Synthesizing Interpretable Strategies for Solving Puzzle Games

Eric Butler
edbutler@cs.washington.edu
Paul G. Allen School of Computer Science and Engineering
University of Washington

Emina Torlak
emina@cs.washington.edu
Paul G. Allen School of Computer Science and Engineering
University of Washington

Zoran Popović
zoran@cs.washington.edu
Paul G. Allen School of Computer Science and Engineering
University of Washington

ABSTRACT

Understanding how players interact with games is an important challenge for designers. When playing games centered around problem solving, such as logic puzzles like Sudoku or Nonograms, people employ a rich structure of domain-specific knowledge and strategies that are not obvious from the description of a game's rules. This paper explores automatic discovery of player-oriented knowledge and strategies, with the goal of enabling applications ranging from difficulty estimation to puzzle generation to game progression analysis. Using the popular puzzle game Nonograms as our target domain, we present a new system for learning human-interpretable rules for solving these puzzles. The system uses program synthesis, powered by an SMT solver, as the primary learning mechanism. The learned rules are represented as programs in a domain-specific language for condition-action rules. Given game mechanics and a training set of small Nonograms puzzles, our system is able to learn sound, concise rules that generalize to a test set of large real-world puzzles. We show that the learned rules outperform documented strategies for Nonograms drawn from tutorials and guides, both in terms of coverage and quality.

CCS CONCEPTS

•Computing methodologies →Artificial intelligence;

KEYWORDS

Automated Game Analysis, Program Synthesis, Artificial Intelligence

ACM Reference format:
Eric Butler, Emina Torlak, and Zoran Popović. 2017. Synthesizing Interpretable Strategies for Solving Puzzle Games. In *Proceedings of FDG'17, Hyannis, MA, USA, August 14-17, 2017,* 12 pages.
DOI: 10.1145/3102071.3102084

1 INTRODUCTION

Automated game analysis is a growing research area that aims to uncover designer-relevant information about games without human testing [21, 24, 31, 35], which can be particularly advantageous in situations where human testing is too expensive or of limited effectiveness [38]. One potential use is automatically understanding game strategies: if we can analyze the rules of the game and automatically deduce what the effective player strategies are, we can support a range of intelligent tools for design feedback, content generation, or difficulty estimation. For humans to use these computer-generated strategies, they need the strategies to be both effective in the domain of interest and concisely expressed so a designer can understand the whole strategy in their mind.

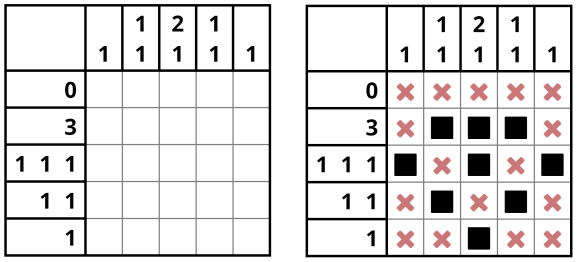


Figure 1: An example of a Nonograms puzzle, with the start state on the left and completed puzzle on the right. The numbered *hints* describe how many contiguous blocks of *cells* are filled with true. We mark cells filled with true as a black square and cells filled with false as a red X. We use the X to distinguish from unknown cells, which are blank.

Modeling player interaction is challenging because the game mechanics do not fully capture how human players might approach the game. This is true for all games, but especially for logic puzzle games such as Sudoku or Nonograms. These puzzles are straightforward for a computer to solve mechanically by reduction to SAT or brute-force search, but humans solve them in very different ways. Rather than search, human players use a collection of interconnected strategies that allow them to make progress without guessing. For example, there are dozens of documented strategies for Sudoku[1] [33], and puzzle designers construct puzzles and rank their difficulty based on which of these strategies are used [34]. The strategies take the form of interpretable condition-action rules that specify (1) where a move can be made, (2) what (easy-to-check) conditions must hold to make it, and (3) how to make the move. Human players solve puzzles by looking for opportunities to apply these strategies rather than by manual deduction or search in the problem space.

Learning and applying these strategies is the core of human expertise in the game. Understanding these strategies as a designer allows one to effectively build and analyze puzzles and progressions. While many such strategies can be uncovered through user testing

FDG'17, Hyannis, MA, USA

978-1-4503-5319-9/17/08…$15.00
DOI: 10.1145/3102071.3102084

[1] http://www.sudokuwiki.org/Strategy_Families contains a rather extensive list.

and designer introspection, they may not effectively cover the puzzle design space or be the most useful or simple strategies. Designers can benefit from tools that, given a game's rules, can help understand its strategy space. While we would prefer finding strategies people use, as a necessary step, we must find strategies we can easily understand and can demonstrate are effective for the problem-solving task.

In this paper, we investigate automatically learning human-friendly game playing strategies expressed as condition-action rules. We focus on the popular puzzle game Nonograms, also known as Picross, Hanjie, O'Ekaki, or Paint-by-Numbers. A nonogram (see Figure 1) is a puzzle in which the player must fill in a grid of cells with either true (black square) or false. Integer *hints* are given for each row and column that specify how many contiguous segments of filled cells exist in each row or column. Solutions often form interpretable pictures, though this is not necessary. By convention, nonograms have unique solutions and, like other logic puzzles such as Sudoku, can be solved by deducing parts of the answer in any order. Also like many logic puzzles, Nonograms is, for arbitrarily sized puzzles, NP-complete [37], but typical puzzles used in commercial books and games can often be solved with a fixed set of greedy strategies. Even for puzzles that require some tricky moves, greedy strategies can suffice to find a large portion of the solution.

A key challenge in learning these strategies is *interpretability*: the learned strategies need to be expressed in terms of game-specific concepts meaningful to human players, such as, in the case of Nonograms, the numerical hints or state of the board. To address this challenge, we developed a new domain-specific programming language (DSL) for modeling interpretable condition-action rules. In contrast to previous DSLs designed for modeling games [5, 18, 23, 24, 27], which focus on encoding the rules and representations of the game, our DSL focuses on capturing the strategies that a player can use when puzzle solving. Thus, the constructs of the language are game-specific notions, such as hint values and the current state of the board. In this way, we frame the problem of discovering player strategies for Nonograms as the problem of finding programs in our DSL that represent (logically) sound, interpretable condition-action rules. This soundness is critical and difficult to ensure: rules should be valid moves that respect the laws of the game, and complex constraints must hold for this to be the case. For this reason, we use a constraint solver at the core of our learning mechanism.

Learning condition-action rules for Nonograms involves solving three core technical problems: (1) automatically discovering specifications for potential strategies, (2) finding sound rules that implement those specifications, and (3) ensuring that the learned rules learned are general yet concise. To tackle these challenges, we built a system using *program synthesis* [9] as a learning mechanism. The system takes as input the game mechanics, a set of small training puzzles, a DSL that expresses the concepts available for rules, and a cost function that measures rule conciseness. Given these inputs, it learns an optimal set of sound rules that generalize to large real-world puzzles. The system works in three steps. First, it automatically obtains potential specifications for rules by enumerating over all possible game states up to a small fixed size. Next, it uses an off-the-shelf program synthesis tool [36] powered by an SMT (Satisfiability Modulo Theories) solver to find programs that encode sound rules for these specifications. Finally, it reduces the resulting large set of rules to an optimal subset that strikes a balance between game-state coverage and conciseness according to the given cost function. We evaluate the system by comparing its output to documented strategies for Nonograms drawn from tutorials and guides, finding that it fully recovers many of these control rules and covers nearly all of game states covered by the remaining rules.

Our approach to learning interpretable strategies by representing them as condition-action rules defined over domain-specific concepts is motivated by cognitive psychology and education research. In this approach, a set of rules represents (a part of) domain-specific *procedural knowledge* of the puzzle game—the strategies a person takes when solving problems. Such domain-specific knowledge is crucial for expert problem solving in a variety of domains [3, 8], from math to chess to professional activities. The DSL defines the (domain-specific) concepts and objects to which the player can refer when solving puzzles, thus constraining the space of strategies that can be learned to human-friendly ones. Our system takes this space as input provided by a designer, and produces the procedural knowledge to be used by players. Thus, the designer can define (and iterate on) the concepts over which rules are defined. The designer also provides a cost function (defined over the syntax of the DSL) that measures interpretability in terms of rule complexity, which allows our system to bias the learning process toward concise, interpretable rules. The eventual goal of this line of research is automatically discovering strategies that players are likely to use. In this work, we focus on the immediate task of finding human-interpretable rules in a structure compatible with evidence of how players behave.

While our implementation and evaluation focuses on Nonograms, the system makes relatively weak assumptions (discussed in detail) about the DSL used as input. Variations could be used, and we expect DSLs representing other logic puzzles could be used in the system. And while many parts of the learning mechanism are specific to logic puzzles, we expect the approach of using program synthesis for learning of human-interpretable strategies to apply more broadly, especially to domains with well-structured problem solving (even beyond games, such as solving algebraic equations [7]).

In summary, this paper makes the following contributions:

- We identify how domain-specific programming languages can be used to represent the problem-solving process for puzzle games in a human-interpretable way.
- We describe an algorithm for automatically learning general and concise strategies for Nonograms in a given DSL.
- We present an implementation of this algorithm and show that the learned rules outperform documented strategies in terms of conciseness and coverage of the problem space.

The remainder of the paper is organized as follows. First, Section 2 discusses related work. Section 3 presents an overview of the system and explains the kinds of strategies we are trying to learn. Section 4 describes our DSL for Nonograms rules. Sections 5–6 discuss technical details of the system. We present an evaluation of our system that compares its output to documented strategies in Section 7, and conclude with a summary of our contribution and discussion of future work in Section 8.

2 RELATED WORK

Automated Game Analysis. Automated game analysis is a growing research area that aims to uncover designer-relevant information about games without human testing [21, 24, 31, 35], which is needed in situations where human testing is too expensive or of limited effectiveness [38]. Researchers have investigated estimating game balance [12, 38] and using constraint solving to ensure design constraints are satisfied [30]. These approaches typically reason at the level of game mechanics or available player actions. We contend that for many domains, such as logic puzzles, the mechanics do not capture the domain-specific features players use in their strategies, necessitating representations that contain such features.

General Game AI [25] is a related area in automatically understanding game strategies. However, it tackles the problem of getting a computer to play a game while we tackle the different problem of finding human-interpretable strategies for playing a game.

Prior research has also looked at analyzing the difficulty of logic puzzles. Browne presents an algorithm called *deductive search* designed to emulate the limits and process of human solvers [4]. Batenburg and Kosters estimate difficulty of Nonograms by counting the number of steps that can be solved one row/column at a time [2]. This line of work relies on general difficulty metrics, while our work models the solving process with detailed features captured by our DSL for rules.

Interpretable Learning. Finding interpretable models has become a broader topic of research because there are many domains where it is important, such as policy decisions. Researchers have looked at making, e.g., machine learning models more explainable [16, 28]. Many of these techniques focus on either (1) learning an accurate model and then trying to find an interpretable approximation (e.g., [11]), or (2) restricting the space of models to only those that are interpretable. We take the latter approach, targeting a domain not addressed by other work with a unique approach of program synthesis.

Modeling Games with Programming Languages. Game description languages are a class of formal representations of games, many of which were proposed and designed to support automated analysis. Examples include languages for turn-based competitive games [17] and adversarial board games [5, 23]. Osborn et al. [24] proposed the use of such a language for computational critics. The Video Game Description Language (VGDL) [27] was developed to support general game playing. Operational logics [19] deal with how humans understand the game, and Ceptre [18] is a language motivated by gameplay. We share goals here, but are arguing to build a new DSL for each game. All of these prior languages model the rules or representation of the game, while our language models concepts meaningful to players in order to capture strategies at a fine-grained level, which necessitates the inclusion of domain-specific constructs.

Program Synthesis. Program synthesis, the task of automatically finding programs that implement given specifications [9], is well-studied and has been used for a variety of applications. Notably, program synthesis has been used for several applications in problem-solving domains, from solution generation [10] to problem generation [1] to feedback generation [29].

One challenging feature of our program synthesis problem is its underspecification. Some methods address this challenge with an interactive loop with the user to refine specifications [9], while others rank possible programs and select a single best one [26]. Our method ranks programs using the metrics of generality and conciseness, and differs from prior work in that we are choosing a set of programs that best implement a set of specifications, rather than a single program for a single specification.

3 OVERVIEW

This section provides a high-level overview of our system for synthesizing interpretable rules for Nonograms. We review the mechanics of the game and show an example of a greedy strategy that our system can learn. Of course, one can always solve a puzzle by some combination of brute-force search and manual deduction, but human players prefer to use a collection of greedy strategies. Puzzles are designed with these strategies in mind, which take the form of condition-action rules. This section illustrates the key steps that our system takes to synthesize such condition-action rules. Sections 4–6 present the technical details of our DSL, the rule synthesis problem, and the algorithms that our system employs at each step.

3.1 Condition-Action Rules for Nonograms

Nonograms puzzles can be solved in any order: as cells are deduced and filled in, monotonic progress is made towards the final solution. In principle, deductions could be made using information from the entire $n \times m$ board. In practice, people focus on some substate of the board. One natural class of substates are the individual rows and columns of the board, which we call *lines*. Since the rules of the game are defined with respect to individual lines, they can be considered in isolation. A solving procedure that uses only lines is to loop over all lines of the board, applying deductions to each. This will reveal more information, allowing more deductions to be applied to crossing lines, until the board is filled. As many puzzle books and games can be completed by only considering (greedy) rules on lines, that is the scope on which we focus for this paper.

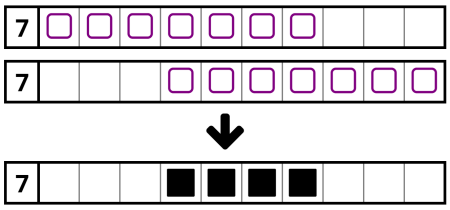
As an example of a greedy condition-action rule for Nonograms, we consider what we call the *big hint* rule (Figure 2), a documented strategy for Nonograms.[2] If a hint value is sufficiently large relative to the size of the line (Figure 2a), then, without any further information, we can fill in a portion of the middle of the row. The big hint rule can be generalized to multiple hints (Figure 2b): if there is any overlap between the furthest left and furthest right possible positions of a given hint in the row, we can fill in that overlap. Our system aims to discover sound strategies of this kind, and synthesize human-readable explanations of them that are (1) general, so they apply to a wide variety of puzzle states, and (2) concise, so they have as simple an explanation as possible.

3.2 System Overview

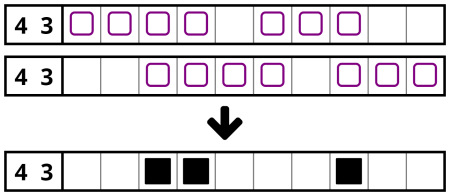
To synthesize sound, general, and concise descriptions of Nonograms strategies, our system (Figure 3) needs the following inputs:

(1) The formal rules of Nonograms to determine the soundness of learned strategies.

[2] https://en.wikipedia.org/wiki/Nonogram#Simple_boxes



(a) An example of the *big hint* rule: for any line with a single, sufficiently large hint, no matter how the hint is placed, some cells in the center will be filled.



(b) An example of the *big hint* rule for multiple hints.

Figure 2: The *big hint* rule for one (a) and many (b) hints. This is an example of the kind of sound greedy strategy for which we aim to learn interpretable descriptions.

(2) A domain-specific language (DSL) defining the concepts and features to represent these strategies.
(3) A cost function for rules to measure their conciseness.
(4) A training set of line states from which to learn rules.
(5) A testing set of line states with which to select an optimal subset of rules (that maximize state coverage).

Given these inputs, the system uses a 3-phase algorithmic pipeline to produce an optimal set of rules represented in the DSL: *specification mining*, *rule synthesis*, and *rule set optimization*. We explain each of these phases by illustrating their operation on toy inputs.

3.2.1 Specification Mining. Before synthesizing interpretable strategies, we need specifications of their input/output behavior. Our system mines these specifications from the given training states as illustrated in Figure 4. For each training state, we use the rules of Nonograms (and an SMT solver) to calculate all cells that can be filled in, producing a maximally filled target state. The resulting pair of line states—the training state and its filled target state—forms a sound *transition* in the state of the game. In our system, an individual transition forms the specification for a rule. A single transition is an underspecification of a strategy since many rules may cover that particular transition. We leave it to the rule synthesis phase to find the most general and concise rule for each mined specification.

3.2.2 Rule Synthesis. The rule synthesis phase takes as input a mined transition, the DSL for rules, and the cost function measuring rule complexity. Given these inputs, it uses standard synthesis techniques to find a program in the DSL that both covers the mined transition and is sound with respect to the rules of Nonograms. Figure 5 shows the output of the synthesis phase for the first transition in our toy example (Figure 4).

The key technical challenges this phase must solve, beyond finding sound rules, is to ensure the rules are general and concise. Generality is measured by the number of line states to which the rule is applicable, and conciseness is measured by the cost function provided by the designer. We use iterative optimization to maximize each of these. We additionally exploit the structure of the DSL for generality, which we detail in Section 6.

3.2.3 Rule Set Optimization. The synthesis phase produces a set of programs in the DSL, one for each mined transition, that represent the interpretable strategies we seek. Because the DSL captures human-relevant features of Nonograms, the concise programs are human-readable descriptions of the strategies. However, this set of rules can be unmanageably large, so the rule optimization phase prunes it to a subset of the most effective rules. In particular, given a set of rules and a set of testing states on which to measure their quality, this phase selects a subset of the given rules that best covers the states in the testing set. In our implementation, testing states are drawn from solution traces to real-world puzzles. Thus, in our toy example, the big hint rule will be selected for the optimal subset because it is widely applicable in real-world puzzles.

4 A DOMAIN-SPECIFIC LANGUAGE FOR NONOGRAMS RULES

Our system uses a domain-specific language (DSL) to represent a space of explainable condition-action rules for Nonograms. Programs in this DSL are human-readable representations of greedy strategies for solving Nonograms puzzles; they could, for instance, be mechanically translated into a written description. Compared to representations such as neural networks, designers can easily inspect DSL rules and comprehend how they work. This section presents the key features of our DSL and discusses how similar DSLs could be developed for other puzzle games.[3]

4.1 Patterns, Conditions, and Actions

In our DSL, a program representing a rule consists of three parts:

(1) A *pattern* (to which part of the state does it apply).
(2) A *condition* (when does it apply).
(3) An *action* (how does it apply).

The high-level semantics of rules are simple: for a given state, if there is a binding assignment to the pattern, and if the condition holds for those bindings, then the action may be applied to the state. We describe these constructs in more detail below.

4.1.1 Patterns. Patterns are the constructs that allow a rule to reference parts of the state, such as "the first block of filled cells." Conditions and actions can only reference state through the pattern elements. The semantics of patterns are non-deterministic and existential: a rule can apply to a state only if there is some satisfactory binding to the pattern, but it may apply to any such satisfactory binding.

Our Nonograms DSL exposes three properties of a line state through patterns, as illustrated in Figure 6. *Hints* are the integer hints specified by a puzzle instance. *Blocks* are contiguous segments of cells that are **true**. *Gaps* are contiguous segments of cells that

[3] Appendix A contains a formal description of the syntax and semantics of the DSL.

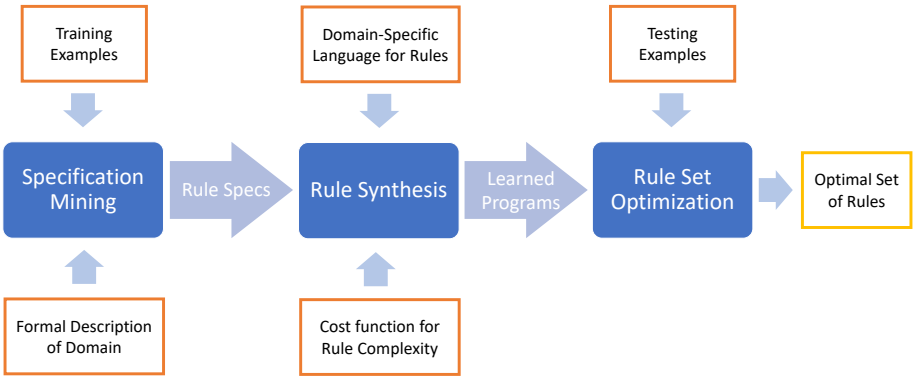


Figure 3: An overview our system's three phases, along with their inputs and outputs.

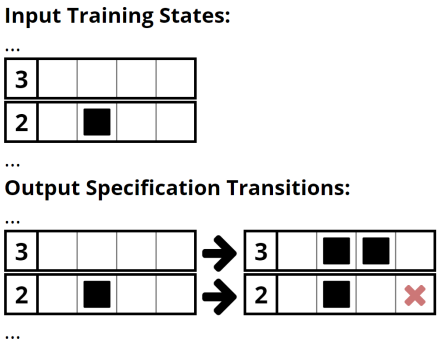


Figure 4: Example of the inputs and outputs for specification mining in our toy problem. Given a set of states, we use the rules of Nonograms to calculate deducible transitions for those states. Each transition serves as the specification for a potential rule.

```
def big_hint_rule:
    with h = singleton(hint):
    if lowest_end_cell(h) > highest_start_cell(h):
    then fill(true, highest_start_cell(h),
                    lowest_end_cell(h))
```

Figure 5: Basic version of the big hint rule. These programs are the output of the rule synthesis phase of the system. The `with`, `if`, and `then` delineate the 3 parts of a rule: the *pattern*, *condition*, and *action*. These are explained in Section 4.

are not **false** (i.e., either unknown or **true**). The lists of all hints, blocks, and gaps can be mechanically enumerated for any state.

These elements of the state can be bound using three constructs:

Arbitrary(e) binds non-deterministically to any element of type e (i.e., hint, block, or gap) that is present in the state.

Constant(e, i) binds to the i^{th} element of type e, if the state contains at least $i + 1$ elements of that type.

Singleton(e) binds to the first element of type e, if the state contains only one element of that type.

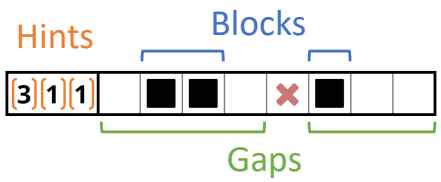


Figure 6: The three types of elements to which patterns can refer. Hints are part of the state, blocks are contiguous segments of true cells, and gaps are contiguous segments of non-false cells.

For example, the state in Figure 6 has two blocks: b_1 starts at index 1 and is length 2, and b_2 starts at index 5 and is length 1. The pattern expression `arbitrary(block)` binds to either b_1 or b_2, `constant(block, 1)` binds only to b_2, and `singleton(block)` does not bind at all because there are multiple blocks. A program may bind any number of state elements (using multiple pattern constructs).

A key property of our pattern constructs is that they form a lattice of generalization. For example, if a sound rule contains an *arbitrary* pattern, and that pattern is replaced with any *constant* pattern, the resulting rule will still be sound, but less general. Similarly, a *constant*(0) pattern can be replaced with a *singleton* pattern to obtain another less general rule. We exploit this property during synthesis as a way to generalize rules, by searching for rules with more general pattern. Figure 7 shows the result of applying this form of generalization to the big hint rule from Figure 5: the new rule uses a more general pattern and is thus applicable to more states.

```
def big_hint_rule_general:
    with h = arbitrary(hint):
    if lowest_end_cell(h) > highest_start_cell(h):
    then fill(true, highest_start_cell(h),
                    lowest_end_cell(h))
```

Figure 7: General version of the big hint rule, which uses an `arbitrary` pattern instead of a `singleton` pattern. This rule applies in strictly more situations than the one in Figure 5 and is therefore is better on the metric of generality.

4.1.2 Conditions. Conditions are boolean expressions describing the constraints necessary for sound application of a rule's action. These expressions can include basic arithmetic operators (e.g., addition and integer comparison), constructs that encode geometric properties of lines (e.g., `lowest_end_cell` in Figure 5), constructs that determine whether a particular bound element is maximal (e.g., is the bound hint the largest hint?), and constructs that determine whether a particular expression is unique for the binding of a given pattern (e.g., is this gap the only gap bigger than the first hint?). When the same condition is expressible in multiple ways, our system uses the designer-provided cost function to choose the best one. For example, Figure 8 shows another program for the basic big hint rule that uses different condition constructs than the equivalent program in Figure 5. Which rule is selected by our system depends on whether the cost function assigns lower values to geometric or arithmetic operations—a decision left to the designers using the system, enabling them to explore the ramifications of various assumptions about player behavior.

```
def big_hint_rule_arithmetic:
    with h = singleton(hint):
    if 2 * h > line_size:
    then fill(true, line_size - h, h)
```

Figure 8: Variant of the basic big hint rule (Figure 5) but using arithmetic constructs instead of geometric ones. The designer-provided cost metric for rules is used to measure their relative complexity and choose the more concise.

4.1.3 Actions. Our DSL limits a rule's actions to filling a single contiguous run of cells in a line. Actions expressions are of the form `fill(b,s,e)`, which says that the board state may be modified by filling in the cells in the range $[s, e)$ (which must both be integer expressions) with the value b (either **true** or **false**). These simple actions are sufficient to express common line-based strategies for Nonograms (see Section 7), but it would be easy to support more complex actions since our algorithms are agnostic to the choice of action semantics.

4.2 Creating DSLs for other Domains

While the detailed constructs of our DSL are domain-specific, the structure is general to other logic puzzles. Our system assumes the DSL has the basic structure of patterns, conditions, and actions, but the rest of it can be varied. Our DSL for Nonograms is one of many plausible DSLs with this structure, and similar DSLs could be crafted for games such as Sudoku.

5 PROBLEM FORMULATION

As illustrated in Section 3, our system (Figure 3) synthesizes *concise* programs in its input DSL that represent *sound* and *general* condition-action rules for Nonograms. In particular, the learned rules cover *transitions* mined from a given set of *line states*. We formalize these notions below and provide a precise statement of the rule synthesis problem solved by our system.

5.1 Line States and Transitions

We focus on lines (Definition 5.1) as the context for applying strategies. Any sound deduction the player makes on a Nonograms line takes the form of a valid transition (Definition 5.3). While our definitions of these notions are specific to Nonograms, analogous notions exists for any puzzle game (e.g., Sudoku) in which the player monotonically makes progress towards the solution. Our problem formulation assumes the domain has (partially ordered) states and transitions, but our algorithm is agnostic to the details.

Definition 5.1 (Line State). A Nonograms *line state* (also called just *line* or *state*) is an ordered sequence of hints, and an ordered sequence of cells. Hints are known positive integers. Cells can be unknown (empty) or filled with either **true** or **false**. A state is *valid* if there is an assignment of all unknown cells such that the rules of the puzzle are satisfied for the hints and cells. Unless otherwise noted, we are implicitly talking about valid states.

Definition 5.2 (Partial Ordering of States). Given any two states s and t, s is *weaker* than t ($s \preceq t$) iff s and t share the same hints, the same number of cells, and, filled cells in s are a subset of the filled cells in t. In particular, $s \preceq t$ if t is the result of filling zero or more of unknown cells in s. Being strictly weaker ($s \prec t$) is being weaker and unequal.

Definition 5.3 (Line Transition). A Nonograms *line transition* (or, simply, a *transition*) is a pair of states $\langle s, t\rangle$ where $s \preceq t$. A transition is *productive* iff $s \prec t$. A transition is *valid* iff both states are valid and the transition represents a sound deduction, meaning t necessarily follows from s and the rules of Nonograms. A transition $\langle s, t\rangle$ is *maximal* iff it is valid and for all valid transitions $\langle s, u\rangle$, u is weaker than t (i.e., $u \preceq t$). As we are only concerned with valid states and sound deductions, unless otherwise mentioned, we are implicitly talking about valid transitions.

5.2 Rules

Strategies, or rules, are defined (Definition 5.4) as the set of transitions they engender. Rules are non-deterministic because they may apply in multiple ways to the same input state, yielding multiple output states. This can be the case, for example, for programs in our DSL that contain an **arbitrary** binding. We therefore treat rules as relations (rather than functions) from states to states. Given this treatment, we define rule generality (Definition 5.5) to favor rules that apply to as many input states as possible. Finally, since we are interested in finding concise representations of these rules in the Nonograms DSL, we define rule conciseness (Definition 5.6) in terms of the cost function provided as input to our system.

Definition 5.4 (Rules). A Nonograms *rule* is a relation from states to states. A rule r is *sound* iff all pairs of states $\langle s, t\rangle \in r$ are valid transitions.

Definition 5.5 (Generality of Rules). Given a state s, we say that a rule r *covers* s if $\langle s, t\rangle \in r$ for some t with $s \prec t$. A rule r is *more general* than a rule q if r covers a superset of states covered by q.

Definition 5.6 (Conciseness of Rules). Let f be a *cost function* that takes as input a program in the Nonograms DSL and outputs a real value. Let R and Q be two programs in the DSL that represent the

rule r (i.e., R and Q are semantically equivalent and their input/output behavior is captured by the relation r). The program R is *more concise* than the program Q iff $f(R) \leq f(Q)$.

5.3 The Rule Synthesis Problem

Given the preceding definitions, we can now formally state the problem of synthesizing rules for Nonograms:

> Given a DSL and a set of states, the rule synthesis problem is to find a set of most concise programs in the DSL that cover the given states and that represent the most general sound rules with respect to those states.

6 ALGORITHMS AND IMPLEMENTATION

This section presents the technical details of our system (Figure 3) for synthesizing sound, general, and concise condition-action rules for Nonograms. We describe our algorithm for specification mining, rule synthesis, and rule set optimization, and discuss key aspects of their implementation. Section 7 shows the effectiveness of this approach to discovering explainable strategies for Nonograms.

6.1 Specification Mining

As illustrated in Figure 4, specification mining takes as input a set of line states (Definition 5.1) and produces a set of maximal transitions (Definition 5.3), one for each given state, that serve as specifications for the rule synthesis phase. In particular, for every training state s, we use an SMT solver to calculate a target state t such that $\langle s, t\rangle$ is a valid transition, and t is stronger than any state u (i.e., $u \preceq t$) for which $\langle s, u\rangle$ is a valid transition. This calculation is straightforward: for each unknown cell in s, we ask the SMT solver whether that cell is necessarily **true** or **false** according to the rules of Nonograms, and fill it (or not) accordingly. The resulting transitions, and therefore rule specifications, represent the strongest possible deductions that a player can make for the given training states.

The effectiveness of our mining process depends critically on the choice of the training set. If the training set is too small or haphazardly chosen, the resulting specifications are unlikely to lead to useful rules. We want a variety of states, so enumerating states up to a given size is a reasonable choice. But for lines of size n, there are between 2^n and 4^n possible transitions, so we choose a relatively small value (in our evaluation, a random subset of lines of size $n \leq 7$), relying on the rule synthesis phase to generalize these rules so they apply on the larger states in the testing set.

6.2 Rule Synthesis

6.2.1 Basic Synthesis Algorithm. Given a transition $\langle s, t\rangle$, we use an off-the-shelf synthesis tool [36] to search for a program in the Nonograms DSL that includes the transition $\langle s, t\rangle$ and that is sound with respect to the rules of the game. Formally, the synthesis problem is to find a program P in our DSL that encodes a sound rule R with $\langle s, t\rangle \in R$. This involves solving the **2QBF** problem $\exists_P \forall_u \varphi(u, P(u))$, where the quantifier-free formula $\varphi(u, P(u))$ encodes the rules of Nonograms and requires $\langle s, t\rangle$ to be included in P's semantics. The synthesis tool [36] solves our **2QBF** problem using a standard algorithm [32] that works by reduction to SMT.

6.2.2 Implementation of the Basic Algorithm. Most synthesis tools that work by reduction to SMT have two key limitations: (1) they can only search a finite space of programs for one that satisfies φ, and (2) they can only ensure the soundness of P on finite inputs. We tackle the first limitation through iterative deepening: our implementation asks the synthesis tool to search for programs of increasing size until one is found or a designer-specified timeout has passed. We address the second challenge by observing that practical puzzle instances are necessarily limited in size. As a result, we do not need to find rules that are sound for all line sizes: it suffices to find rules that are sound for practical line sizes. Our implementation takes this limit on line size to be 30. As a result, learned rules are guaranteed to be sound for puzzles of size 30×30 or less.

6.2.3 Synthesizing General and Concise Rules. Our basic synthesis algorithm suffices to find sound rules, but we additionally want to find general and concise rules. Generalization has two potential avenues for optimization: generalizing the patterns (to bind more states) or generalizing the conditions (to accept more bound states). Finding concise rules involves minimizing the cost of synthesized programs according to the designer-provided cost function for the Nonograms DSL. We discuss each of these optimizations in turn.

Enumerating over patterns to generalize rules. As described in Section 4.1.1, the pattern constructs of our DSL are partially ordered according to how general they are: **arbitrary** is more general than **constant** which is more general than **singleton**. We can exploit this structure to find general rules with the following method: once we find a sound rule, we attempt to find another sound rule while constraining the pattern to be strictly more general.

Our implementation performs this generalization through brute-force enumeration. For each specification, we calculate all the possible elements of a state (see Figure 6 for an example), and translate each to the most specific set of patterns possible. For the example in Figure 6, there would 7 of them: `constant(hint,0)`, `constant(hint,1)`, `constant(hint,2)`, `constant(block,0)`, `constant(block,1)`, `constant(gap,0)`, `constant(gap,1)`. We fix these and try to synthesize a rule program with that pattern set. Upon success, we enumerate over all possible ways to make the patterns one step more general (e.g., by replacing a **constant** with an **arbitrary**) and try to find rules for those. We explore the entire graph of possible patterns this way, and in doing so find the most general (with respect to the patterns) rules for each specification. There may be multiple maximally general rules; our system will output all of them, relying on the rule set optimization phase to choose the best.

In practice, useful general rules use relatively few bound elements (*big hint* uses only one, for example). We can significantly improve the performance of pattern generalization by searching for programs with fewer patterns first. Referencing our previous example, rather than finding rules with all 7 patterns, we would search for programs that use small subsets of them, in increasing order. Our implementation stops after a fixed upper bound on size but in principle could enumerate over all of them.

Iterative optimization of conditions to generalize rules. Even with a fixed pattern, the generality of a rule can change depending on the condition. We want to choose the condition that covers the maximal number of training states. As we do not have a structure of

the DSL to easily exploit, we instead rely on iterative optimization. After finding a sound rule program P_0, we attempt to synthesize a new program P with the additional constraints that

(1) for any state that P_0 covers, P must also cover it, and
(2) there exists at least one state that P covers but P_0 does not.

Looping until failure, we can be certain we have a most general rule with respect to the coverage of the condition.

This technique is greedy; we will find some arbitrary locally most general rule. But there can be many ways to generalize the condition of a rule (as suggested by our results; see Section 7.1). While our implementation produces only one locally most general rule, we could easily extend the system to produce all such rules by restarting the search after hitting a local optimum and adding constraints to cover states not covered by *any* of the previous rules.

Iterative optimization for concise rules. We use a designer-provided cost function f on the syntax of the DSL to measure the complexity of the rules. The problem of finding most concise rules is one of minimizing this cost. As with condition generalization, we do this with iterative optimization: after finding a sound rule program P_0, we attempt to synthesize a new semantically equivalent program P with the additional constraint that $f(P) < f(P_0)$. Repeating until failure will give us a globally most concise rule.

Combining generalization and cost minimization. We combine all of the above optimizations in a nested loop. For each given specification, we enumerate over all patterns and synthesize a set of programs with the most general pattern. Next, we generalize the condition of each program with the most general pattern. Finally, we make each resulting program optimally concise without sacrificing either pattern or condition generality. The resulting large set of rule programs is then pruned using rule set optimization.

6.3 Rule Set Optimization

6.3.1 Basic Rule Set Optimization Algorithm. Given a set of rule programs, the rule set optimization algorithm selects a subset of those programs that best covers the states in the designer-provided testing test. While any set of states can be used for testing, our evaluation uses a set of states drawn from solution traces of real-world puzzles. To choose a subset of rules with the best coverage of the testing set, we set up a discrete optimization problem with the following objective: select k rules (for some fixed k) that cover the greatest proportion of (maximal) transitions from the testing set. For this optimization, we measure coverage by the total number of cells filled. That is, the coverage of a test item can be partial; the objective function awards a score for each cell filled. Greedy methods suffice for this optimization.

6.3.2 Using an oracle for decomposition. When human players apply greedy strategies, they do so by considering both states, such as lines, and substates, such as parts of a line. If a player can deduce that certain hints must be constrained to certain cell ranges (as illustrated in Figure 9), then the player can focus on the identified substate (essentially, a smaller line), which might make new rules available, or at least make case-based-reasoning simpler. This form of problem decomposition is often required for applying strategies, especially on very large boards.

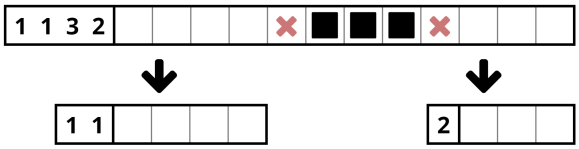


Figure 9: An example of state decomposition. Because hints are ordered, if we know where one hint lies (in this case, hint 3), then we can consider sub-lines in isolation. This allows us to apply the *big hint* rule (Figure 5) to the right sub-line.

In order to account for this player behavior when evaluating our objective function, we use an SMT solver as an oracle for decomposition.[4] That is, to measure how much of a transition can be solved with a given set of rules, we apply both the rules and the decomposition oracle to a fixed point. This allows us to measure the impact of the rules in a more realistic way than under the assumption that greedy strategies are used on their own, without decomposition.

7 EVALUATION

To evaluate our system, we compared its output to a set of documented strategies from Nonograms guides and tutorials. Unlike Sudoku, there is no comprehensive database of strategies for Nonograms, so we recovered these *control rules* from various sources: the introduction to a puzzle book [20], the tutorial of a commercial digital game [14], and the Wikipedia entry for Nonograms.[5] These sources demonstrate rules through examples and natural language, so, to encode them in our DSL, some amount of interpretation is necessary. In particular, while these sources often explain the reasoning behind a strategy, the strategy is demonstrated on a simple case, letting the reader infer the general version. We encoded the most general (when possible, multiple) variants of the demonstrated rules in our DSL, for a total of 14 control rules.

We evaluated our system by asking the following questions:

(1) Can the system recover known strategies by learning rules in the control set?
(2) How does the learned set compare to the control set when measuring coverage of the testing data?

Training Data. For this evaluation, we trained the system using a random subset of maximal transitions of lines up to length 7. There were 295 such states. Note that, excepting tutorial puzzles, no commercial puzzles are on boards this small, so none of the testing examples are this small.

Testing Data. Our testing data is drawn from commercial Nonograms puzzle books and digital games *Original O'Ekaki* [22], *The Essential Book of Hanjie and How to Solve It* [20], and the *Pircoss e* series [13–15]. We randomly selected 17 puzzles from these sources. The lines range in size from 10 to 30. All puzzles are solvable by considering one line at a time. To get test data from these boards, we created solution traces with random rollouts by selecting lines,

[4] The *Picross* series of videogames for the Nintendo DS and 3DS actually provide a limited version of this oracle to the player. For each partially-filled line, the game tells the player which (if any) of the hints are satisfied. The reasoning is based only on the partial state, not the solution; it uses some deductive/search-based procedure.
[5] https://en.wikipedia.org/wiki/Nonogram#Solution_techniques

using our oracle to fill the maximum possible cells, and repeating until the puzzle was solved. We took each intermediate state from these solution traces as testing states. This resulted in 2805 testing states.

Learned Rules. From our training examples, the first two phases of the system (specification mining and rule synthesis) learned 1394 semantically distinct rules as determined by behavior on the test set. Two learned rules are shown in Figures 10 and 11. We measure the quality of the learned rules by comparing them to the control rule set, both on whether the learned set includes the control rules and how well the learned rules perform on the testing data.

7.1 Can the system automatically recover the control rules?

Our system recovered 9 of the 14 control rules, either exactly, or as a more general rule. Figure 10 shows an example of a rule from the control set for which our system learned a syntactically identical rule. While the training set included example states for all control rules, our greedy generalization algorithm choose a different generalization than the one represented by the missed control rules. As discussed in Section 6.2.3, we could extend the system to explore multiple generalizations. Given sufficient training time, such a system would find these particular rules as well. In some cases where our system did not match a control rule, it did find a qualitatively similar rule that covered many of the same states (as in Figure 11).

```
# crossing out the cell next to a satisfied hint,
# which can be determined because it's (one of)
# the biggest hints.
def punctuate_maximum:
    # for any arbitrary hint and block
    with
        h = arbitrary(hint) and
        b = arbitrary(block):
    # if the hint is maximal,
    # the block and hint have the same size,
    # and the block is strictly right of the left edge,
    if maximal(h) and size(b) = size(h) and start(b) > 0:
    # then cross out the cell to the left of the block
    then fill(false, start(b) - 1, start(b))
```

Figure 10: An example of a control rule that our system recovers exactly, annotated with comments. This is a top-10 rule as determined by the rule set optimization.

7.2 How does an optimal subset of rules compare on coverage?

In order to quantitatively compare the coverage of our learned set to the control set, we measured the proportion of the maximal transitions of the testing examples that each rule set covered. As described in Section 6.3, we measure this by the proportion of transitions covered; for a set of rules $\mathcal{R}$, the coverage $C(\mathcal{R})$ is the total number of cells over all testing examples covered by applying rules in $\mathcal{R}$ and the decomposition oracle to a fixed point.

```
# crossing out the left side of the line if a block
# is more than hint-value distance from the edge.
def mercury_variant:
    # for singleton hint and arbitrary block
    with
        h = singleton(hint) and
        b = arbitrary(block):
    # if the right side of the block is
    # greater than the value of the hint
    if start(b) + size(b) > size(h):
    # then cross out cells from 0 up through the
    # one that is hint-value many cells away from
    # the right edge of the block.
    then fill(false, 0, start(b) + size(b) - size(h))
```

Figure 11: An example top-10 rule learned by our system that is *not* in the control set, annotated with comments. This rule is similar to what we call the *mercury* control rule, which is not recovered exactly. But the learned rule covers a large portion of the same states. While slightly less general, it is significantly more concise than the control rule, using one fewer pattern, one fewer condition, and less complex arithmetic. The learned rule is also a reasonable interpretation of the description on which the control rule is based.[6]

Coverage of learned rules. First, we compared the entire rule sets. On our test examples, the 14 control rules $\mathcal{R}_0$ have a coverage $C(\mathcal{R}_0)$ of 4652. Our trained rule set $\mathcal{R}_t$ has a coverage $C(\mathcal{R}_t)$ of 7558. These sets are incomparable; the control rules cover some items that the learned do not and vice-versa. Looking at the union of the two sets, they have a total coverage $C(\mathcal{R}_t \cup \mathcal{R}_0)$ of 7673. That is, the learned set alone covers over 98% of the transitions covered by the learned and control sets together. This means that, even though we do not recover the control rules exactly, the learned rules cover nearly all test cases covered by the missed control rules.

Coverage of a limited set of rules. We would expect that the very small control set would have less coverage than the large set of learned rules. For a more equitable comparison, we measure the top-10 rules from each set, using the rule set optimization phase of our system. Choosing the top 10 rules, the top 10 control rules have a coverage of 4652 (unchanged from the full 14), and the top 10 learned rules have a coverage of 6039. The learned rules, when limited to the same number as the control rules, still outperform the control on the testing examples. Figure 11 shows an example of a learned rule in the top-10 set that was not in the control set. Comparing the complexity of these rules with the cost function, the top-10 control rules have a mean cost of 31.7 while the top-10 from the learned set have a mean cost of 33.7. Though our learning algorithm minimizes individual rule cost, the optimization greedily maximizes coverage while ignoring cost.

These results suggest that our system can both recover rules for known strategies and cover more states from real-world puzzles.

8 CONCLUSION

This paper presented a system for automated synthesis of interpretable strategies for the puzzle game Nonograms. Our system

[6] https://en.wikipedia.org/wiki/Nonogram#Mercury

takes as input the rules of Nonograms, a domain-specific language (DSL) for expressing strategies, a set of training examples, and a cost function for measuring rule complexity. Given these inputs, it learns general, concise programs in the DSL that represent effective strategies for making deductions on Nonograms puzzles. The DSL defines the domain-specific constructs and features of Nonograms, ensuring that our learned strategies are human-interpretable. Since the DSL is taken as input, the designer can use it to encode domain-specific considerations and bias. When compared with documented strategies recovered from guides and tutorials, the rules our system learns recover many existing strategies exactly and outperform them when measured by coverage on states from real-world puzzles.

This work focused on automatic synthesis of human-interpretable strategies. But the eventual goal is to find strategies that humans are likely to use. Avenues of future work thus include using player solution traces for cost estimation. For example, rather than using a designer-provided cost function to estimate rule complexity, we can attempt to learn a cost function from play traces.

Our work enables a range of applications, such as game-design feedback, difficulty estimation, and puzzle and progression generation. For example, previous puzzle game generation research relied on a designer-authored space of concepts and solution features to generate a progression of puzzles [6]. Tools like the one presented in this paper can serve as input for such systems.

While we applied our system to Nonograms, we expect it to be applicable to other puzzle games as well. The system assumes that the input DSL has the basic structure of patterns, conditions, and actions, but is agnostic to the detailed constructs. The presented system is designed for logic puzzles of this structure, but we believe that program synthesis can be used to learn human-interpretable strategies in a wider range of games and problem-solving domains.

A FORMAL DSL DEFINITION

This appendix provides formal definitions for the syntax (Figure 12) and semantics (Figure 13) of the Nonograms DSL (Section 4) used for rule learning. The semantics use the following definitions in addition to those from Section 5.

Definition A.1 (Block Element). A *block* of line s is a maximally sized, contiguous sequence of cells from s where every cell is filled with **true**. Blocks are defined by a pair $\langle i, n \rangle$, where i is the index of the starting cell of the block and n is the number of cells in the block. The blocks of line s are the sequence of all blocks, ordered ascending by their starting cell index.

Definition A.2 (Gap Element). A *gap* of line s is a maximally sized, contiguous sequence of cells from s where every cell is *not* filled with **false**. Gaps are defined by a pair $\langle i, n \rangle$, where i is the index of the starting cell of the gap and n is the number of cells in the gap. The gaps of line s are the sequence of all gaps, ordered ascending by their starting cell index.

Cost Function. The cost function used to measure rule conciseness in our evaluation is a monotonic linear function defined over the program syntax. That is, each syntactic element (e.g., `maximal`) is given a fixed real value, and the cost of a given piece of syntax is the value of the particular element plus the sum of the costs of all subexpressions.

rule	R	::=	**with** P: **if** b: **then** A
patterns	P	::=	d [**and** d]*
pattern declaration	d	::=	x = p
pattern expression	p	::=	**singleton**(t)
		|	**constant**(t, k)
		|	**arbitrary**(t)
pattern type	t	::=	**hint** | **block** | **gap**
action	A	::=	**fill**(B, e, e)
boolean expression	b	::=	**true**
		|	b **and** b
		|	e o_b e
		|	x **is unique where** b
		|	**maximal**(x)
		|	**minimal**(x)
integer expression	e	::=	(e)
		|	k
		|	**line_size**
		|	e o_e e
		|	**start**(x)
		|	**size**(x)
		|	lowest_end_cell(x)
		|	highest_start_cell(x)
arithmetic operator	o_e	::=	+ | -
comparison operator	o_b	::=	= | >= | >
identifier	x	::=	identifier
integer	k	::=	integer literal
boolean	B	::=	**true** | **false**

Figure 12: Syntax for the Nonograms DSL. The notation [*form*]* means zero or more repetitions of the given form.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. The work is supported by the National Science Foundation under Grant No. 1639576 and 1546510 and by Oak Foundation under Grant No. 16-644.

REFERENCES

[1] Erik Andersen, Sumit Gulwani, and Zoran Popović. 2013. A Trace-based Framework for Analyzing and Synthesizing Educational Progressions. In *CHI*.
[2] K Joost Batenburg and Walter A Kosters. 2012. On the difficulty of Nonograms. *ICGA Journal* 35, 4 (2012), 195–205.
[3] John D Bransford, Ann L Brown, Rodney R Cocking, and others. 2000. *How people learn*. Washington, DC: National Academy Press.
[4] Cameron Browne. 2013. Deductive search for logic puzzles. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE.
[5] Cameron Browne and Frederic Maire. 2010. Evolutionary Game Design. *IEEE Transactions on Computational Intelligence and AI in Games* 2 (2010), 1–16.
[6] Eric Butler, Erik Andersen, Adam M. Smith, Sumit Gulwani, and Zoran Popović. 2015. Automatic Game Progression Design Through Analysis of Solution Features. In *Proc. of the 33rd ACM Conf. on Human Factors in Computing Systems (CHI '15)*.
[7] Eric Butler, Emina Torlak, and Zoran Popović. 2016. A Framework for Parameterized Design of Rule Systems Applied to Algebra. In *Intelligent Tutoring Systems*. Springer.
[8] Mary L Gick. 1986. Problem-solving strategies. *Educational psychologist* 21, 1-2 (1986), 99–120.
[9] Sumit Gulwani. 2010. Dimensions in Program Synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative*

$[\![\texttt{with } P\texttt{: if } b\texttt{: then } A]\!]s = \textbf{if } \sigma \neq \bot \wedge [\![b]\!](s,\sigma) \textbf{ then } [\![A]\!](s,\sigma) \textbf{ else } \bot \quad \textbf{where } \sigma = [\![P]\!](s,\emptyset)$

$[\![d_1 \texttt{ and } d_2]\!](s,\sigma) = [\![d_1]\!](s, [\![d_2]\!](s,\sigma))$
$[\![x \texttt{ = } p]\!](s,\sigma) = \textbf{if } v \neq \bot \wedge \sigma \neq \bot \textbf{ then } \sigma \cup x \mapsto v \textbf{ else } \bot \quad \textbf{where } v = [\![p]\!]s$
$[\![\texttt{singleton(}t\texttt{)}]\!]s = \mathit{singleton}(\mathit{elements}(s,t))$
$[\![\texttt{constant(}t\texttt{, } k\texttt{)}]\!]s = \mathit{listref}(\mathit{elements}(s,t),k)$
$[\![\texttt{arbitrary(}t\texttt{)}]\!]s = \mathit{arbitrary}(\mathit{elements}(s,t))$

$[\![\texttt{fill(}B\texttt{,}e_1\texttt{,}e_2\texttt{)}]\!](s,\sigma) = \mathit{fill}([\![B]\!](s,\sigma), [\![e_1]\!](s,\sigma), [\![e_2]\!](s,\sigma))$

$[\![\texttt{true}]\!](s,\sigma) = \text{true}$
$[\![\texttt{false}]\!](s,\sigma) = \text{false}$
$[\![b_1 \texttt{ and } b_2]\!](s,\sigma) = [\![b_1]\!](s,\sigma) \wedge [\![b_2]\!](s,\sigma)$
$[\![e_1 \texttt{ = } e_2]\!](s,\sigma) = [\![e_1]\!](s,\sigma) = [\![e_2]\!](s,\sigma)$
$[\![e_1 \texttt{ >= } e_2]\!](s,\sigma) = [\![e_1]\!](s,\sigma) \geq [\![e_2]\!](s,\sigma)$
$[\![e_1 \texttt{ > } e_2]\!](s,\sigma) = [\![e_1]\!](s,\sigma) > [\![e_2]\!](s,\sigma)$
$[\![x \texttt{ is unique where } b]\!](s,\sigma) = [\![b]\!](s,\sigma) \wedge \forall_{v \in \mathit{elements}(s,\mathit{type}(u)) \setminus u} \neg [\![b]\!](s, \sigma \cup x \mapsto v) \quad \textbf{where } u = \sigma[x]$
$[\![\texttt{maximal(}x\texttt{)}]\!](s,\sigma) = \forall_{v \in \mathit{elements}(s,\mathit{type}(u))} \mathit{size}(u) \geq \mathit{size}(v) \quad \textbf{where } u = \sigma[x]$
$[\![\texttt{minimal(}x\texttt{)}]\!](s,\sigma) = \forall_{v \in \mathit{elements}(s,\mathit{type}(u))} \mathit{size}(u) \leq \mathit{size}(v) \quad \textbf{where } u = \sigma[x]$

$[\![\texttt{(}e\texttt{)}]\!](s,\sigma) = [\![e]\!](s,\sigma)$
$[\![k]\!](s,\sigma) = k$
$[\![\texttt{line\_size}]\!](s,\sigma) = \text{number of cells in line } s$
$[\![e_1 \texttt{ + } e_2]\!](s,\sigma) = [\![e_1]\!](s,\sigma) + [\![e_2]\!](s,\sigma)$
$[\![e_1 \texttt{ - } e_2]\!](s,\sigma) = [\![e_1]\!](s,\sigma) - [\![e_2]\!](s,\sigma)$
$[\![\texttt{start(}x\texttt{)}]\!](s,\sigma) = \mathit{start}(\sigma[x])$
$[\![\texttt{size(}x\texttt{)}]\!](s,\sigma) = \mathit{size}(\sigma[x])$
$[\![\texttt{lowest\_end\_cell(}x\texttt{)}]\!](s,\sigma) = \mathit{lowestend}(s,\sigma[x])$
$[\![\texttt{highest\_start\_cell(}x\texttt{)}]\!](s,\sigma) = \mathit{higheststart}(s,\sigma[x])$

$\mathit{elements}(s, \textbf{hint}) = \text{The ordered sequence of hints of } s \text{ (Definition 5.1)}$
$\mathit{elements}(s, \textbf{block}) = \text{The ordrered sequence of blocks of } s \text{ (Definition A.1)}$
$\mathit{elements}(s, \textbf{gap}) = \text{The ordrered sequence of gaps of } s \text{ (Definition A.2)}$

$\mathit{type}(u) = \text{whether } u \text{ is a hint, block or gap}$
$\mathit{start}(u) = i \textbf{ if } u \text{ is the } i^{\text{th}} \text{ hint}$
$\quad i \textbf{ if } u \text{ is a block } (i,n)$
$\quad i \textbf{ if } u \text{ is a gap } (i,n)$
$\mathit{size}(u) = h \textbf{ if } u \text{ is a hint } h$
$\quad n \textbf{ if } u \text{ is a block } (i,n)$
$\quad n \textbf{ if } u \text{ is a gap } (i,n)$

$\mathit{singleton}(L) = \textbf{if } \mathit{length}(L) = 1 \textbf{ then } L[0] \textbf{ else } \bot$
$\mathit{listref}(L,i) = \textbf{if } \mathit{length}(L) > i \geq 0 \textbf{ then } L[i] \textbf{ else } \bot$
$\mathit{arbitrary}(L) = \textbf{if } \mathit{length}(L) > 0 \textbf{ then a non-deterministically chosen element from } L \textbf{ else } \bot$

$\mathit{lowestend}(s,u) = i + \sum_{j=0}^{i} h_j \textbf{ if } [h_0,\ldots,h_{k-1}] \text{ are the hints of } s \text{ and } u \text{ is } h_i$
$\quad 0 \textbf{ otherwise}$
$\mathit{higheststart}(s,u) = N - k + i + 1 - \sum_{j=i}^{k-1} h_j \textbf{ if } [h_0,\ldots,h_{k-1}] \text{ are the hints of } s, u \text{ is } h_i, \text{ and } N \text{ is the number of cells in } s$
$\quad 0 \textbf{ otherwise}$

Figure 13: Semantics for the Nonograms DSL (Figure 12). The notation s is the line state (Definition 5.1), and σ is a map from identifiers to values (i.e., the bindings of the pattern).

Programming (PPDP '10). ACM.
[10] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. 2011. Synthesizing Geometry Constructions. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM.
[11] Lisa Anne Hendricks, Zeynep Akata, Marcus Rohrbach, Jeff Donahue, Bernt Schiele, and Trevor Darrell. 2016. Generating visual explanations. In *European Conference on Computer Vision*. Springer, 3–19.
[12] Alexander Jaffe, Alex Miller, Erik Andersen, Yun-En Liu, Anna Karlin, and Zoran Popović. 2012. Evaluating Competitive Game Balance with Restricted Play. In *AIIDE*.
[13] Jupiter. 2015. Picross e6. (2015).
[14] Jupiter. 2015. Pokémon Picross. (2015).
[15] Jupiter. 2016. Picross e7. (2016).
[16] Himabindu Lakkaraju, Stephen H. Bach, and Jure Leskovec. 2016. Interpretable Decision Sets: A Joint Framework for Description and Prediction. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, New York, NY, USA.
[17] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. 2008. *General game playing: Game description language specification*. Technical Report. Stanford University.
[18] Chris Martens. 2015. Ceptre: A language for modeling generative interactive systems. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*.
[19] Michael Mateas and Noah Wardrip-Fruin. 2009. Defining operational logics, In DiGRA. *Digital Games Research Association (DiGRA)* (2009).
[20] Gareth Moore. 2006. *The Essential Book of Hanjie and How to Solve It*.
[21] Mark J Nelson. 2011. Game Metrics Without Players: Strategies for Understanding Game Artifacts.. In *Artificial Intelligence in the Game Design Process*.
[22] Tetsuya Nishio. 2008. *Original O'Ekaki: Intelligent Designs from Its Creator*.
[23] John Orwant. 2000. EGGG: Automated programming for game generation. *IBM Systems Journal* 39, 3.4 (2000), 782–794.
[24] Joseph Carter Osborn, April Grow, and Michael Mateas. 2013. Modular Computational Critics for Games.. In *AIIDE*.
[25] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon M Lucas. 2016. General video game ai: Competition, challenges and opportunities. In *Thirtieth AAAI Conference on Artificial Intelligence*.
[26] Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN Inter. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*. ACM.
[27] Tom Schaul. 2013. A Video Game Description Language for Model-based or Interactive Learning. In *IEEE Conference on Computational Intelligence in Games*.
[28] Zhangzhang Si and Song-Chun Zhu. 2013. Learning and-or templates for object recognition and detection. *IEEE transactions on pattern analysis and machine intelligence* 35, 9 (2013), 2189–2205.
[29] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA.
[30] Adam M Smith and Michael Mateas. 2011. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 187–200.
[31] Adam M Smith, Mark J Nelson, and Michael Mateas. 2010. Ludocore: A logical game engine for modeling videogames. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*. IEEE, 91–98.
[32] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th Inter. Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 12. DOI: http://dx.doi.org/10.1145/1168857.1168907
[33] Andrew C Stuart. 2007. *The Logic of Sudoku*. Michael Mepham Publishing.
[34] Andrew C Stuart. 2012. Sudoku Creation and Grading. (January 2012). http://www.sudokuwiki.org/Sudoku_Creation_and_Grading.pdf [Online, accessed 8 Mar 2017].
[35] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. 2011. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 3 (2011), 172–186.
[36] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 12.
[37] Nobuhisa Ueda and Tadaaki Nagao. 1996. NP-completeness results for NONOGRAM via parsimonious reductions. *Technical Report* (1996).
[38] Alexander Zook, Brent Harrison, and Mark O Riedl. 2015. Monte-carlo tree search for simulation-based strategy analysis. In *Proceedings of the 10th Conference on the Foundations of Digital Games*.