` as green and `False` as red. The transition function for these signals take two Boolean inputs indicating the *current* states of the two signals (which we refer to as `sig1` and `sig2`), and produce one output indicating its next state. As in real life, traffic rules are implemented as penalties that are imposed under certain conditions. The penalty decision function for each of the two agents takes 4 boolean inputs, which include `sig1` and `sig2`, along with two additional Boolean values that indicate whether each agent has just crossed into the intersection at the current timestep, called `cross1` and `cross2`. For example, the 'usual' traffic control mechanism could be implemented by having `True` mean *go* and `False` mean *stop* (or more specifically, there will be a penalty if the agent doesn't stop when the signal is `False`).

The search space corresponds to a subset of possible 4-tuples of Boolean expressions from the DSL presented in Table 1. The 4 expressions in each given tuple implement the transition and penalty decision functions described above. Other than the operation of the signal and the penalty decision function, all environments in the search space are identical. Limiting ourselves to Normal Form expressions with at most 2 binary operators, along with a few other optimizations such as using commutativity to reduce double counting, it can be seen that we have 1,411,344 4-tuples of expressions.

The above is an example of how programs can impose priors, namely that effective mechanisms exist that can be expressed as Boolean formulas, which are easily interpretable, and also impose limits on the complexity of the rules. Further, we can impose priors on the specific behavior of the mechanism. For instance, we could restrict the search space of penalty decision functions to programs of the form `exp1&exp2`, where `exp1` and `exp2` are literals, that is the set of conjunctive Boolean expressions. This prior could express our knowledge that there is likely to be a useful set of rules that is a conjunction of some literals.

Let us consider as a running example, one (unconventional) mechanism from this search space, whose signal next-states are given by the expressions `sig1->!(sig1)` and `sig1->!(sig1)` respectively (note, they are in fact the same expression). The penalty rules are given by `cross1&(!(sig1))` and `cross2&sig2` respectively for the two agents. One can work out that these signals both take the same value on all steps but the first (when they are started with opposite values), and toggle between `True` and `False`. Similarly, it can be seen that the penalty rules penalize the first agent

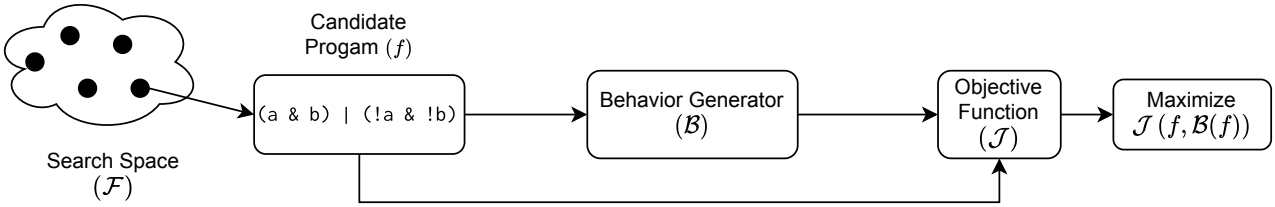


Figure 1: The Programmatic Mechanism Design Problem Formulation

Terminals	`True`, `False`	Logical True/False
Operators	!	Logical negation Operator
	&&	Logical AND Operator
	||	Logical OR Operator

Table 1: DSL of Boolean formulae for the traffic light domain.

for crossing into the intersection when the signal is `False`, while the other agent is penalized for crossing when the signal is `True`.

3.4.2 Behavior Generator. We would like our behavior generator to encode the idea that agents try to act in their own self interest to maximize their own rewards, i.e rational behavior. Although Game theoretic solution concepts like Nash Equilibria do so, they are notoriously difficult to compute or analyze[18] even in relatively simple domains such as this one. Therefore, we instead use an MARL algorithm to generate behaviors that are reasonably close to rational, and ask which mechanisms produce desirable outcomes under behavior generated by MARL. Specifically, we use Independent Q-Learning (IQL)[33] as the MARL algorithm.

3.4.3 Objective Function. For a given mechanism from the search space and a distribution over behaviors, the objective function for a particular trial is 0 if the agents meet at the intersection (regardless of whether a collision occurs), if either of them incur a penalty, or either of them is not able to reach the destination. Otherwise it is $e^{-\Delta t}$, where Δt is the random variable that gives difference in the number of timesteps that the agents took on that trial to get to their respective destinations. The overall objective is the expected value of the above over all trials.

Finding a mechanism maximizing this objective function means trying to ensure that the agents acting according to policies generated by the behavior generator don't meet at the intersection (*safety*), while simultaneously ensuring that the agents are able to reach their destinations in an equal amount of time (*fairness*), and without penalties. For the running example, as noted earlier, the agents are able to reach their destinations without incurring any penalty. However, we see that on every run, one of the agents waits for an additional timestep at the signal, so Δt is always 1, making the objective function $e^{-1} = 0.368$. In the experiments, we attempt to find mechanisms that perform as best as possible, which means asking if there is a safe and just mechanism that has $\Delta t = 0$.

4 SAMPLING-BASED SYNTHESIS OF PROGRAMMATIC MECHANISMS

We desire an approach that can search for desirable mechanisms in large, programmatic spaces of candidate mechanisms involving many combinatorial components. It should also be able to do so while only being able to sample the objective function, as well as the behavior generator's output. This rules out gradient-based optimization, as well as many existing program synthesis approaches that require access to the "source code" of the objective and behavior generator.

Markov Chain Monte-Carlo (MCMC) is a sampling-based search technique that is capable of producing a Markov Chain of candidates in which, in steady state, the likelihood of occurrence of a candidate increases with its performance according to the objective function being optimized. It also satisfies the requirement for being able to handle discrete spaces, and has been successfully used to synthesize optimized machine code[30]. Thus, we turn to this technique for our problem of synthesizing programmatic mechanisms.

There are 2 components in our MCMC-based approach, which we describe below:

Objective Evaluation. We defined the objective function for the problem as $\mathcal{J}(f, \mathcal{B}(f))$, where f is a given mechanism. As mentioned earlier, the behavior generator and objective function can, in general, only be estimated by sampling. We assume N samples are used, and denote the resulting estimate as $\hat{J}(f)$.

Proposal distribution. The proposal distribution, $p(f'|f)$ "rewrites" a given candidate mechanism f to make small, random changes to it to produce a proposal f'. These changes could alter the structure of the program, which defines the organization of operators, operands and other programmatic constructs, in a syntactically consistent way. Additionally, they could also change the values of the operands, such as constants (e.g `True`/`False`, or numeric parameters). These changes can then be accepted or rejected based on the change in the performance.

The MCMC procedure starts with a random candidate F_0. It then constructs a Markov chain $F_0, F_1 \ldots$ of candidate mechanisms using a procedure based on the Metropolis-Hastings algorithm[17], which is performed by iteratively sampling a proposal from the proposal distribution, and accepting or rejecting it based on whether it performs better than the current program. If the proposal is accepted, then it becomes the current program.

Under the condition of *reversibility* of the proposal distribution, it can be shown that the resulting Markov Chain $F_0, F_1 \ldots$ has a steady state distribution where the probability of a candidate mechanism f

Algorithm 1 MCMC-based Synthesis Procedure using Bayesian Optimization (BO) for tuning parameters.

function SYNTHESIZE-BOMCMC
  $F_0 \leftarrow$ Random mechanism
  $C \leftarrow [F_0]$
  for $i = 0, 1 \ldots N_{itr} - 1$ **do**
    $f'_s \sim p(f'|F_i)$
    $f' \leftarrow \text{TuneParameters}(f'_s)$
    $F_{i+1} \leftarrow f'$
    if $\hat{J}(f') < \hat{J}(F_i)$ **then**
      $r \sim \text{Bernoulli}\left(e^{\beta(\hat{J}(f') - \hat{J}(F_i))}\right)$
      if $r == 0$ **then**
        $F_{i+1} \leftarrow F_i$
      end if
    end if
  end for
end function

is proportional to $e^{\beta \hat{J}(f)}$. In practice, this means that, given enough iterations, good mechanisms will eventually be found.

4.1 Bayesian Optimization for Efficient Parameter Tuning

The above MCMC approach is able to search through both combinatorial spaces of program structures, or skeletons, as well as continuous spaces of parameters that they might involve, with an appropriately chosen proposal distribution. However, the MCMC process might not spend enough time tuning the parameters in a given program (before changing the structure of the program, which could render existing parameter values useless) in order to assess the best possible performance for a given program structure. While the best parameter configurations will eventually be discovered, the frequent switching of the program structure would substantially delay this process, an effect that we observed in our experiments.

To improve the efficiency of the search, we propose to add another level of search that tunes the parameters of the program proposed by the previous level. This tuning is accomplished by using Bayesian Optimization (BO)[31], where the input space of the BO algorithm is the space of possible program parameters, and the output, or objective is the same as the objective function as defined in Section 3. BO algorithms have been known to rapidly find good solutions to optimization problems, while working with noisy estimates. We make use of these capabilities by significantly reducing the number of samples used to estimate the objective while tuning. The BO algorithm is encapsulated in a subroutine TuneParams(f). We defer the discussion of this subroutine to the supplementary material. The entire procedure incorporating BO search is summarized in Algorithm 1.

5 EXPERIMENTS

Through the experiments below, we validate the effectiveness of our methods under variations in the following factors:

(1) The characteristics and complexity of the domain.
(2) Structure of the programs being synthesized.
(3) The Behavior Generator.

We summarize the above information in Table 2 for each of the two sets of experiments we describe below.

	Traffic Domain	Hunt-and-Gather
State/Action Space	Discrete/Discrete	Continuous/Discrete
Program Structure	Boolean Expressions	Decision Lists with Boolean and Numeric Conditions
Behavior Generator	MARL Algorithm	Pre-defined Behavior

Table 2: A summary of the characteristics and complexity of the domain, structure of the programs being synthesized, and the behavior generator used.

	Agent	Expression
Signals	1	`!(sig1)`
	2	`!(sig2)`
Penalty	1	`cross1&(!(sig1))`
	2	`cross2&(!(sig2))`

Table 3: Boolean expressions synthesized for the regular traffic lights problem.

5.1 Traffic Domain

We begin by evaluating our approach on the traffic domain introduced in Section 3.4, and present more experimental details in the supplementary material.

One of the top performing expressions found are shown in Table 3. We see that the signals toggle between true and false each timestep, while penalties are imposed when an agent crosses into the intersection while the signal state is false. Assuming that true corresponds to green, and false to red, these signals and rules are the same as real-life traffic rules. This shows that sensible mechanisms can be found even when the behavior generator involves learning rules, and in combinatorial search spaces such as our search space of Boolean expressions.

Interestingly, by evaluating the objective as in Section 3.4.3 this mechanism too has performance 0.368, just like the example in Section 3.4. This shows that while the latter is an unusual way to design a signal, it is nevertheless just as valid and just as useful as the conventional signal.

5.2 Hunt and Gather Domain

Now, we introduce a domain with significant complexity based on the domain by Zheng et al. [39], which we similarly call the *hunt and gather* domain. In this domain, 3 agents operate in a gridworld, where they can move around the map, gather resources (wood and stone) and use them to build houses. This is illustrated in Figure 2. The agents are part of an economy, where they receive coin for building houses, or by selling resources to other agents (alternatively, they can spend money to buy resources from other agents). Each episode is divided into "fiscal years", at the end of

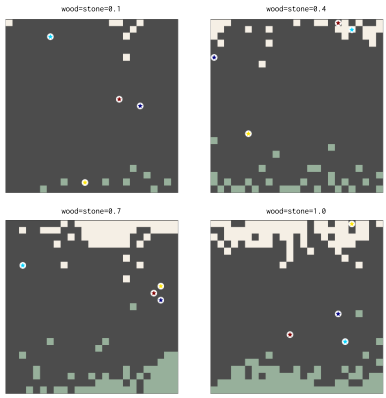


Figure 2: The Hunt and Gather domain: In each image, agents are denoted by star markers of different colors. Green and Beige tiles represent the resources wood and stone respectively. The images show different possible environments with varying abundances of each resource, with the normalized abundances mentioned above the image.

which tax is collected from the agents based on their income, and the total tax amount is redistributed to all agents equally.

In each episode, the agents could face different environmental conditions, particularly the abundances of the two resources, wood and stone. The abundances are set at the beginning of every episode from a uniform distribution over a grid of 25 values, and remain fixed through the episode. It should be clarified here that resources take time to replenish once they are collected by an agent. The abundance of a resource refers to the number of sites on the map where the resource can spawn.

The goal is to design a mechanism that acts as an economic policy to improve the welfare of the agents (which we define later) across these conditions. It can do so by setting the following parameters at the beginning of the episode based on the observed abundances of the resources: 1) Taxes: There are two different tax schedules to choose from, one with low tax rates, and the other with high rates. 2) Market prices: the agents buy and sell resources at a fixed price decided by the economic policy. There are 3 price levels (low, moderate and high) to choose from, separately for each resource.

5.2.1 Search Space. We wish to find a program that decides at the beginning of the episode, the taxes and prices as described above, with the mechanisms from the search space being as described above and identical, save for this program. Each program takes as input two numeric values between 0 and 1 called `wood` and `stone`. These represent the observed abundances of wood and stone respectively, normalized to that range. The output consists of 3 values, or output variables representing the choices above, called `tax`, `price_wood`, and `price_stone`.

The program takes the structure of a decision list, being a sequence of statements of the form:

```
if(condition) then: variable:=value
```

where the condition is a Boolean expression involving comparisons of the input values. The resulting conditional statement sets the value of one of the output variables listed above to a value representing an appropriate choice from the ones described above, which we assume to be from a set of the form $\{1, 2, \ldots\}$. To reduce program complexity, the output variables are all assigned a default value, so programs need not set every output variable. An instance of such a statement is as follows:

```
if((wood<0.5)&&(stone<0.5)) then:
    tax:=2
```

The above statement sets a high tax rate when the (normalized) abundances of both wood and stone are observed to be less than 0.5.

Considering programs of up to 4 statements, we see that the search space is much more complex than in the Traffic domain. Not only are there Boolean expressions, they also involve numeric parameters. There are a similarly large number of program *skeletons* (sets of programs differing only in the values of the parameters involved). Each skeleton is further associated with a continuous, usually multi-dimensional space of parameters to choose from.

5.2.2 Behavior Generator. We use a set of predefined, fixed behaviors as the behavior generator (meaning that the distribution over generated behaviors is the same regardless of the mechanism, and has all probability mass assigned to one particular policy), which is as follows. For two of the three agents, the behavior is to continually collect a certain amount of a resource (one agent collects wood and the other collects stone) and list it for sale. The other agent buys the wood and stone that were listed for sale by the other two agents, and uses them to build houses. Thus, the agents form a supply chain that converts resources to houses. The problem can then be thought of as maximizing the welfare of the agents who are carrying out their roles in the above supply chain. Note that since the behavior generator does not make use of the agents' rewards, they do not need to be specified, though a straightforward choice as in [39] is the utility experienced by each agent as defined in the next section.

5.2.3 Objective function. Each agent receives utility from the coin it earns during an episode. Gathering resources and building houses takes labor, which negatively impacts the utility of agents. Each agent accrues a certain net utility at the end of an episode. Let the vector of every agent's utilities be $\mathbf{u}$. Now, we define a performance metric as $1 - \text{gini}(\mathbf{u})$, where $\text{gini}(\cdot)$ refers to the standard Gini Index. Thus, it measures welfare in terms of equality of utility, with 1.0 being the best possible outcome (all utilities equal), and 0.0 being the worst .This performance metric is averaged over all possible values for the resource abundances to produce the final objective function. When evaluating using sampling, one episode per environment configuration is executed and the performances averaged to produce one sample of the final objective.

5.2.4 Experiment Setup. We apply the following approaches to the problem as described above:

BO-MCMC. This is the BO-MCMC algorithm as described in Algorithm 1. We use a proposal distribution that chooses a statement at random and adds, deletes or rewrites the operators, operands with randomly chosen symbols. Any new parameters that are added are set to 0.5, which are tuned along with existing parameters by the BO algorithm.

BO-MCMC - Cur-Rand. This is also the BO-MCMC algorithm as before, but performs an additional BO search starting from an initial value for the program parameters as they were present in the previous iteration of the search. For newly introduced parameters, a random value is used. The tuned parameters are then the best performing values among those obtained during this additional search, as well as the usual search starting from a random initialization.

MCMC. As an ablation experiment, this is a version that does not use BO to tune program parameters, but rather makes parameter tuning a part of the MCMC search. That is, the proposal distribution has a chance to make changes to the program parameters, which include increments or decrements of 0.05 of any one random parameter.

Decision Tree Oracle. We train a Decision Tree (DT) oracle in order to establish an approximate upper bound on the performance attainable while maintaining interpretability. We explain how it is trained, and why it is an oracle in the supplementary material.

We describe hyperparameters, proposal distribution and other details in the supplementary material. We used BoTorch[1] to implement our BO algorithms and Scikit-Learn[26] for Decision Tree learning.

5.2.5 Results. In Figure 3, we present the results obtained from the above approaches along with the DT oracle and a "constant" baseline policy that always uses the default values for taxes and prices. Figure 3 also presents one of the top programs synthesized by each approach. The graphs show the best performance attained by any candidate mechanism up to a particular iteration. The methods that are being compared do different amounts of work in each iteration, in terms of the number of environment runs executed. In order to facilitate comparison, we plot this best performance for each method as a function of an "effort-adjusted" step number. This effort-adjusted step multiplies the iteration number on the x-axis by a factor given by the ratio between the number of runs executed by the method in question and the plain MCMC approach (making the factor 1 for MCMC). We make the following observations from these results:

(1) Both BO-MCMC as well as BO-MCMC - Cur-Rand are able to find good mechanisms with performance close to the DT oracle, and quickly find mechanisms that are significantly better than the constant policy baseline.
(2) While the plain MCMC approach is also able to surpass the baseline, it is unable to make significant progress beyond the baseline. This demonstrates that the use of Bayesian Optimization for parameter tuning is an important contributor to the effectiveness of the algorithm.
(3) BO-MCMC-Cur-Rand is able to improve its performance more quickly in the initial iterations, indicating that reusing previously tuned parameters can potentially hasten the discovery of effective programs. However, later its average performance remains below that of BO-MCMC.
(4) The programs found are also highly interpretable. The program found by BO-MCMC (Figure 3) is highly intuive. Although the program found by Cur-Rand is longer and contains redundant statements and conditions, a quick inspection suffices to deduce that it raises the price of wood if its availability is lower than a certain value, while the price of stone is constantly set high.

The above observations establish that our approach is able to find good mechanisms on problems with significant complexity. It is also able to handle complex program structures involving continuous valued parameters, which are necessary in many real-world situations.

6 RELATED WORK

We now compare aspects of the programmatic mechanism design problem with existing work across mechanism design, as well as program synthesis work connected to our problem.

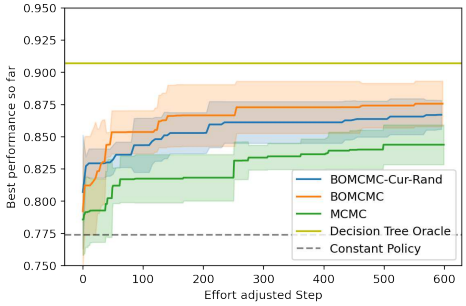
Algorithmic Mechanism Design. Algorithmic mechanism design[24, 36] (separate from Automated Mechanism Design, which is discussed below) takes a computational perspective to mechanism design with a focus on establishing theoretical guarantees such as optimality. Although it employs analytic tools from theoretical computer science towards this end, the design process remains manual. Further, it does not consider the use of programs as an interpretable representation for mechanisms, or the synthesis of such programs.

Automated Mechanism Design. Automated Mechanism Design [9, 10, 29] aims to develop methods to solve problems involving multiple agents with private types. Although these works are more general than their predecessors, they are nevertheless restricted to certain kinds of settings (those involving voting, auctions, or some other form of allocation), models of agent behavior (such as individual rationality) and design objectives (e.g truthful reporting of types/preferences).

Several solution approaches for Automated Mechanism Design, including those employing Machine Learning [12, 23, 28], employ the *revelation principle*, which allows the design algorithm to assume that desirable (e.g truthful) behavior is optimal, avoiding the need to learn agent behaviors. On the other hand, approaches such as [2, 6, 27] employ machine learning to learn rational behaviors, which are used to optimize the mechanism. Our general problem framework is able to seamlessly handle both these classes of ideas by using the appropriate behavior generator, allowing the solution approach to be agnostic to such details. Indeed, we have demonstrated this in the above experiments, with learning being used for agent behavior in section 3.4, while the fixed behavior used in section 5.2 can be seen as applying the revelation principle for that behavior.

Adaptive Mechanism Design [25] considers auction design problems involving unknown and varying bidder behavior. We have already tested an aspect of such problems in our work, namely changing the mechanism's behavior based on observed data (resource abundances). In principle, our work also applies to situations where conditions (e.g bidder behavior) change continually.

Several recent works use RL in some form for designing mechanisms [4, 34, 39]. Perhaps the closest work to ours in terms of generality of the mechanism design problem considered is [39]. The approach presented there aims to learn a taxation policy in the form of a Neural Network using Deep MARL in a domain involving large, continuous state spaces. Other approaches that learn agent



Method	Discovered Program	Performance

BO-MCMC	`if (((stone<0.525)|(stone<0.328)))`
	`  then price_Stone = 3;`
	`if (((wood<0.778)&(wood>0.0)))`
	`  then price_Wood = 3;`
		0.88

BO-MCMC-Cur-Rand	`if (((stone>0.158)&(stone<0.551)))`
	`  then price_Stone = 1;`
	`if ((wood<0.563))`
	`  then price_Wood = 3;`
	`if (((stone>0.0)&(stone<1.0)))`
	`  then price_Stone = 3;`
	`if ((wood<0.0))`
	`  then price_Stone = 1;`
		0.88

Figure 3: Left: Performance of best program found up to each step, over 5 different runs for each method. The step numbers are normalized to an effort adjusted step which is equivalent to the amount of work done. Right: Best performing programs discovered by BO-MCMC and BO-MCMC-Cur-Rand in particular runs.

behavior often solve two-level learning problems[2, 6], while [39] jointly learn the agent behaviors, as well as the taxation policy using MARL (essentially treating the mechanism as an agent), thus solving a one-level problem, and as a result achieving significant scalability. While there is potential to harness this scalability for programmatic mechanism design in the case of a learning-based behavior generator by combining this approach with program synthesis approaches that convert learned neural models to programs such as [37], there are still some challenges involved: First is that a suitable reward function is needed to guide the reinforcement learning of the mechanism. In many situations, feedback on the performance of the mechanism can only be obtained at the end of a trajectory, leading to sparse rewards that require further reward engineering steps to enable RL algorithms to learn properly. Next, representing many kinds of programs using neural networks can require specialized architectures such as Neural Turing Machines, while Reinforcement Learning has been shown to be difficult using such models[38].

Program Synthesis for Mechanism Design. [14] consider the synthesis of Linear Temporal Logic (LTL) mechanisms that satisfy a given LTL specification under particular definitions of rational agent behavior. However, the work is limited to small, discrete systems due to the sheer complexity of the LTL synthesis problem. It is also limited to programs that are temporal logic formulas, rather than from any given DSL as we consider here. To the best of our knowledge, we are the first to present a general problem framework for programmatic mechanism design, as well as a solution approach that is applicable generally.

7 CONCLUSION AND FUTURE WORK

We have developed a novel problem framework that considers mechanism design in programmatic form. It is able to deal with many of the messy realities that arise in practice, greatly broadening its scope in terms of the structure and complexity of the problem, assumptions about agent behavior, and design objectives. Our solution approach based on stochastic search is already able to solve problems of significant complexity. We see this achievement as a successful first step towards using program synthesis for designing interpretable mechanisms.

A natural question that arises is that of scalability. A major limitation in our proof of concept in the larger domain is that the behaviors are fixed. This property essentially reduces the problem to a single-level optimization, rather than the fully general bi-level one where the agent behaviors are also learned for each candidate mechanism considered. One possibility for circumventing this hard bi-level problem alluded to in section 6, is learning a neural network mechanism using a single-level approach such as in [39], and converting the learned model to a program. Despite the potential RL challenges, this is an exciting direction for future work, as such approaches can avoid the need to perform two-level learning.

Speeding up MARL algorithms in order to allow quick evaluations of the behavior generator will be key to achieving scalability when using methods such as ours. To this end, it may be feasible to learn parameterized, or task-conditioned policies for the agents, where the task is the program from the mechanism itself with an embedding applied to it.

ACKNOWLEDGMENTS

This work has taken place in the Learning Agents Research Group (LARG) and the Trishul Laboratory at the Department of Computer Science, The University of Texas at Austin. LARG research is supported in part by the National Science Foundation (CPS-1739964, IIS-1724157, FAIN-2019844), the Office of Naval Research (N00014-18-2243), Army Research Office (W911NF-19-2-0333), DARPA, Lockheed Martin, General Motors, Bosch, and Good Systems, a research grand challenge at the University of Texas at Austin. Research in Trishul on this topic was supported by the National Science Foundation (CCF-1704883 and CCF-1918651). The views and conclusions contained in this document are those of the authors alone. Peter Stone serves as the Executive Director of Sony AI America and receives financial compensation for this work. The terms of this arrangement have been reviewed and approved by the University of Texas at Austin in accordance with its policy on objectivity in research.

REFERENCES

[1] Maximilian Balandat, Brian Karrer, Daniel Jiang, Samuel Daulton, Ben Letham, Andrew G Wilson, and Eytan Bakshy. 2020. BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 21524–21538. https://proceedings.neurips.cc/paper/2020/file/f5b1b89d98b7286673128a5fb112cb9a-Paper.pdf

[2] David Balduzzi, Marta Garnelo, Yoram Bachrach, Wojciech Czarnecki, Julien Perolat, Max Jaderberg, and Thore Graepel. 2019. Open-ended learning in symmetric zero-sum games. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 434–443. https:

//proceedings.mlr.press/v97/balduzzi19a.html

[3] Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. 2009. Better Quality in Synthesis through Quantitative Objectives. In *Proceedings of the 21st International Conference on Computer Aided Verification* (Grenoble, France) *(CAV '09)*. Springer-Verlag, Berlin, Heidelberg, 140–156. https://doi.org/10.1007/978-3-642-02658-4_14

[4] Gianluca Brero, Alon Eden, Matthias Gerstgrasser, David Parkes, and Duncan Rheingans-Yoo. 2021. Reinforcement Learning of Sequential Price Mechanisms. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 6 (May 2021), 5219–5227. https://ojs.aaai.org/index.php/AAAI/article/view/16659

[5] Roberto Capobianco, Varun Kompella, James Ault, Guni Sharon, Stacy Jong, Spencer Fox, Lauren Meyers, Peter R. Wurman, and Peter Stone. 2021. Agent-Based Markov Modeling for Improved COVID-19 Mitigation Policies. *The Journal of Artificial Intelligence Research (JAIR)* 71 (August 2021), 953–92.

[6] Micah Carroll, Rohin Shah, Mark K Ho, Tom Griffiths, Sanjit Seshia, Pieter Abbeel, and Anca Dragan. 2019. On the Utility of Learning about Humans for Human-AI Coordination. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/f5b1b89d98b7286673128a5fb112cb9a-Paper.pdf

[7] Rich Caruana, Yin Lou, Johannes Gehrke, Paul Koch, Marc Sturm, and Noemie Elhadad. 2015. Intelligible Models for HealthCare: Predicting Pneumonia Risk and Hospital 30-Day Readmission. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Sydney, NSW, Australia) *(KDD '15)*. Association for Computing Machinery, New York, NY, USA, 1721–1730. https://doi.org/10.1145/2783258.2788613

[8] Haipeng Chen, Bo An, Guni Sharon, Josiah P. Hanna, Peter Stone, Chunyan Miao, and Yeng Chai Soh. 2018. DyETC: Dynamic Electronic Toll Collection for Traffic Congestion Alleviation. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI-18)* (New Orleans, Lousiana, USA).

[9] Vincent Conitzer and Tuomas Sandholm. 2003. Automated Mechanism Design: Complexity Results Stemming from the Single-Agent Setting. In *Proceedings of the 5th International Conference on Electronic Commerce* (Pittsburgh, Pennsylvania, USA) *(ICEC '03)*. Association for Computing Machinery, New York, NY, USA, 17–24. https://doi.org/10.1145/948005.948008

[10] Vincent Conitzer and Tuomas Sandholm. 2007. Incremental Mechanism Design. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence* (Hyderabad, India) *(IJCAI'07)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1251–1256.

[11] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 990–998.

[12] Paul Duetting, Zhe Feng, Harikrishna Narasimhan, David Parkes, and Sai Srivatsa Ravindranath. 2019. Optimal Auctions through Deep Learning. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 1706–1715. https://proceedings.mlr.press/v97/duetting19a.html

[13] Kevin Ellis, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2015. Unsupervised Learning by Program Synthesis. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 973–981.

[14] Dana Fisman, Orna Kupferman, and Yoad Lustig. 2010. Rational Synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems*, Javier Esparza and Rupak Majumdar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–204.

[15] Cordell Green. 1969. Application of Theorem Proving to Problem Solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence* (Washington, DC) *(IJCAI'69)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 219–239.

[16] William R. Harris and Sumit Gulwani. 2011. Spreadsheet Table Transformations from Examples. *SIGPLAN Not.* 46, 6 (jun 2011), 317–328. https://doi.org/10.1145/1993316.1993536

[17] W. K. Hastings. 1970. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika* 57, 1 (1970), 97–109. http://www.jstor.org/stable/2334940

[18] Junling Hu and Michael P. Wellman. 2003. Nash Q-Learning for General-Sum Stochastic Games. *J. Mach. Learn. Res.* 4, null (Dec. 2003), 1039–1069.

[19] Raphael Koster, Jan Balaguer, Andrea Tacchetti, Ari Weinstein, Tina Zhu, Oliver Hauser, Duncan Williams, Lucy Campbell-Gillingham, Phoebe Thacker, Matthew Botvinick, and Christopher Summerfield. 2022. Human-centered mechanism design with Democratic AI. arXiv:2201.11441 [cs.AI]

[20] Michael L. Littman. 1994. Markov Games as a Framework for Multi-Agent Reinforcement Learning. In *Proceedings of the Eleventh International Conference on International Conference on Machine Learning* (New Brunswick, NJ, USA) *(ICML'94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 157–163.

[21] Zohar Manna and Richard J. Waldinger. 1971. Toward Automatic Program Synthesis. *Commun. ACM* 14, 3 (mar 1971), 151–165. https://doi.org/10.1145/362566.362568

[22] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2018. Neural Sketch Learning for Conditional Program Generation. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HkfXMz-Ab

[23] Harikrishna Narasimhan and David C. Parkes. 2016. A General Statistical Framework for Designing Strategy-Proof Assignment Mechanisms *(UAI'16)*. AUAI Press, Arlington, Virginia, USA, 527–536.

[24] Noam Nisan and Amir Ronen. 2001. Algorithmic Mechanism Design. *Games and Economic Behavior* 35, 1 (2001), 166–196. https://doi.org/10.1006/game.1999.0790

[25] David Pardoe, Peter Stone, Maytal Saar-Tsechansky, and Kerem Tomak. 2006. Adaptive Mechanism Design: A Metalearning Approach. In *The Eighth International Conference on Electronic Commerce*. 92–102.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[27] S Phelps, P McBurney, and Parsons S. 2010. Evolutionary mechanism design: a review. *Auton Agent Multi-Agent Syst 21* (2010). https://doi.org/10.1007/s10458-009-9108-7

[28] Ariel D. Procaccia, Aviv Zohar, Yoni Peleg, and Jeffrey S. Rosenschein. 2009. The Learnability of Voting Rules. 173, 12-13 (aug 2009), 1133–1149. https://doi.org/10.1016/j.artint.2009.03.003

[29] Tuomas Sandholm and Anton Likhodedov. 2015. Automated Design of Revenue-Maximizing Combinatorial Auctions. *Oper. Res.* 63, 5 (oct 2015), 1000–1025.

[30] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. *SIGARCH Comput. Archit. News* 41, 1 (March 2013), 305–316. https://doi.org/10.1145/2490301.2451150

[31] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. 2016. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proc. IEEE* 104, 1 (2016), 148–175. https://doi.org/10.1109/JPROC.2015.2494218

[32] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

[33] Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. 2017. Multiagent cooperation and competition with deep reinforcement learning. *PLOS ONE* 12, 4 (04 2017), 1–15. https://doi.org/10.1371/journal.pone.0172395

[34] Pingzhong Tang. 2017. Reinforcement mechanism design. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*. 5146–5150. https://doi.org/10.24963/ijcai.2017/739

[35] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 287–296. https://doi.org/10.1145/2491956.2462174

[36] Hal R. Varian. 2009. Online Ad Auctions. *American Economic Review* 99, 2 (May 2009), 430–34. https://doi.org/10.1257/aer.99.2.430

[37] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. 2018. Programmatically Interpretable Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 5045–5054. https://proceedings.mlr.press/v80/verma18a.html

[38] Wojciech Zaremba and Ilya Sutskever. 2016. Reinforcement Learning Neural Turing Machines - Revised. arXiv:1505.00521 [cs.LG]

[39] Stephan Zheng, Alexander Trott, Sunil Srinivasa, Nikhil Naik, Melvin Gruesbeck, David C. Parkes, and Richard Socher. 2020. The AI Economist: Improving Equality and Productivity with AI-Driven Tax Policies. arXiv:2004.13332 [econ.GN]