ECAI 2023
K. Gal et al. (Eds.)
© 2023 The Authors.
This article is published online with Open Access by IOS Press and distributed under the terms
of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/FAIA230502

Synthesis of Procedural Models for Deterministic Transition Systems

Javier Segovia-Aguas[a;*], **Jonathan Ferrer-Mestres**[b] **and Sergio Jiménez**[c]

[a]Universitat Pompeu Fabra
[b]CSIRO
[c]Universitat Politècnica de València

Abstract. This paper introduces a general approach for synthesizing procedural models of the *state-transitions* of a given *discrete system*. The approach is general in that it accepts different *target languages* for modeling the state-transitions of a discrete system; different model acquisition tasks with different target languages, such as the synthesis of STRIPS action models, or the update rule of a *cellular automaton*, fit as particular instances of our general approach. We follow an inductive approach to synthesis meaning that a set of examples of state-transitions, represented as *(pre-state, action, post-state)* tuples, are given as input. The goal is to synthesize a structured program that, when executed on a given pre-state, outputs its associated post-state. Our synthesis method implements a combinatorial search in the space of well-structured terminating programs that can be built using a *Random-Access Machine* (RAM), with a minimalist instruction set, and a finite amount of memory. The combinatorial search is guided with functions that asses the complexity of the candidate programs, as well as their fitness to the given input set of examples.

1 Introduction

Transition systems make it possible to describe the behavior of discrete systems and white-box models of such systems are used in AI with different purposes; to name a few, the building of *simulators* [19, 18], *causal inference* [41], *automated planning* [25, 22], *model-checking* [13], *system diagnosis* [43], *epistemic reasoning* [16], or *game-playing* [23]. For many domains hand-coding those models is an arduous task that faces *the knowledge acquisition bottleneck*, so inducing them from examples becomes an appealing approach. *Neural networks* (NNs) have proved to be effective learning transition models [47]. Unfortunately NNs represent knowledge as a huge number of coupled parameters, where it becomes challenging to identify the piece of knowledge responsible for modeling a particular feature of the examples, or to understand whether that knowledge will be useful at unseen examples.

In this paper we pursue a different research direction and focus on the synthesis of *white-box* models of state transitions, that are represented as *structured programs*. On the one hand programs of different kinds are proposed as an alternative to traditional action description languages [46, 31, 7, 10]. On the other hand, the flexibility of programming allows us to address model acquisition tasks with different *target languages*, without modifying the underlying method; methods for acquiring white-box models usually assume a particular

* Corresponding Author. Email: javier.segovia@upf.edu.

Figure 1: Example of a state-transition in a five-pancake instance of the *pancake sorting* domain.

target language, whose syntax and semantics is known, and that it is exploited to confine the hypothesis space of the possible models [21]. A prominent example are systems for modeling *planning actions* that take STRIPS as their *target language* [56, 14, 34, 1, 5, 30, 45, 35], and that struggle when modeling systems whose state transitions do not fit the assumptions of the target language.

We illustrate the previous issue in the *pancake sorting* domain [12], where a disordered stack of pancakes must be sorted, in order of size, using a spatula; the spatula can be inserted at any point in the stack to *flip* all the pancakes above. Figure 1 shows a state-transition of a *pancake sorting* instance. STRIPS model learners struggle at this domain because state transitions cannot be modeled by a single STRIPS action schema; STRIPS assumes that the number of state variables checked and updated at a state transition is bounded (and small). However in *pancake sorting*, the number of pancakes to flip is unbound, it depends on the number of world objects. Further, a compact procedural model requires *latent variables* that are missing in the state representation, since they do not describe observed properties/relations of the world objects. Figure 2 shows the `flip` procedure, a compact model of the state transitions of the *pancake sorting* domain that leverages two *latent variables*; the `flip` procedure models any state-transition, no matter the number of pancakes.

The paper introduces a general approach for synthesising white-box models of the state transitions of a given discrete system, represented as structured programs. This synthesis problem is challenging, since the set of state transitions of a given discrete system may be infinite e.g. in domains like *pancake sorting* where the number of world objects is unbound. Our approach is general since it accepts different *target languages*; relevant model acquisition tasks with different target languages, such as modeling STRIPS actions or the update rule of a *cellular automaton* [55], fit as particular instances of our general approach. Our synthesis method implements a combinatorial search in the space of well-structured terminating programs that can be built using a *Random-Access Machine* (RAM),

```
State flip(State pre_state, int z1, int z2){
    State post_state=pre_state;
    for(z1=0; z1<|Ω|; z1++){
        if (z1 < z2){
           post_state(z1) = pre_state(z2);
           post_state(z2) = pre_state(z1);
           z2 = z2 - 1;
        }
    }
    return post_state;
}
```

Figure 2: Structured program that models state-transitions in the *pancake sorting* domain, no matter the number of pancakes $|\Omega|$, and that leverages two *latent variables* $\{z_1, z_2\}$. States are represented as tuples (whose length is the number of pancakes), that store the size of the pancake at the corresponding position in the stack.

with a minimalist instruction set, and a finite amount of memory. This method is able to synthesize compact white-box transition models, even for domains that require *latent variables*. In addition our method can exploit a *target language*, when available, to confine the solution space, without modifying the synthesis algorithm.

2 Preliminaries

2.1 Deterministic Transition Systems

The notion of *transition system* is widely used in AI to model the behavior of discrete systems [6]. A transition system can be graphically represented as a directed graph and hence formalized as a pair $(S, \rightarrow)$, where S is a set of states, and $\rightarrow$ denotes a relation of state transitions $S \times S$. Transition systems differ from *finite automata* since the sets of states and transitions are not necessarily finite. Further a *transition system* does not necessary define a *start/initial* state or a subset of *final/goal* states.

Transitions between states may be labeled[1], and the same label may appear on more than one transition; a prominent example is the transition system that corresponds to a *classical planning problem* [25, 22], where state transitions are labeled with *actions* s.t. between two different states $s, s' \in S$, there exists a transition $(s \xrightarrow{a} s')$ iff the execution of action a in state s produces the state s'. Given a state s and an action label a, if there exists only a single tuple (s, a, s') in $\rightarrow$ then the transition is said to be *deterministic*. In this paper we focus on *deterministic* transitions systems, i.e. transition systems such that all their transitions are deterministic.

2.2 The Random Access Machine

The *Random-Access Machine* (RAM) is an abstract computation machine, in the class of the *register machines*, that is polynomially equivalent to a *Turing machine* [9]. The RAM enhances a multiple-register *counter machine* [36] with indirect memory addressing; indirect memory addressing is useful for defining RAM programs that access an unbounded number of registers, no matter how many there are. A *register* in a RAM machine is then a memory location with both an *address* i.e. a unique identifier that works as a natural number (that we denote as r), and a *content* i.e. a single natural number (that we denote as $[r]$).

A *RAM program* Π is a finite sequence of n instructions, where each program instruction $\Pi[i]$, is associated with a *program line* $0 \leq i < n$. The execution of a RAM program starts at its first program instruction $\Pi[0]$. The execution of program instruction $\Pi[i]$ updates the RAM *registers* and the *current program line*. Diverse *base instructions sets*, that are Turing complete, can be defined. We focus on the three *base sets* of RAM instructions:

- Base1. $\{\texttt{inc}(r), \texttt{dec}(r), \texttt{jmpz}(r, i), \texttt{halt}() \mid r \in R\}$. Respectively, these instructions *increment/decrement* a register by one, jump to program line $0 \leq i < n$ if the content of a register r is zero (i.e. if $[r] == 0$), or end the program execution.
- Base2. $\{\texttt{inc}(r_1), \texttt{clear}(r_1), \texttt{jmpz}(r_1, r_2, i), \texttt{halt}() \mid r_1, r_2 \in R\}$. In this set the value of a register cannot be decremented but instead, it can be set to zero with a *clear* instruction. In addition, *jump instructions* go to program line $0 \leq i < n$ if the content of two given registers is the same (i.e. if $[r_1] == [r_2]$).
- Base3. $\{\texttt{inc}(r_1), \texttt{set}(r_1, r_2), \texttt{jmpz}(r_1, r_2, i), \texttt{halt}() \mid r_1, r_2 \in R\}$. This set comprises no instruction to decrement, or clear, a register but instead, it includes an instruction to *set* a register to the value of another register.

The three *base sets* are equivalent [9]; one can build the instructions of one base set with instructions of another base set. Further, *the expansive instruction set* (that contains the instructions of *Base* 1,2 and 3) does not increase the expressiveness of the individual *Base sets*, since each of them already is Turing complete. The choice of the set of RAM instructions depends on the convenience of the programmer for the problem being addressed.

3 Transition Systems as RAM Programs

This section formalizes our representation for the state-transitions of a given deterministic discrete system.

3.1 The RAM model

WLOG we assume that the states of a transition system are factored; given a set of world objects Ω, a *state* is factored into a finite set of variables X s.t. each variable $x \in X$ either represents a *property* of a world object, or a *relation* over k world objects. Formally $x = f(o_1, \ldots, o_k)$, where f is a k-ary function in $\mathbb{N}$, and $\{o_i\}_1^k$ are objects in Ω. We compactly model the set of state *transitions* of a given discrete system as RAM programs $\Pi(s) = s'$ that map a given *pre-state* into its corresponding *post-state*. Next we formalize our particular RAM model for the compact representation of the state-transitions of a given discrete system.

Our RAM model partitions **the set of RAM registers** into four disjoint subsets, $R = \{R_X, R_{X'}, R_Y, R_Z\}$:

- R_X and R'_X, that are the **pre-state** and **post-state registers**. Our RAM model contains, for each state variable $x \in X$, two registers $r_x \in \mathcal{R}_X$ and $r'_x \in \mathcal{R}_{X'}$ representing the value of that variable at the *pre-state* and the *post-state*.
- $R_Y = \{CF, ZF\}$, the **FLAGS registers**. Our RAM model includes two dedicated Boolean registers, the *zero flag* (ZF) and the *carry flag* (CF), storing the outcome of *three-way comparisons* [11] between two registers. FLAGS registers allow us to keep the RAM instruction set compact, reducing the set of *jump* instructions; the four joint values of $\{ZF, CF\}$ can model a large space of state features, including $= 0, \neq 0, < 0, > 0, \leq 0, \geq 0$ as well as relations $=, \neq, <, >, \leq, \geq$ on pairs of registers.
- R_Z, the **latent registers** that play the role of auxiliary variables. *Latent registers* allow to writing programs that implement sequences of state updates of unbounded size (a key property for

[1] When the set of labels is a singleton, the transition system is essentially *unlabeled*, so the simpler definition that omits labels applies.

computing compact transition models that apply no matter the actual number of world objects).

Our RAM model has then $2|X| + 2 + |R_Z|$ registers, where $|X| = |R_X| = |R'_X|$ is the number of state variables.

The **instruction set** of our RAM model is $\{\texttt{inc}(r), \texttt{dec}(r), \texttt{test}(r), \texttt{set}(r, 0|1), \texttt{set}(r_1, r_2), \texttt{cmp}(r_1, r_2), \texttt{jmp}(\neg\texttt{ZF}, \neg\texttt{CF}, \texttt{i}), \texttt{jmp}(\neg\texttt{ZF}, \texttt{CF}, \texttt{i}), \texttt{jmp}(\texttt{ZF}, \neg\texttt{CF}, \texttt{i}), \texttt{jmp}(\texttt{ZF}, \texttt{CF}, \texttt{i}), \texttt{halt}()\}$. Respectively, these RAM instructions *increment/decrement* by one a register, *test* whether the value of a register is zero (or greater than zero), *set* the value of a register to `zero`, `one`, or to another register, *compare* two registers (or their content), jump to program line $0 \leq i < n$ according to the joint value of the FLAGS registers, or end the program. To keep the instruction set as compact as possible, and hence the space of candidate RAM programs, certain RAM instructions are only applicable to a subset of the RAM registers. In more detail, *pre-state registers* R_X are *read-only*, so $test$ and cmp instructions, are the only RAM instructions that apply to them. Likewise the *post-state registers* R'_X are *write-only*, so test and comparisons cannot be defined over them. FLAGS registers are dedicated to store the outcome of *three-way comparisons*, so only jump instructions can be applied to them. The value of FLAGS registers is given by the outcome of RAM instructions; each RAM instruction updates the FLAGS according to the result of the corresponding RAM instruction (which is denoted here by res):

$$\begin{aligned} inc(r) &\implies res := r + 1, \\ dec(r) &\implies res := r - 1, \\ test(r) &\implies res := r, \\ set(r_1, r_2) &\implies res := r_2, \\ cmp(r_1, r_2) &\implies res := r_1 - r_2, \\ ZF &:= (res == 0), \\ CF &:= (res > 0). \end{aligned}$$

3.2 The space of RAM programs

We consider RAM programs $\Pi : R \rightarrow R_{X'}$, that map a given *pre-state* into its corresponding *post-state*, with the assistance of the FLAGS and the *latent* registers. In this particular kind of RAM program, registers $R_{X'}$ are initialized with the value of their corresponding *pre-state* register, while FLAGS and *latent* registers are initialized to zero. We restrict ourselves to $\Pi : R \rightarrow R_{X'}$ RAM programs that are both *well-structured* and *terminating*:

- **Well-structured**. We only consider RAM programs whose jump instructions do not interleave, since this particular kind of program is more intelligible. Formally, given two program instructions $\Pi[i] = \mathsf{jump}(*, i')$ and $\Pi[j] = \mathsf{jump}(*, j')$ s.t. $i < j$, it cannot hold that $min(i, i') < min(j, j') < max(i, i') < max(j, j')$.
- **Terminating**. WLOG we restrict ourselves to RAM programs that are by definition terminating. In more detail, RAM programs where any loop caused by a jump instruction is implementing a `for loop` that iterates over the set of world objects. Formally, we only consider RAM programs where any *jump instruction* that corresponds to a loop (i.e. such that $\Pi[i] = \mathsf{jump}(*, i')$ and $i' < i$) iterates over the set of world-objects i.e. loops of the form $\Pi[i] = set(r, 0), \Pi[i + k] = inc(r), \Pi[i + k + 1] = jmpz(ZF, CF, i)$.

Restricting to *well-structured* and *terminating* RAM programs, allows us to keep the space of candidate RAM programs compact. It also improves the intelligibility of candidate RAM programs, since it allows to replace $test$, cmp and jmp instructions by their equivalent `If conditionals` and `For loops` control flow constructs from *structured programming*; in a *well-structured* and *terminating* RAM program a $\Pi[i] = \mathsf{jump}(*, i')$ instruction always represents either an *if conditional* (when $i' > i$) or a *for loop* (when $i' < i$).

Now we are ready to formulate the formal grammar that defines the space of *well-structured* and *terminating* RAM programs. In this grammar the non-terminal symbol ***RAMInstruction*** refers to an instruction from our set of primitive RAM instructions (in more detail, refers to an *inc*, *dec* or *set* instruction since $test$, cmp and jmp instructions are replaced by their corresponding control flow constructs from *structured programming*). Please note that the grammar accepts the structured program of Figure 2; we exemplify our representation with a generic high-level structured programming language-like, that supports `If` conditionals and `For` loops, as well as `tuples` (to store the state variables) as could be the case of common structured programming languages like *Python*, *C++* or *Java*.

$$\begin{aligned} \Pi &::= post_state = pre_state; \\ &\quad\ Instruction(s) \\ &\quad\ return\ post_state; \\ Instruction(s) &::= If; Instruction(s) \mid \\ &\quad\ For; Instruction(s) \mid \\ &\quad\ RAMInstruction; Instruction(s) \mid \\ &\quad\ ; \\ If &::= if(Condition)\{Instruction(s)\} \\ Condition &::= (r > 0) \mid (r == 0) \mid (r_1 > r_2) \mid (r_1 == r_2) \mid (r_1 < r_2) \\ For &::= for(r = 0; r < |\Omega|; r++)\{Instruction(s)\}| \\ &\quad\ for(r = |\Omega| - 1; z \geq 0; r--)\{Instruction(s)\} \end{aligned}$$

Example. Our RAM model for the *pancake sorting* domain has $2 \times |\Omega| + 2 + 2$ registers, where $|R_X| = |R'_X| = |\Omega|$ indicates the number of pancakes. State variables are represented with $|\Omega|$ RAM registers $r = f_i(o)$, with domain $D_r = [0, \Omega)$, where $f_i(o)$ indicates the size of pancake $o \in \Omega$ located the ith at the stack of pancakes. In this domain only $1-$ary properties of the pancakes are necessary to represent states, so one can use a single auxiliary RAM register to enumerate the pancakes and iterate over the state variables; Figure 2, showed a *well-structured* and *terminating* RAM program that enumerates the pancakes with the *latent register* $z_1 \in R_Z$, no matter the actual number of pancakes.

4 Target Languages as RAM Program Spaces

When available, a particular *target language* can be exploited to implement more effective enumerations of the space of candidate RAM programs. This section shows that different formal grammars can be defined over our RAM *instruction set* to exploit the particular structure of relevant modeling tasks, such as modeling STRIPS planning actions or the update rule of a *cellular automaton*.

4.1 Target Language 1: STRIPS

Classical planning addresses the computation of sequences of deterministic actions that transform an initial state, into a state that satisfies a given set of goals. Figure 3 illustrates a state-transition from a three-disk classical planning instance of the *Tower of Hanoi* (ToH). The infinite set of state transitions of the classical planning domain of the ToH are compactly modeled with a single STRIPS operator (Figure 4), no matter the actual number of disks. The `move` operator leverages three first-order predicates that indicate: whether an object is `on` top of another object, whether an object is `smaller` than another object and whether an object is `clear` i.e. has no other object on top of it.

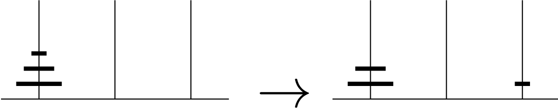

Figure 3: Example of a state-transition in a three-disk instance of the *Tower of Hanoi* domain, where the smallest disk is moved from the leftmost peg to the rightmost peg.

```
(:action move
 :parameters (?disc ?from ?to)
 :precondition (and (smaller ?to ?disc) (on ?disc ?from)
                    (clear ?disc) (clear ?to))
 :effect  (and (clear ?from) (on ?disc ?to)
               (not (on ?disc ?from)) (not (clear ?to)))
```

Figure 4: STRIPS operator that models all the state transitions of the *Tower of Hanoi*, no matter the number of disks.

Let Ψ be the set of first-order k-arity predicates of a classical planning domain, then $\Psi(X)$ is the set of atomic formulas $f(x_1, \ldots, x_k)$ obtained by applying the k-ary predicates $f \in \Psi$ to any m-tuple $X = \langle x_1, \ldots, x_m \rangle$ of distinct symbols (either constants or variables), with $1 \leq \ldots, m \leq k$. For instance, if Ψ contains the single binary predicate on, and $X = \langle x_1, x_2, x_3 \rangle$. Then, $\Psi(X) = \{on(x_i, x_j) | 1 \leq i, j \leq 3\}$. A k-ary STRIPS operator $op \in \mathcal{O}$, defined on the set of predicates Ψ, is a tuple $\langle par(op), pre(op), eff^+(op), eff^-(op) \rangle$, where $par(op)$ is a k-tuple of variables, $pre(op)$, $eff^+(op)$, and $eff^-(op)$ are three sets of atoms on $\Psi(par(op))$ representing the *preconditions*, *positive*, and *negative effects* of the operator. WLOG, the STRIPS formalism assumes that effects are consistently defined, i.e. $eff^-(op) \subseteq pre(op)$ and $eff^+(op) \bigcap eff^-(op) = \emptyset$.

Next, we formulate our RAM representation for the state transitions of a STRIPS system:

- **States**. Each ground predicate $p(o_1, \ldots, o_k)$ is represented by one RAM register $r \in R_X$, with domain $D_r = \{0, 1\}$, and another one in R'_X. This state representation requires then a tuple of $|R_X| = |R'_X| = \sum_{k \geq 0} n_k |\Omega|^k$ Boolean registers, where n_k is the number of first-order predicates with arity k, and $|\Omega|$ is the number of world objects[2]. The RAM model defines also $|R_Z| = \max_{p \in \Psi} arity(p)$ latent registers, i.e. given by the largest arity of a predicate in Ψ. The *latent registers* have domain $D_z = [0, \Omega)$ so they can index registers in R_X and R'_X.
- **Transitions**. Each STRIPS operator is represented as a $\Pi : R \rightarrow R_{X'}$ RAM program that first, it copies the *pre-state* into the *post-state*. Then the program checks whether the preconditions of the STRIPS operator hold in the *pre-state*, and iff this is the case, it updates and returns the *post-state*. Figure 5 shows a RAM program modeling the move STRIPS operator, that represents the state-transitions of the ToH, no matter the number of disks.

Next we show the formal grammar that confines the space of candidate RAM programs to the particular structure of STRIPS.

[2] $\sum_{k \geq 0} n_k |\Omega|^k$ is also the number of propositions that result from grounding a STRIPS classical planning instance.

```
State move(State pre_state, int disc, int from, int to){
   State post_state=pre_state;
   if (pre_state(smaller,to,disc) == 1){
       if (pre_state(on,disc,from) == 1){
           if (pre_state(clear,disc) == 1){
               if (pre_state(clear,to) == 1){
                   post_state(clear,from)=1;
                   post_state(on,disc,to)=1;
                   post_state(on,disc,from)=0;
                   post_state(clear,to)=0;
    }}}}
    return post_state;
}
```

Figure 5: RAM program modeling the move operator from the *Tower of Hanoi*.

$$
\begin{aligned}
\Pi &::= post_state = pre_state; \\
&\quad\; Instruction(s) \\
&\quad\; return\; post_state; \\
Instruction(s) &::= If; Instruction(s); \\
If &::= if(Condition)\{Instruction(s)\} \mid \\
&\quad\; if(Condition)\{Assigment\} \\
Condition &::= (pre_state(f, z_1, \ldots, z_k) == 0) \mid \\
&\quad\; (pre_state(f, z_1, \ldots, z_k) == 1) \\
Assigment(s) &::= post_state(f, z_1, \ldots, z_k) = 0;\; Assigment(s) \mid \\
&\quad\; post_state(f, z_1, \ldots, z_k) = 1;\; Assigment(s) \mid \\
&\quad\; ;
\end{aligned}
$$

where $Condition(s)$ is a conjunction of assertions over predicates $p(z_1, \ldots, z_k)$ instantiated with the action arguments (i.e. the latent registers $z \in R_Z$), and representing the operator preconditions ($==$ denotes the equality operator, $=$ indicates an assignment, and a semicolon denotes the end of a program instruction). *Assignment(s)* is a conjunction of assignments representing the operator positive/negative effects; in more detail $p(z_1, \ldots, z_k) = 1$ denotes a *positive* effect while $p(z_1, \ldots, z_k) = 0$ denotes a *negative* effect.

Universally Quantified Conditional Effects. STRIPS biases modeling towards state transitions that only require checking and updating a fixed (and small) number of state variables. As a matter of fact, this number is actually upper bounded by the *header* (number and type of parameters) of the corresponding STRIPS operator. Our approach goes beyond STRIPS and can also naturally model actions with *universally quantified preconditions* and *conditional effects* that are able to check and update and unbound number of state variables; that is the particular case of the fragment of the PDDL planning language [26] that is obtained by setting the :universal-preconditions and :conditional-effects requirements. In this case our state representation is the same as for STRIPS systems but the target language is extended with *for loop* instructions from RAM programs, that leverage latent variables to iterate over the set of world objects.

4.2 Target language 2: Cellular automata

A cellular automaton is a *zero-player game*, defined as a collection of finite-domain variables, called *cells*, that are situated on a grid of a defined shape; the grid can be defined in any finite number of dimensions and the number of cells is unbounded. The initial state of a cellular automaton is set assigning a value to every cell in the automaton. The state transitions of a cellular automaton are given by a fixed set of *update rules*, that define the next value of a cell w.r.t.: (i), its current value and (ii), the current value of its neighbour cells.

```
State R30(State pre_state, int z1, int z2, int z3){
    State post_state=pre_state;
    for(z1=0; z1<|Ω|; z1++){
        z2=z1; z2--;
        z3=z1; z3++;
        if(pre_state(z1)==1){
            if(pre_state(z2)==1){
                if(pre_state(z3)==1){
                    post_state(z1)=0;
                }}}
        ...
        if(pre_state(z1)==0){
            if(pre_state(z2)==0){
                if(pre_state(z3)==0){
                    post_state(z1)=0;
                }}}
    }
    return post_state;
}
```

Figure 6: Fragment of RAM program modeling the *Rule 30* automaton that leverages three *latent registers* $R_Z = \{z_1, z_2, z_3\}$.

The simplest non-trivial cellular automaton is *one-dimensional*, with two possible values per cell $\{0, 1\}$, and the cell's neighborhood defined as the adjacent cells on either side of a given cell. A cell and its two neighbors form then a three-cell neighborhood, so there are $2^3 = 8$ possible *neighborhood patterns*, and $2^8 = 256$ different one-dimensional cellular automata, generally named by their *Wolfram code* [54]. Table 1 shows the *Rule 30* cellular automaton.

Neighbrhd. pattern	111	110	101	100	011	010	001	000
Next value	0	0	0	1	1	1	1	0

Table 1: *Rule 30* cellular automaton. For each neighborhood pattern, the rule specifies the next value for the center cell.

We formulate our RAM representation for the state transitions of a one-dimensional cellular automaton as follows:

- **States**. The RAM representation for the state of a one-dimensional cellular automaton is a tuple of $|R_X| = |R'_X| = |\Omega|$ Boolean registers (i.e. one register per each cell in the automaton), plus three *latent registers* $R_Z = \{z_1, z_2, z_3\}$, with domain $D_z = [0, \Omega)$, to index the registers corresponding to a given cell and its two neighbour cells.
- **Transitions**. The set of update rules of a one-dimensional cellular automaton is represented as a $\Pi : R \to R_{X'}$ RAM program that first, it copies the *pre-state* into the *post-state*. Then, for each cell in the automaton, the program updates the cell value according to the value of its corresponding neighborhood. Figure 6 shows a fragment of the RAM program that models the state-transitions of the *Rule 30* automaton (no matter its number of cells).

Next, we show the formal grammar that confines the space of RAM programs to the particular structure of *one-dimensional cellular automata*.

$$
\begin{aligned}
\Pi &::= post_state = pre_state;\\
&\quad for(z1 = 0; z1 < |\Omega|; z1++)\{\\
&\quad z2 = z1; dec(z2);\\
&\quad z3 = z1; inc(z3);\\
&\quad Instruction(s)\\
&\quad \}\\
&\quad return(post_state);\\
Instruction(s) &::= If; Instruction(s);\\
If &::= if(Condition)\{Instruction(s)\} \mid\\
&\quad if(Condition)\{Assigment\}\\
Condition &::= (pre_state(z) == 0) \mid (pre_state(z) == 1)\\
Assigment &::= post_state(z1) = 0; \mid\\
&\quad post_state(z1) = 1;
\end{aligned}
$$

5 Synthesis of Procedural Models

This section formalizes the problem of synthesizing structured programs that model the *state-transitions* of a given *deterministic transition system*. Then, the section details our synthesis method and its theoretical properties.

5.1 The problem

The *problem* of synthesizing the RAM programs that model the *state-transitions* of a *deterministic transition system* is formalized as a tuple $\langle \mathcal{E}, \mathcal{M} \rangle$, where:

- $\mathcal{E}$ is a *set of examples*, s.t. each example $e = (s_t, a, s_{t+1})$ comprises (i), a *pre-state* (ii), a *transition label* and (iii), its *post-state*. We denote $\mathcal{E}_a \subseteq \mathcal{E}$, the subset of examples with transition label a.
- $\mathcal{M}$ is a *RAM model* (as defined above) that comprises registers $R = \{R_X, R_{X'}, R_Y, R_Z\}$, and s.t. registers in R_X (and in $R_{X'}$ too) model the state space of the *deterministic transition system*.

A *solution* to the synthesis problem is a finite and non-empty set of RAM programs $\Pi_a : R \to R_{X'}$ s.t. for all the examples $e = (s_t, a, s_{t+1})$ in the subset $\mathcal{E}_a \subseteq \mathcal{E}$, the execution of Π_a in the *pre-state* s_t, exactly outputs its corresponding *post-state* s_{t+1}.

Example. We exemplify the synthesis problem in the *pancake sorting* domain, with a set of three examples $\mathcal{E} = \{e_1, e_2, e_3\}$ that represent three transitions of a four-pancake instance; $e_1 = ((3, 2, 1, 4), \texttt{flip}(2), (1, 2, 3, 4))$, $e_2 = ((2, 3, 1, 4), \texttt{flip}(1), (3, 2, 1, 4))$ and $e_3 = ((1, 3, 2, 4), \texttt{flip}(2), (2, 3, 1, 4))$. The RAM model $\mathcal{M}$ comprises four registers $R_X = \{r_{x_1}, r_{x_2}, r_{x_3}, r_{x_4}\}$ encoding the *pre-state* of a transition (each register stores the size of the pancake at its corresponding position in the stack), four registers $R'_X = \{r'_{x_1}, r'_{x_2}, r'_{x_3}, r'_{x_4}\}$ encoding the *post-state* of a transition, the two FLAGS registers $R_Y = \{CF, ZF\}$, two latent registers $R_Z = \{z_1, z_2\}$, and the RAM instruction set defined above.

5.2 The method

Our synthesis method implements a *Best First Search* (BFS) in the space of *well-structured* and *terminating* RAM programs, that can be built with a maximum number of program lines n and latent registers $|R_z|$. In more detail, each search node corresponds to a *partially specified* RAM program $\Pi_a : R \to R_{X'}$; by partially specified we mean that some of the program lines may be undefined, because they are not programmed yet. Starting from the *empty program* (i.e. the partially specified program whose all lines are undefined), the space of programs is enumerated with a search operator that programs an instruction, at an undefined program line $0 \leq i < n$. This search operator is only applicable when program line i is undefined; initially $i := 0$, and after line i is programmed $i := i + 1$.

To reduce memory requirements we implement a *frontier BFS* [33], that store only the open list of generated nodes but not the closed list of expanded nodes. Nodes in the open list are sorted w.r.t. the following evaluation functions:

- $f_{\#ifs}(\Pi_a)$, returns the number of *conditional if* instructions of a given RAM program Π_a.
- $f_{\#loops}(\Pi_a)$, returns the number of *for loop* instructions of a given RAM program Π_a.

- $f_{gc}(\Pi_a, \mathcal{E}_a) = \sum_{e \in \mathcal{E}_a} \sum_{x \in s_{t+1}} |s_{t+1}[x] - s'_{t+1}[x]|$, where Π_a is a RAM program, $\mathcal{E}_a \subseteq \mathcal{E}$ is the subset of examples with label a, and s'_{t+1} is the state produced by executing Π_a on the pre-state s_t of a example $e \in \mathcal{E}_a$. For all the examples $e \in \mathcal{E}_a$, the function f_{gc} accumulates the number of state variables in s'_{t+1} whose value mismatch with the corresponding post-state s_{t+1}.

In more detail, nodes in the open list are sorted maximizing the function $f_{\#loops}(\Pi_a)$, then breaking ties by maximizing $f_{\#ifs}(\Pi_a)$ and last, breaking ties by minimizing $f_{gc}(\Pi_a, \mathcal{E}_a)$. This combination of the evaluation functions biases synthesis towards models with larger number of conditions that cover the given set of examples. In addition we implement a *pruning rule* $R1$ that reduces the search space while it preserves the solution space (i.e. preserves completeness). The rule leverages the fact that between the *pre-state* and the *post-state* of an example there is no intermediate states:

- $R1$. We do not allow programming RAM instructions that set a register in R'_X with a value different from its value at the post-state of a given example.

Properties. Our method for synthesizing RAM programs that model the set of *state-transitions* of a given *discrete system* is terminating; termination follows from a terminating *search algorithm* and *evaluation functions*. Regarding the former, our method implements a *frontier BFS* which is known to be terminating at finite search spaces [33], we recall that our search space is finite since the maximum number of program lines n is bounded. Regarding the evaluation functions, $f_{\#loops}$ and $f_{\#ifs}$, they terminate in n steps, where n is the number of lines of the program to evaluate. Last, f_{gc} terminates iff the program executions terminate, which is always true since our candidate RAM programs are by definition *well-structured* and *terminating*. Our method is *sound*, since it only outputs RAM programs Π_a that cover the full set of examples $\mathcal{E}_a$. Last, our method is *complete* provided that there exists a solution within the given number of program lines n and latent registers $|R_Z|$.

6 Evaluation

We report results on the **synthesis** and **validation** of procedural models for a wide range of *discrete systems*, and using the following *target languages*:

1. STRIPS. We synthesize 49 action schemes from 13 domains (*FD benchmarks*[3] and PLANNING.DOMAINS [39], ranging from *Gripper* to *Transport* in Table 2). For each domain, examples are generated using 10 IPC instances and computing: (i), one random walk per instance of at most 50 steps with TARSKI [20] and (ii), solution plans for those instances with the LAMA [44] setting of FD [27]. *Synthesis examples* $\mathcal{E}_{synth}$, and *validation examples* $\mathcal{E}_{test}$, are the traversed *(pre-state, action, post-state)* tuples.
2. STRIPS *with universally quantified conditional effects*. We synthesize 7 action schemes from 3 different classical planning domains (namely, *briefcase*, *maintenance*, and *elevators*) that contain actions with universally quantified conditional effects. Examples are generated as for the STRIPS domains.
3. *Cellular automaton*. We synthesize the update rule of four well-known one-dimensional *cellular automaton* [54], namely rule *30*, rule *90*, rule *110* and rule *184*. For each rule, $\mathcal{E}_{synth}$ examples are 20 state-transitions of a 19-cell automaton which starts with 3 ones in the three central cells, while $\mathcal{E}_{test}$ examples are 100 state-transitions of 99-cell automata.
4. *RAM*. We synthesize the program that models the `flip` action scheme from *pancake sorting*; $\mathcal{E}_{synth}$ examples comprise 16 flips of a randomly generated 9-pancake instance, while $\mathcal{E}_{test}$ examples comprise 98 flips of a randomly generated 50-pancake instance.

For STRIPS, STRIPS *with quantified effects* and the *Cellular automaton* domains, our synthesis method exploits their corresponding grammars that confine the space of possible programs. For the *pancake sorting* we consider the full space of RAM programs. All experiments are performed in an Ubuntu 22.04.2 LTS, with Intel® Core i5-10300H @ 2.50GHz x 8-core processor and 16GB of RAM[4].

Synthesis. Our synthesis method is fed with two input parameters: (i) the maximum number of *program lines* n and (ii), the maximum number of *latent variables* $|R_Z|$. Target languages like STRIPS or *Cellular Automaton* have a known upper-bound in the number of required program lines. In these domains we used those upper-bounds otherwise (in *pancake sorting* and domains with quantified effects) we incrementally increase program size until a program is synthesized that satisfies the input examples; similar to what is done in SAT-planning with the *planning horizon* [32]. The first six columns of Table 2 report the obtained *synthesis results*. For each domain and action schema we report, the number of synthesis examples $|\mathcal{E}_{synth}|$, the number of lines in the solution (n_{sol}) out of upper-bound n_{max}, the time elapsed T_{synth}, and the number of expanded (Exp.) and evaluated (Eval.) search nodes. Note that T_{synth} takes less than a sec in almost half of the cases, and less than a minute in all of them, even when $|\mathcal{E}_{synth}|$ comprises several hundreds of examples; we effectively deal with hundreds of synthesis examples progressively adding input examples to the synthesis set, so they may act as counter-examples of candidate solutions [48]. Results show that in STRIPS domains the node to expand is perfectly selected by BFS, since n_{sol} equals the number of expansions, only loosing time in node evaluation. For the rest of languages, there are 3 cases where expansions go beyond 10K nodes, but still keeping a good trade-off between time and memory (expanded and evaluated are nodes are close).

Validation. Validation examples in $\mathcal{E}_{test}$ are larger than the $\mathcal{E}_{synth}$ examples (i.e. larger number of objects). The three last columns of Table 2 (namely, the number of validation examples $|\mathcal{E}_{test}|$, validation time T_{test}, and rate of $\mathcal{E}_{test}$ examples *successfully validated* %✓) summarize the obtained results when validating the synthesized models in the $\mathcal{E}_{test}$ examples. An example is *successfully validated* iff the execution of the synthesized model, at the example pre-state, exactly produces the corresponding post-state of the example; all the synthesized action schemes generalize well to the new and larger examples in $\mathcal{E}_{test}$, validating every domain in less than 10s.

We also ran two experiments to compare our method with the performance of FAMA [1] at STRIPS domains, which are summarized as follows: (i), When we use our benchmarks to synthesize action models, FAMA meets the limitations of SAT-based approaches to handle hundreds of examples, getting stuck in the synthesis. (ii) When using the minimal learning sets from the FAMA benchmarks our synthesis performance gets close to FAMA, despite some of the synthesized models fail to generalize because those minimal learning sets miss representative transitions. Last but not least, systems for learning planning action models, like FAMA [1], or ARMS [56], assume that any state transition is modeled with a single STRIPS action so they fail to learn actions with *quantified effects*, models for cellular automata, or for the *pancake sorting* domain.

[3] https://github.com/aibasel/downward-benchmarks

[4] Framework: https://github.com/jsego/bfgp-pp

Domain	Action Schema	$\|\mathcal{E}_{synth}\|$	n_{sol}/n_{max}	T_{synth}	Exp./Eval.	$\|\mathcal{E}_{test}\|$	T_{test}	%✓
gripper	move	138	31/82	0.45	31/554	231	0.12	100%
	pick	116	58/82	1.21	58/1271	203	0.11	100%
	drop	111	58/82	1.11	58/1271	201	0.11	100%
miconic	depart	40	35/49	1.54	35/458	66	0.21	100%
	up	162	35/49	5.09	35/458	172	0.55	100%
	down	137	35/49	4.37	35/458	168	0.52	100%
	board	64	30/49	1.74	30/359	107	0.33	100%
driverlog	disembark-truck	29	94/241	3.58	94/4977	46	0.07	100%
	board-truck	34	94/241	3.90	94/4977	54	0.10	100%
	drive-truck	42	163/241	10.89	163/9962	80	0.13	100%
	walk	90	93/241	6.16	93/4944	88	0.14	100%
	unload-truck	67	91/241	4.94	91/4724	71	0.10	100%
	load-truck	73	91/241	5.24	91/4724	76	0.11	100%
parking	move-car-to-curb	124	24/61	0.31	24/361	150	0.06	100%
	move-curb-to-curb	43	17/61	0.10	17/233	40	0.01	100%
	move-curb-to-car	135	22/61	0.37	24/410	153	0.06	100%
	move-car-to-car	144	29/61	0.52	29/440	158	0.12	100%
visitall	move	548	14/19	5.37	14/79	1183	7.59	100%
grid	move	187	47/232	26.14	47/2092	254	1.08	100%
	unlock	9	155/232	31.66	155/9085	8	0.05	100%
	pickup	33	50/232	8.82	50/2295	22	0.11	100%
	putdown	31	50/232	8.18	50/2295	21	0.09	100%
	pickup-and-loose	11	97/232	16.66	97/5086	15	0.08	100%
blocks	stack	89	22/28	0.07	22/194	155	0.04	100%
	unstack	91	22/28	0.07	22/194	160	0.04	100%
	put-down	79	13/28	0.03	13/95	131	0.03	100%
	pick-up	77	13/28	0.03	13/95	126	0.03	100%
satellite	calibrate	11	104/265	2.12	104/5918	9	0.03	100%
	switch-on	73	50/265	1.73	50/2546	91	0.14	100%
	switch-off	65	49/265	1.36	49/2459	68	0.11	100%
	turn-to	143	103/265	9.01	103/5915	520	0.93	100%
	take-image	32	174/265	9.01	174/11483	254	0.46	100%
npuzzle	move	729	17/19	1.63	17/107	3390	5.17	100%
hanoi	move	307	27/46	0.75	27/333	2229	1.57	100%
rovers	communicate-rock-data	90	24/160	1.45	24/673	46	0.09	100%
	calibrate	92	14/160	0.56	14/353	61	0.12	100%
	communicate-soil-data	74	26/160	1.40	26/752	43	0.09	100%
	sample-rock	42	23/160	0.68	23/728	24	0.12	100%
	communicate-image-data	91	24/160	1.47	24/673	44	0.09	100%
	sample-soil	36	23/160	0.66	23/728	25	0.05	100%
	navigate	174	17/160	1.34	17/483	113	0.22	100%
	drop	51	11/160	0.29	11/289	32	0.07	100%
	take-image	70	17/160	0.62	17/475	44	0.09	100%
ferry	debark	81	30/40	1.87	30/340	93	0.15	100%
	sail	216	27/40	3.54	27/285	239	0.39	100%
	board	81	30/40	1.77	30/340	96	0.16	100%
transport	drive	217	11/40	0.40	11/89	427	0.58	100%
	pick-up	79	19/40	0.26	19/205	47	0.06	100%
	drop	76	19/40	0.25	19/205	45	0.07	100%
briefcase	move	167	15/15	2.27	27215/27245	278	0.07	100%
	take-out	68	4/4	0.01	3/4	119	0.03	100%
	put-in	71	8/8	0.01	7/18	124	0.03	100%
elevators	stop	127	18/18	9.22	67283/67326	142	0.18	100%
	up	94	7/7	0.04	17/28	134	0.19	100%
	down	85	7/7	0.04	17/30	119	0.16	100%
maintenance	workat	47	9/9	0.08	59/65	53	0.03	100%
cellular	rule184	20	95/95	0.15	242/257	100	0.26	100%
	rule30	20	95/95	0.09	155/257	100	0.25	100%
	rule90	20	95/95	0.08	158/256	100	0.23	100%
	rule110	20	95/95	0.11	189/257	100	0.22	100%
pancakes	pancakes-flip	16	8/8	12.17	20163/20837	98	0.07	100%

Table 2: For each domain and action schema we report, the number of synthesis examples $|\mathcal{E}_{synth}|$, the number of lines in the solution (n_{sol}) out of an upper-bound n_{max}, the time elapsed in seconds T_{synth}, the number of expanded (Exp.) and evaluated (Eval.) nodes by the BFS, the number of validation examples $|\mathcal{E}_{test}|$, the validation time in secs T_{test} and the rate of $\mathcal{E}_{test}$ examples successfully validated %✓.

7 Conclusions

We presented an innovative approach for synthesizing white-box models of transition systems as structured programs. Our approach synthesize compact models of state transitions that cannot be modeled by a single STRIPS action and that require *latent variables* missing in the state representation; we showed that this kind of state transitions are produced by actions with *universally quantified conditional effects*, update rules of cellular automata, or vector manipulations like the `flip` action from *pancake sorting*. We also showed that our approach can leverage different *target languages* (when available) to confine the solution space, without modifying the synthesis method. Our work is connected to the *heuristic search* approach to *generalized planning* [29, 49, 50]; our research agenda is formalizing an effective heuristic search framework that subsumes both tasks.

The computation of properties and relations that generalize over world objects is studied since the early days of AI, with seminal works on the *blocksworld* [17, 53]. This family of computational problems is traditionally characterized by the use of FOL representations and it is studied under different names and approaches, including ILP and *relational learning* [38, 8, 24, 15]. Full-observability of state transitions (the setting of this paper) is the most basic setting for this family of computational problems. We showed however that such setting is still challenging at domains where the number of state variables that are checked, or updated, is unbounded. We plan to extend our *structured programming* approach to address more challenging settings that consider *action discovery* [52, 51], *partial observability* [3], *noisy examples* [37], or *non-deterministic models* [40], as it has been done in the *automated planning* [28, 4, 2] and the ILP literature [42].

Acknowledgments

This work is supported by the H2020 AIPLAN4EU project #101016442. Javier Segovia-Aguas is funded by TAILOR, AGAUR SGR and the Spanish grant PID2019-108141GB-I00. Jonathan Ferrer-Mestres is supported by the CSIRO MLAI Future Science Platform. Sergio Jiménez is funded by the Spanish MINECO project PID2021-127647NB-C22.

References

[1] Diego Aineto, Sergio Jiménez Celorrio, and Eva Onaindia, 'Learning action models with minimal observability', *AIJ*, **275**, 104–137, (2019).
[2] Diego Aineto, Sergio Jiménez, and Eva Onaindia, 'A comprehensive framework for learning declarative action models', *JAIR*, **74**, 1091–1123, (2022).
[3] Eyal Amir and Allen Chang, 'Learning partially observable deterministic action models', *JAIR*, **33**, 349–402, (2008).
[4] Ankuj Arora, Humbert Fiorino, Damien Pellier, Marc Métivier, and Sylvie Pesty, 'A review of learning planning action models', *KER*, **33**, (2018).
[5] Masataro Asai and Christian Muise, 'Learning neural-symbolic descriptive planning models via cube-space priors: The voyage home (to strips)', in *IJCAI*, (2020).
[6] Christel Baier and Joost-Pieter Katoen, *Principles of model checking*, MIT press, 2008.
[7] Jorge A Baier and Sheila A McIlraith, 'Knowledge-based programs as building blocks for planning', *AIJ*, **303**, 103634, (2022).
[8] Francesco Bergadano and Daniele Gunetti, *Inductive Logic Programming: from machine learning to software engineering*, MIT press, 1996.
[9] George S Boolos, John P Burgess, and Richard C Jeffrey, *Computability and logic*, Cambridge university press, 2002.
[10] Ronen I Brafman, David Tolpin, and Or Wertheim, 'Probabilistic programs as an action description language', in *AAAI*, (2023).
[11] J Burton Browning and Bruce Sutherland, 'Working with numbers', in *C++ 20 Recipes*, 115–145, Springer, (2020).
[12] Josef Cibulka, 'On average and highest number of flips in pancake sorting', *Theoretical Computer Science*, **412**(8-10), 822–834, (2011).
[13] Edmund M Clarke, 'Model checking', in *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pp. 54–56. Springer, (1997).
[14] Stephen N Cresswell, Thomas L McCluskey, and Margaret M West, 'Acquiring planning domain models using locm', *KER*, **28**(2), 195–213, (2013).
[15] Luc De Raedt, *Logical and relational learning*, Springer Science & Business Media, 2008.
[16] Ronald Fagin, Joseph Y Halpern, Yoram Moses, and Moshe Vardi, *Reasoning about knowledge*, MIT press, 2004.
[17] Richard E Fikes, Peter E Hart, and Nils J Nilsson, 'Learning and executing generalized robot plans', *AIJ*, **3**, 251–288, (1972).
[18] Paul A Fishwick and Richard B Modjeski, *Knowledge-based simulation: methodology and application*, volume 4, Springer Science & Business Media, 2012.
[19] Mark S Fox, Nizwer Husain, Malcolm McRoberts, and YV Reddy, 'Knowledge based simulation: An artificial intelligence approach to system modeling and automating the simulation life cycle.', Technical report, The Robotics Institute Carnegie Mellon University, (1988).
[20] Guillem Francés, Miquel Ramirez, and Collaborators. Tarski: An AI planning modeling framework. https://github.com/aig-upf/tarski, 2018.
[21] Hector Geffner, 'Target languages (vs. inductive biases) for learning to act and plan', in *AAAI*, pp. 12326–12333, (2022).
[22] Hector Geffner and Blai Bonet, 'A concise introduction to models and methods for automated planning', *Synthesis Lectures on AI and Machine Learning*, **8**(1), 1–141, (2013).
[23] Michael Genesereth and Yngvi Björnsson, 'The international general game playing competition', *AI Magazine*, **34**(2), 107–107, (2013).
[24] Lise Getoor, Nir Friedman, Daphne Koller, Avi Pfeffer, and Benjamin Taskar, 'Probabilistic relational models', *Introduction to statistical relational learning*, **8**, (2007).
[25] Malik Ghallab, Dana Nau, and Paolo Traverso, *Automated Planning: theory and practice*, Elsevier, 2004.
[26] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise, 'An introduction to the planning domain definition language', *Synthesis Lectures on AI and Machine Learning*, **13**(2), 1–187, (2019).
[27] Malte Helmert, 'The fast downward planning system', *JAIR*, **26**, 191–246, (2006).
[28] Sergio Jiménez, Tomás De La Rosa, Susana Fernández, Fernando Fernández, and Daniel Borrajo, 'A review of machine learning for automated planning', *The Knowledge Engineering Review*, **27**, (2012).
[29] Sergio Jiménez, Javier Segovia-Aguas, and Anders Jonsson, 'A review of generalized planning', *KER*, **34**, (2019).
[30] Brendan Juba, Hai S Le, and Roni Stern, 'Safe learning of lifted action models', in *KR*, (2021).
[31] Michael Katz, Dany Moshkovich, and Erez Karpas, 'Semi-black box: rapid development of planning based solutions', in *AAAI*, (2018).
[32] Henry Kautz and Bart Selman, 'Unifying sat-based and graph-based planning', in *IJCAI*, volume 99, pp. 318–325, (1999).
[33] Richard E Korf, Weixiong Zhang, Ignacio Thayer, and Heath Hohwald, 'Frontier search', *JACM*, **52**(5), 715–748, (2005).
[34] Jiří Kučera and Roman Barták, 'Louga: learning planning operators using genetic algorithms', in *Pacific Rim Knowledge Acquisition Workshop*, (2018).
[35] Leonardo Lamanna, Alessandro Saetti, Luciano Serafini, Alfonso Gerevini, and Paolo Traverso, 'Online learning of action models for pddl planning.', in *IJCAI*, pp. 4112–4118, (2021).
[36] Marvin L Minsky, 'Recursive unsolvability of post's problem of "tag" and other topics in theory of turing machines', *Annals of Mathematics*, 437–455, (1961).
[37] Kira Mourao, Luke S Zettlemoyer, Ronald Petrick, and Mark Steedman, 'Learning strips operators from noisy and incomplete observations', in *UAI*, (2012).
[38] Stephen Muggleton, *Inductive logic programming*, Morgan Kaufmann, 1992.
[39] Christian Muise, 'Planning. domains', *ICAPS system demonstration*, 242–250, (2016).
[40] Hanna M Pasula, Luke S Zettlemoyer, and Leslie Pack Kaelbling, 'Learning symbolic models of stochastic domains', *JAIR*, **29**, 309–352, (2007).
[41] Judea Pearl and Dana Mackenzie, *The book of why: the new science of cause and effect*, Basic books, 2018.
[42] Luc De Raedt, Kristian Kersting, Sriraam Natarajan, and David Poole, *Statistical relational artificial intelligence: Logic, probability, and computation*, Morgan & Claypool Publishers, 2016.
[43] Raymond Reiter, 'A theory of diagnosis from first principles', *AIJ*, **32**(1), 57–95, (1987).
[44] Silvia Richter, Matthias Westphal, and Malte Helmert, 'Lama 2008 and 2011', in *International Planning Competition*, pp. 117–124, (2011).
[45] Ivan D Rodriguez, Blai Bonet, Javier Romero, and Hector Geffner, 'Learning first-order representations for planning from black-box states: New results', in *KR*, pp. 539–548, (2021).
[46] Gabriele Röger, Malte Helmert, and Bernhard Nebel, 'On the relative expressiveness of adl and golog: The last piece in the puzzle.', in *KR*, pp. 544–550, (2008).
[47] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, et al., 'Mastering atari, go, chess and shogi by planning with a learned model', *Nature*, **588**(7839), 604–609, (2020).
[48] Javier Segovia-Aguas, Sergio Jiménez Celorrio, Laura Sebastiá, and Anders Jonsson, 'Scaling-up generalized planning as heuristic search with landmarks', in *SoCS*, (2022).
[49] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson, 'Generalized planning as heuristic search', in *ICAPS*, pp. 569–577, (2021).
[50] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson, 'Computing programs for generalized planning as heuristic search', in *IJCAI*, (2022).
[51] Alejandro Suárez-Hernández, Javier Segovia-Aguas, Carme Torras, and Guillem Alenyà, 'Online action recognition', in *AAAI*, (2021).
[52] Alejandro Suárez Hernández, Javier Segovia Aguas, Carme Torras, and Guillem Alenyà Ribas, 'Strips action discovery', in *AAAI2020 workshop on Generalization in Planning*, pp. 1–9, (2020).
[53] Patrick H. Winston, 'Learning structural descriptions from examples', Technical report, Massachusetts Institute of Technology, USA, (1970).
[54] Stephen Wolfram et al., *A new kind of science*, volume 5, Wolfram media Champaign, 2002.
[55] N Wulff and J A Hertz, 'Learning cellular automaton dynamics with neural networks', *NeurIPS*, (1992).
[56] Qiang Yang, Kangheng Wu, and Yunfei Jiang, 'Learning action models from plan examples using weighted max-sat', *AIJ*, **171**, (2007).