Journal of Artificial Intelligence Research 51 (2014) 255-291 Submitted 03/14; published 09/14

Automaton Plans

Christer Bäckström CHRISTER.BACKSTROM@LIU.SE
Department of Computer Science
Linköping University
SE-581 83 Linköping, Sweden

Anders Jonsson ANDERS.JONSSON@UPF.EDU
Dept. Information and Communication Tecnologies
Universitat Pompeu Fabra
Roc Boronat 138
08018 Barcelona, Spain

Peter Jonsson PETER.JONSSON@LIU.SE
Department of Computer Science
Linköping University
SE-581 83 Linköping, Sweden

Abstract

Macros have long been used in planning to represent subsequences of operators. Macros can be used in place of individual operators during search, sometimes reducing the effort required to find a plan to the goal. Another use of macros is to compactly represent long plans. In this paper we introduce a novel solution concept called automaton plans in which plans are represented using hierarchies of automata. Automaton plans can be viewed as an extension of macros that enables parameterization and branching. We provide several examples that illustrate how automaton plans can be useful, both as a compact representation of exponentially long plans and as an alternative to sequential solutions in benchmark domains such as LOGISTICS and GRID. We also compare automaton plans to other compact plan representations from the literature, and find that automaton plans are strictly more expressive than macros, but strictly less expressive than HTNs and certain representations allowing efficient sequential access to the operators of the plan.

1. Introduction

In this paper we introduce a novel solution concept for planning that we call *automaton plans*. For ease of presentation we divide the introduction into two parts. In the first part we discuss existing concepts for plan representation from the literature. In the second part we describe the novel representation that we propose.

1.1 Plan Representations

Following the introduction of STRIPS planning (Fikes & Nilsson, 1971), it did not take researchers long to discover the utility of storing sequences of planning operators, or macros (Fikes, Hart, & Nilsson, 1972). Macros were first used as a tool during plan execution and analysis. However, macros turned out to have several other useful properties that have been exploited by researchers in the planning community ever since.

One such property is the possibility to compute cumulative preconditions and effects, effectively making macros indistinguishable from individual operators. A planning instance can be augmented with a set of macros, potentially speeding up the search for a solution since macros can reach further into the state space than individual operators. In the extreme, the search space over macros can be exponentially smaller than the search space over the original planning operators (Korf, 1987).

Moreover, if subsequences of operators are repeated, a hierarchy of macros can represent a plan more compactly than a simple operator sequence, replacing each occurrence of a repeating subsequence with a single operator (i.e. a macro). In the extreme, one can represent an exponentially long plan using polynomially many macros of polynomial length (Giménez & Jonsson, 2008; Jonsson, 2009). Sometimes it is even possible to generate such a compact macro plan in polynomial time, in which case macros can be viewed as a tool for complexity analysis, reducing the complexity of solving a particular class of planning instances.

Macros clearly show that there are advantages associated with plan representations that do not simply store the plan as a sequence of actions. Apart from the obvious purpose of saving space, there are other reasons for considering alternative representations. One important reason is to highlight properties of a plan that might not be apparent from a sequential representation and that can be exploited for increased planning efficiency. One prominent example is *partially ordered plans* that represent plans as sets of actions with associated partial orders. Partially ordered plans have often been used in planning to speed up search (McAllester & Rosenblitt, 1991). In general, the fact that there exists a compact representation of a plan implies that the planning instance exhibits some form of *structure* that might be possible to exploit for simpler and more efficient reasoning.

Potentially, there exist many alternative plan representations that can store plans compactly. Such compact representations can broadly be divided into two categories. The first type of plan representation stores a single plan, while the second type stores a set of plans. Macros are an illustrative example of the first type, and we have already seen that macro plans can be exponentially smaller than a sequential representation. An example of the second type is *reactive plans*, also known as universal plans or reactive systems, which represent one plan for each state from which the goal is reachable.

The usefulness of a compact representation depends of several factors.

1.1.1 COMPRESSION PROPERTIES

Clearly, one important property of a compact plan representation is its size. However, there is an information-theoretic bound on the compressibility of plans: a representation containing n bits can only distinguish between 2^n different plans, limiting the applicability of highly compact representations. There exist STRIPS instances with n variables having 2^{2^n-1} different plans (Bäckström & Jonsson, 2012, Construction 10), implying that at least $2^n - 1$ bits are needed to distinguish one particular solution from the rest. However, it may often suffice to represent a single solution that is not arbitrarily chosen. In the extreme, we can represent a solution compactly by storing the planning instance together with an algorithm for solving it.

1.1.2 ACCESS PROPERTIES

Another important property of a compact plan representation is the ability to access a particular action of the plan. Two alternative concepts have been proposed (Bäckström & Jonsson, 2012): *sequential* and *random* access. Sequential access implies that the actions of the plan can be retrieved in order, while random access implies that we can retrieve the action at any given position i of the plan. For both forms of access, ideally we should be able to retrieve actions in polynomial time, something that is not always possible.

1.1.3 VERIFICATION

A third property of a compact plan representation is being able to verify that the plan actually constitutes a solution to a given planning instance. The complexity of plan verification is directly related to the complexity of plan existence, i.e. determining whether or not an instance has a solution. Assume that the problem of plan verification for a compact plan representation R is in complexity class $\mathcal{C}$. Let X be the set of STRIPS instances $\boldsymbol{p}$ satisfying the following condition: if $\boldsymbol{p}$ is solvable there exists a solution to $\boldsymbol{p}$ that can be represented with R using $O(p(||\boldsymbol{p}||))$ bits, where p is a polynomial function and $||\boldsymbol{p}||$ is the number of bits in the representation of $\boldsymbol{p}$. Under these assumptions, the problem of plan existence for X is in complexity class $\mathsf{NP}^{\mathcal{C}}$: non-deterministically guess a compact plan in R and verify that the plan solves $\boldsymbol{p}$. For many choices of $\mathcal{C}$, plan existence for X is bounded away from **PSPACE**, i.e. easier than general STRIPS planning. We conclude that simple verification comes at a price: decreased expressiveness of the corresponding planning instances.

It is obviously difficult to identify representations that satisfy all three properties while being able to express reasonably complex plans. A reasonable approach is to look for *balanced* representations that are both expressive and computationally efficient. Let us evaluate macros according to the three properties above. In this paper we consider grounded macros that are totally ordered and allow nesting, i.e. a macro can involve other macros as long as this does not create cyclic dependencies among macros. We know that there exist examples for which macros provide a powerful compact representation mechanism. Macro plans have excellent access properties: both sequential and random access can be performed in polynomial time (Bäckström & Jonsson, 2012). They are also verifiable in polynomial time (Bäckström, Jonsson, & Jonsson, 2012b), implying that planning instances whose solutions can be represented using polynomial-size macro plans are easier than general STRIPS planning, but also less expressive.

1.2 Automaton Plans

In this paper we introduce a novel solution concept for planning, inspired by macros, that we call *automaton plans*. An automaton plan consists of a hierarchy of automata, each endowed with the ability to call other automata. At the bottom level of the hierarchy are the individual plan operators. Automaton plans can be viewed as an extension of macro plans along two dimensions. First, an automaton is parameterized, enabling it to compactly represent not just a single operator sequence but a whole family of sequences. Second, an automaton can branch on input, making it possible to store different subsequences of operators and distinguish between them by providing different input to the automaton.

The main motivation for automaton plans is to express plans compactly that macro plans cannot, while maintaining access properties and verification at a reasonable level. We present several examples of automaton plans, and show how they can be useful in a variety of ways. In domains such as Towers of Hanoi, automaton plans can represent exponentially long plans even more compactly than macro plans. Even when plans are not necessarily very long, the ability to parameterize plans makes it possible to store repeating families of action subsequences in common benchmark domains.

To test the usefulness of automaton plans, we formally compare automaton plans to other compact plan representations from the literature along the three dimensions discussed in Section 1.1: compression, access, and verification. Each macro plan is also an automaton plan, implying that automaton plans are at least as compressed as macro plans. Just like macro plans, automaton plans can be sequentially accessed in polynomial time. We also show that a subclass of automaton plans admit polynomial-time random access, although it is still unknown whether this result generalizes to all automaton plans. Finally, verification of automaton plans is $\mathbf{\Pi}_2^p$-complete, causing automaton plans to be strictly more expressive than macros.

Hierarchical Task Networks (Erol, Hendler, & Nau, 1996), or HTNs for short, can also be viewed as a type of compact plan representation. In addition to planning operators, an HTN defines a set of tasks, each with a set of associated methods for expanding the task. As the name suggests, the tasks are organized in a hierarchy with planning operators at the bottom. This hierarchy may be considerably more compact than the actual sequence of operators in a plan. In general, the plans represented by HTNs are not unique and search may be required to find a valid plan.

Instead of comparing automaton plans to general HTNs, we only consider totally ordered HTNs with unique plans, i.e. the methods associated with each task are mutually exclusive and specify a totally ordered expansion. We show that each automaton plan can be efficiently translated to such an HTN, causing HTNs to be at least as compressed as automaton plans. HTNs with unique plans can be sequentially accessed in polynomial time, but the same is not true for random access. Finally, plan existence for totally ordered HTNs is known to be **PSPACE**-hard (Erol et al., 1996), implying that verification of HTNs is harder than for automaton plans, in turn causing HTNs to be strictly more expressive.

Combining these results, an automaton plan appears to offer a reasonable tradeoff between compression, access, and verification, making it an interesting candidate for the type of balanced plan representation that we discussed earlier. Since automaton plans are strictly more expressive than macros, we can use them to represent plans compactly for a wider range of planning instances. However, this does not come at the expense of prohibitively expensive computational properties, since verification is easier for automaton plans than for HTNs as well as general STRIPS planning.

Automaton plans were first introduced in a conference paper (Bäckström, Jonsson, & Jonsson, 2012a). The present paper makes the following additional contributions:

- A formalization of automaton plans using Mealy machines, providing a stronger theoretical foundation of automaton plans in automaton theory.

- A proof that plan verification for automaton plans is $\mathbf{\Pi}_2^p$-complete, a result that is used to compare the expressive power of automaton plans to that of other compact plan representations.

- A reduction from automaton plans to HTNs, proving that HTNs are strictly more expressive than automaton plans, which comes at the price of more expensive computational properties.

The rest of the paper is organized as follows. Section 2 describes notation and basic concepts, while Section 3 introduces automaton plans. Section 4 illustrates automaton plans using several practical examples. Section 5 compares the computational properties of automaton plans to those of other compact plan representations from the literature. Section 6 describes related work, while Section 7 concludes with a discussion.

2. Notation

In this section we describe the notation used throughout the paper. We first introduce a formal definition of STRIPS planning domains based on function symbols, and show how STRIPS planning instances are induced by associating sets of objects with planning domains. The same idea is used in the PDDL language to compactly express planning domains and planning instances, and our definition can be viewed as a mathematical adaptation of PDDL.

Given a set of symbols Σ, let Σ^n denote the set of strings of length n composed of symbols in Σ. Let $x \in \Sigma^n$ be such a string. For each $1 \leq k \leq n$, we use x_k to denote the k-th symbol of x. As is customary, we use ϵ to denote the empty string, which satisfies $\epsilon x = x\epsilon = x$. Given a set of elements S, let S^* and S^+ denote sequences and non-empty sequences of elements of S, respectively. Given a sequence $\pi \in S^*$, let $|\pi|$ denote its length. For any construct X, let $||X||$ denote its size, i.e. the number of bits in its representation.

2.1 Function Symbols

A planning domain is an abstract description that can be instantiated on a given set of objects Σ to form a planning instance. In this section we introduce *function symbols* to facilitate the description of planning domains. Formally, a function symbol f has fixed arity $ar(f)$ and can be applied to any vector of objects $x \in \Sigma^{ar(f)}$ to produce a new object $f[x]$. Let F be a set of function symbols and let Σ be a set of objects. We define $F_\Sigma = \{f[x] : f \in F, x \in \Sigma^{ar(f)}\} \subseteq F \times \Sigma^*$ as the set of new objects obtained by applying each function symbol in F to each vector of objects in Σ of the appropriate arity.

Let f and g be two function symbols in F. An *argument map from f to g* is a function $\varphi : \Sigma^{ar(f)} \rightarrow \Sigma^{ar(g)}$ mapping arguments of f to arguments of g. Intuitively, as a result of applying φ to an argument $x \in \Sigma^{ar(f)}$ of f, each argument of g is either a component of x or a constant object in Σ independent of x. Formally, for each $1 \leq k \leq ar(g)$, either $\varphi_k(x) = x_j$ for a fixed index j satisfying $1 \leq j \leq ar(f)$, or $\varphi_k(x) = \sigma$ for a fixed object $\sigma \in \Sigma$. An argument map φ from f to g enables us to map an object $f[x] \in F_\Sigma$ to an object $g[\varphi(x)] \in F_\Sigma$.

Since argument maps have a restricted form, we can characterize an argument map φ from f to g using an *index string from f to g*, i.e. a string $\nu \in (\{1, \ldots, ar(f)\} \cup \Sigma)^{ar(g)}$ containing indices of f and/or objects in Σ. An index string ν from f to g induces an argument map φ from f to g such that for each $1 \leq k \leq ar(g)$, $\varphi_k(x) = x_{\nu(k)} \in \Sigma$ if $\nu(k) \in \{1, \ldots, ar(f)\}$ and $\varphi_k(x) = \nu(k) \in \Sigma$ otherwise.

To illustrate the idea, let $F = \{f, g\}$ with $ar(f) = ar(g) = 2$ and let $\Sigma = \{\sigma_1, \sigma_2\}$. An example index string from f to g is given by $\nu = 2\sigma_1$, which induces an argument map φ from f to g such that on input $x \in \Sigma^2$, the first component of $\varphi(x)$ always equals the second component of x, and the second component of $\varphi(x)$ always equals σ_1. Given φ, the object $f[\sigma_1\sigma_2] \in F_\Sigma$ maps to the object $g[\varphi(\sigma_1\sigma_2)] = g[\sigma_2\sigma_1] \in F_\Sigma$.

2.2 Planning

Let V be a set of propositional variables or fluents. A literal l is a non-negated or negated fluent, i.e. $l = v$ or $l = \overline{v}$ for some $v \in V$. Given a set of literals L, let $L_+ = \{v \in V : v \in L\}$ and $L_- = \{v \in V : \overline{v} \in L\}$ be the set of fluents that appear as non-negated or negated in L, respectively. We say that a set of literals L is *consistent* if $v \notin L$ or $\overline{v} \notin L$ for each $v \in V$, which is equivalent to $L_+ \cap L_- = \varnothing$. A *state* $s \subseteq V$ is a set of fluents that are currently true; all fluents not in s are assumed to be false. A set of literals L *holds* in a state s if $L_+ \subseteq s$ and $L_- \cap s = \varnothing$. We define an update operation on a state s and a set of literals L as $s \oplus L = (s \setminus L_-) \cup L_+$.

In this paper we focus on STRIPS planning with negative preconditions. Formally, a STRIPS planning domain is a tuple $\boldsymbol{d} = \langle P, A \rangle$, where P is a set of function symbols called *predicates* and A is a set of function symbols called *actions*. Each action $a \in A$ has an associated precondition $\mathrm{pre}(a) = \{(p_1, \varphi_1, b_1), \ldots, (p_n, \varphi_n, b_n)\}$ where, for each $1 \leq k \leq n$, p_k is a predicate in P, φ_k is an argument map from a to p_k, and b_k is a Boolean. To be well-defined, $\mathrm{pre}(a)$ should not simultaneously contain $(p, \varphi, \mathsf{true})$ and $(p, \varphi, \mathsf{false})$ for some predicate p and argument map φ from a to p. The postcondition $\mathrm{post}(a)$ of a is similarly defined.

A STRIPS planning instance is a tuple $\boldsymbol{p} = \langle P, A, \Sigma, I, G \rangle$, where $\langle P, A \rangle$ is a planning domain, Σ a set of objects, I an initial state, and G a goal state, i.e. a set of literals implicitly defining a set of states in which G holds. P and Σ implicitly define a set of fluents $P_\Sigma = \{p[x] : p \in P, x \in \Sigma^{ar(p)}\}$ by applying each predicate to each vector of objects in Σ. Likewise, A and Σ implicitly define a set of operators A_Σ. Thus fluents correspond to grounded predicates, and operators correspond to grounded actions, which is the reason we distinguish between "action" and "operator" in the text. The initial state $I \subseteq P_\Sigma$ and goal state $G \subseteq P_\Sigma$ are both subsets of (non-negated) fluents.

For each action $a \in A$ and each $x \in \Sigma^{ar(a)}$, the precondition of operator $a[x] \in A_\Sigma$ is given by $\mathrm{pre}(a[x]) = \{b_1 p_1[\varphi_1(x)], \ldots, b_n p_n[\varphi_n(x)]\}$, where $\mathrm{pre}(a) = \{(p_1, \varphi_1, b_1), \ldots, (p_n, \varphi_n, b_n)\}$, $bp[y] = p[y]$ if b is false, and $bp[y] = \overline{p}[y]$ if b is true. In other words, $\mathrm{pre}(a[x])$ is the result of applying each argument map φ_k in the precondition of a to the argument x to obtain a fluent $p_k[\varphi_k(x)] \in P_\Sigma$, which is then appropriately negated. The postcondition $\mathrm{post}(a[x])$ of $a[x]$ is similarly defined. Note that if $\mathrm{pre}(a)$ and $\mathrm{post}(a)$ are well-defined, $\mathrm{pre}(a[x])$ and $\mathrm{post}(a[x])$ are consistent sets of literals on P_Σ.

An operator $o \in A_\Sigma$ is applicable in state s if and only if $\mathrm{pre}(o)$ holds in s, and the result of applying o in s is $s \oplus \mathrm{post}(o)$. A *plan* for $\boldsymbol{p}$ is a sequence of operators $\pi = \langle o_1, \ldots, o_n \rangle \in A_\Sigma^*$ such that $\mathrm{pre}(o_1)$ holds in I and, for each $1 < k \leq n$, $\mathrm{pre}(o_k)$ holds in $I \oplus \mathrm{post}(o_1) \oplus \cdots \oplus \mathrm{post}(o_{k-1})$. We say that π *solves* $\boldsymbol{p}$ if G holds in $I \oplus \mathrm{post}(o_1) \oplus \cdots \oplus \mathrm{post}(o_n)$. Given two operator sequences π and π', let $\pi; \pi'$ denote their concatenation.

Note that $\boldsymbol{p}$ has $\sum_{p \in P} |\Sigma|^{ar(p)}$ fluents and $\sum_{a \in A} |\Sigma|^{ar(a)}$ operators, which can both be exponential in $||\boldsymbol{p}||$, the description length of $\boldsymbol{p}$. To avoid errors due to discrepancies in instance description length and actual instance size, we only consider P and A such that, for each $p \in P$ and $a \in A$, $ar(p)$ and $ar(a)$ are constants that are independent of $||\boldsymbol{p}||$. We sometimes describe planning instances directly on the form $\boldsymbol{p} = \langle P_\Sigma, A_\Sigma, \Sigma, I, G \rangle$ by defining predicates and actions of arity 0, implying that each predicate is a fluent and each action an operator.

3. Automaton Plans

In this section we define the concept of automaton plans, which is similar to macro plans. Just like a macro plan consists of a hierarchy of macros, an automaton plan consists of a hierarchy of automata. Unlike macros, the output of an automaton depends on the input, making it possible for a single automaton to represent a family of similar plans. We can either use an automaton plan to represent a single plan by explicitly specifying an input string of the root automaton, or allow parameterized plans by leaving the input string of the root automaton undefined.

3.1 Mealy Machines

To represent individual automata we use a variant of deterministic finite state automata called Mealy machines (Mealy, 1955), each defined as a tuple $M = \langle S, s_0, \Sigma, \Lambda, T, \Gamma \rangle$ where

- S is a finite set of states,
- $s_0 \in S$ is an initial state,
- Σ is an input alphabet,
- Λ is an output alphabet,
- $T : S \times \Sigma \to S$ is a transition function,
- $\Gamma : S \times \Sigma \to \Lambda$ is an output function.

A Mealy machine M is a *transducer* whose purpose it is to generate a sequence of output for a given input string. This is in contrast to *acceptors* that generate binary output (accept or reject). Executing a Mealy machine M on input $x = x_1 x_2 \cdots x_n \in \Sigma^n$ generates the output $\Gamma(s_0, x_1)\Gamma(s_1, x_2) \cdots \Gamma(s_{n-1}, x_n) \in \Lambda^n$ where $s_k = T(s_{k-1}, x_k)$ for each $1 \leq k < n$.

We extend Mealy machines to allow ϵ-transitions (i.e. transitions that do not consume input symbols in Σ). While this may in general cause Mealy machines to be non-deterministic, we include several restrictions that preserve determinism:

- We redefine T as a partial function $T : S \times (\Sigma \cup \{\epsilon\}) \to S$ such that for each $s \in S$, either $T(s, \epsilon)$ is defined or $T(s, \sigma)$ is defined for each $\sigma \in \Sigma$, but not both. This is still, in a sense, a total function on S since there is always exactly one possible transition from each state $s \in S$, but the transition may or may not consume an input symbol.
- We do not allow ϵ-cycles, i.e. there must not exist any subset $\{s_1, \ldots, s_n\} \subseteq S$ of states such that $T(s_{k-1}, \epsilon) = s_k$ for each $1 < k \leq n$ and $T(s_n, \epsilon) = s_1$.
- We further require that ϵ-transitions must always fire, in order to make the behavior of Mealy machines well defined also when all input symbols have been consumed.

We also allow ϵ as output, i.e. a transition may or may not generate an output symbol in Λ. To allow for ϵ-transitions and ϵ-output we redefine Γ as a partial function $\Gamma : S \times (\Sigma \cup \{\epsilon\}) \to \Lambda \cup \{\epsilon\}$. The definition of Γ should be consistent with T, i.e. for each state $s \in S$, if $T(s, \epsilon)$ is defined then $\Gamma(s, \epsilon)$ is defined, else $\Gamma(s, \sigma)$ is defined for each $\sigma \in \Sigma$. We define an extended output function $\Gamma^* : S \times \Sigma^* \to \Lambda^*$ such that for each state $s \in S$, input symbol $\sigma \in \Sigma$ and input string $x \in \Sigma^*$,

$$\Gamma^*(s, \epsilon) = \begin{cases} \Gamma(s, \epsilon)\Gamma^*(T(s, \epsilon), \epsilon), & \text{if } T(s, \epsilon) \text{ is defined,} \\ \epsilon, & \text{otherwise,} \end{cases}$$

$$\Gamma^*(s, \sigma x) = \begin{cases} \Gamma(s, \epsilon)\Gamma^*(T(s, \epsilon), \sigma x), & \text{if } T(s, \epsilon) \text{ is defined,} \\ \Gamma(s, \sigma)\Gamma^*(T(s, \sigma), x), & \text{otherwise.} \end{cases}$$

The deterministic output of a Mealy machine M on input x is given by $\Gamma^*(s_0, x) \in \Lambda^*$.

As is customary we use graphs to represent automata. The graph associated with a Mealy machine M has one node per state in S. An edge between states s and t with label i/o, where $i \in (\Sigma \cup \{\epsilon\})$ and $o \in (\Lambda \cup \{\epsilon\})$, implies that $T(s, i) = t$ and $\Gamma(s, i) = o$. To simplify the graphs we adopt the following conventions:

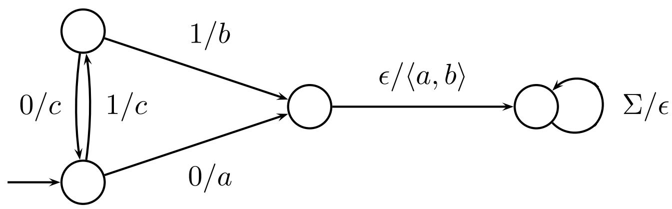

Figure 1: An example Mealy machine.

- An edge between states s and t with label $i/\langle\lambda_1, \ldots, \lambda_n\rangle \in (\Sigma \cup \{\epsilon\}) \times \Lambda^n$ is used as shorthand to describe a series of intermediate states $s'_2, \ldots, s'_n$ such that $T(s, i) = s'_2$, $T(s'_{k-1}, \epsilon) = s'_k$ for each $2 < k \leq n$, $T(s'_n, \epsilon) = t$, $\Gamma(s, i) = \lambda_1$, and $\Gamma(s'_k, \epsilon) = \lambda_k$ for each $2 \leq k \leq n$.

- An edge between states s and t with label Σ^n/ϵ is used as shorthand to describe a series of intermediate states $s'_2, \ldots, s'_n$ such that for each $\sigma \in \Sigma$, $T(s, \sigma) = s'_2$, $T(s'_{k-1}, \sigma) = s'_k$ for each $2 < k \leq n$, $T(s'_n, \sigma) = t$, $\Gamma(s, \sigma) = \epsilon$, and $\Gamma(s'_k, \sigma) = \epsilon$ for each $2 \leq k \leq n$.

Figure 1 shows an example Mealy machine $M = \langle S, s_0, \Sigma, \Lambda, T, \Gamma\rangle$ with $|S| = 4$, $\Sigma = \{0, 1\}$, and $\Lambda = \{a, b, c\}$. The initial state s_0 is identified by an incoming edge without origin. Two example outputs are $\Gamma^*(s_0, 01) = aab$ and $\Gamma^*(s_0, 1011) = cccbab$.

3.2 Automaton Hierarchies

In this section we explain how to construct hierarchies of automata in order to represent plans. Each automaton hierarchy is associated with a STRIPS planning instance $\boldsymbol{p} = \langle P, A, \Sigma, I, G\rangle$. We define a set Au of function symbols called automata, i.e. each automaton $M \in Au$ has fixed arity $ar(M)$, something that is unusual in automaton theory. The motivation is that the automata used to represent plans can be viewed as abstract actions, and the input to each automaton serves a dual purpose: to determine how to fire the transitions of the automaton in order to generate the output, and to copy input symbols onto actions and other automata.

Each automaton $M \in Au$ corresponds to a Mealy machine $\langle S_M, s_0, \Sigma, \Lambda_M, T_M, \Gamma_M\rangle$, where Σ is the set of objects of the STRIPS instance $\boldsymbol{p}$ and $\Lambda_M = (A \cup Au) \times (\{1, \ldots, ar(M)\} \cup \Sigma)^*$. Each output symbol $(u, \nu) \in \Lambda_M$ is a pair consisting of an action $u \in A$ or automaton $u \in Au$ and an index string ν from M to u. For each input string $x \in \Sigma^{ar(M)}$, we require the output of automaton M on input x to be non-empty, i.e. $\Gamma^*_M(s_0, x) = \langle(u_1, \nu_1), \ldots, (u_n, \nu_n)\rangle \in \Lambda^+_M$.

Given Au, we define an *expansion graph* that denotes dependencies among the automata in Au:

Definition 1. *Given a set Au of automata, the expansion graph $G_{Au} = \langle Au, \prec\rangle$ is a directed graph over automata where, for each pair $M, M' \in Au$, $M \prec M'$ if and only if there exists a state-input pair $(s, \sigma) \in S_M \times \Sigma$ of M such that $\Gamma_M(s, \sigma) = (M', \nu)$ for some index string ν.*

Thus there is an edge between automata M and M' if and only if M' appears as an output of M.

We next define an *automaton hierarchy* as a tuple $H = \langle \Sigma, A, Au, r\rangle$ where

- Σ, A, and Au are defined as above,

- G_{Au} is acyclic and weakly connected,

- $r \in Au$ is the root automaton, and
- there exists a directed path in G_{Au} from r to each other automaton in Au.

For each $M \in Au$, let $Succ(M) = \{M' \in Au : M \prec M'\}$ be the set of successors of M. The height $h(M)$ of M is the length of the longest directed path from M to any other node in G_{Au}, i.e.

$$h(M) = \begin{cases} 0, & \text{if } Succ(M) = \varnothing, \\ 1 + \max_{M' \in Succ(M)} h(M'), & \text{otherwise.} \end{cases}$$

Given an automaton hierarchy H, let $\mathcal{S} = \max_{M \in Au} |S_M|$ be the maximum number of states of the Mealy machine representation of each automaton, and let $Ar = 1 + \max_{u \in A \cup Au} ar(u)$ be the maximum arity of actions and automata, plus one.

The aim of an automaton $M \in Au$ is not only to generate the output $\Gamma_M^*(s_0, x)$ on input x, but to define a decomposition strategy. This requires us to process the output of M in a concrete way described below. An alternative would have been to integrate this processing step into the automata, but this would no longer correspond to the well-established definition of Mealy machines.

We first define a notion of grounded automata, analogous to the notion of grounded actions (i.e. operators) of a planning instance. An *automaton call* $M[x]$ is an automaton M and associated input string $x \in \Sigma^{ar(M)}$, representing that M is called on input x. The sets Au and Σ define a set of automaton calls $Au_\Sigma = \{M[x] : M \in Au, x \in \Sigma^{ar(M)}\}$, i.e. automata paired with all input strings of the appropriate arity. We next define a function Apply that acts as a decomposition strategy:

Definition 2. *Let* $\text{Apply} : Au_\Sigma \to (A_\Sigma \cup Au_\Sigma)^+$ *be a function such that for* $M[x] \in Au_\Sigma$, $\text{Apply}(M[x]) = \langle u_1[\varphi_1(x)], \ldots, u_n[\varphi_n(x)]\rangle$, *where* $\Gamma_M^*(s_0, x) = \langle (u_1, \nu_1), \ldots, (u_n, \nu_n)\rangle$ *and, for each* $1 \le k \le n$, φ_k *is the argument map from* M *to* u_k *induced by the index string* ν_k.

The purpose of Apply is to replace an automaton call with a sequence of operators and other automaton calls. Recursively applying this decomposition strategy should eventually result in a sequence consisting exclusively of operators. We show that Apply can always be computed in polynomial time in the size of an automaton.

Lemma 3. *For each automaton call* $M[x] \in Au_\Sigma$, *the complexity of computing* $\text{Apply}(M[x])$ *is bounded by* $\mathcal{S} \cdot Ar^2$.

Proof. Our definition of Mealy machines requires each cycle to consume at least one input symbol. In the worst case, we can fire $|S_M| - 1$ ϵ-transitions followed by a transition that consumes an input symbol. Since the input string x has exactly $ar(M)$ symbols, the total number of transitions is bounded by $(|S_M| - 1)(1 + ar(M)) + ar(M) \le |S_M| \cdot (1 + ar(M)) \le \mathcal{S} \cdot Ar$.

Let $\langle (u_1, \nu_1), \ldots, (u_n, \nu_n)\rangle$ be the output of M on input x. For each $1 \le k \le n$, u_k is a single symbol, while ν_k contains at most $Ar - 1$ symbols. Applying the argument map φ_k induced by ν_k to the input string x is linear in $|\nu_k| \le Ar$. Thus computing $\text{Apply}(M[x]) = \langle u_1[\varphi_1(x)], \ldots, u_n[\varphi_n(x)]\rangle$ requires at most Ar time and space for each element, and since $n \le \mathcal{S} \cdot Ar$, the total complexity is bounded by $\mathcal{S} \cdot Ar^2$. □

To represent the result of recursively applying the decomposition strategy, we define an *expansion function* Exp on automaton calls and operators:

Definition 4. *Let* Exp *be a function on* $(A_\Sigma \cup Au_\Sigma)^+$ *defined as follows:*

1. $\mathrm{Exp}(a[x]) = \langle a[x] \rangle$ *if* $a[x] \in A_\Sigma$*,*

2. $\mathrm{Exp}(M[x]) = \mathrm{Exp}(\mathrm{Apply}(M[x]))$ *if* $M[x] \in Au_\Sigma$*,*

3. $\mathrm{Exp}(\langle u_1[y_1], \ldots, u_n[y_n] \rangle) = \mathrm{Exp}(u_1[y_1]); \ldots ; \mathrm{Exp}(u_n[y_n])$.

In the following lemma we prove that the expansion of any automaton call is a sequence of operators.

Lemma 5. *For each automaton call* $M[x] \in Au_\Sigma$*,* $\mathrm{Exp}(M[x]) \in A_\Sigma^+$*.*

Proof. We prove the lemma by induction over $h(M)$. If $h(M) = 0$, $\mathrm{Apply}(M[x])$ is a sequence of operators $\langle a_1[x_1], \ldots, a_n[x_n] \rangle \in A_\Sigma^+$, implying

$$\begin{aligned} \mathrm{Exp}(M[x]) &= \mathrm{Exp}(\mathrm{Apply}(M[x])) = \mathrm{Exp}(\langle a_1[x_1], \ldots, a_n[x_n] \rangle) = \\ &= \mathrm{Exp}(a_1[x_1]); \ldots ; \mathrm{Exp}(a_n[x_n]) = \langle a_1[x_1] \rangle; \ldots ; \langle a_n[x_n] \rangle = \\ &= \langle a_1[x_1], \ldots, a_n[x_n] \rangle \in A_\Sigma^+. \end{aligned}$$

We next prove the inductive step $h(M) > 0$. In this case, $\mathrm{Apply}(M[x])$ is a sequence of operators and automaton calls $\langle u_1[y_1], \ldots, u_n[y_n] \rangle \in (A_\Sigma \cup Au_\Sigma)^+$, implying

$$\begin{aligned} \mathrm{Exp}(M[x]) &= \mathrm{Exp}(\mathrm{Apply}(M[x])) = \mathrm{Exp}(\langle u_1[x_1], \ldots, u_n[x_n] \rangle) = \\ &= \mathrm{Exp}(u_1[x_1]); \cdots ; \mathrm{Exp}(u_n[x_n]). \end{aligned}$$

For each $1 \leq k \leq n$, if $u_k[x_k]$ is an operator we have $\mathrm{Exp}(u_k[x_k]) = \langle u_k[x_k] \rangle \in A_\Sigma^+$. On the other hand, if $u_k[x_k]$ is an automaton call, $\mathrm{Exp}(u_k[x_k]) \in A_\Sigma^+$ by hypothesis of induction since $h(u_k) < h(M)$. Thus $\mathrm{Exp}(M[x])$ is the concatenation of several operator sequences in A_Σ^+, which is itself an operator sequence in A_Σ^+. □

Note that the proof depends on the fact that the expansion graph G_{Au} is acyclic, since otherwise the height $h(M)$ of automaton M is ill-defined. We also prove an upper bound on the length of the operator sequence $\mathrm{Exp}(M[x])$.

Lemma 6. *For each automaton call* $M[x] \in Au_\Sigma$*,* $|\mathrm{Exp}(M[x])| \leq (\mathcal{S} \cdot Ar)^{1+h(M)}$*.*

Proof. By induction over $h(M)$. If $h(M) = 0$, $\mathrm{Apply}(M[x]) = \langle a_1[x_1], \ldots, a_n[x_n] \rangle$ is a sequence of operators in A_Σ, implying $\mathrm{Exp}(M[x]) = \mathrm{Apply}(M[x]) = \langle a_1[x_1], \ldots, a_n[x_n] \rangle$. It follows that $|\mathrm{Exp}(M[x])| \leq (\mathcal{S} \cdot Ar)^{1+0}$ since $n \leq \mathcal{S} \cdot Ar$.

If $h(M) > 0$, $\mathrm{Apply}(M[x]) = \langle u_1[x_1], \ldots, u_n[x_n] \rangle$ is a sequence of operators and automaton calls, implying $\mathrm{Exp}(M[x]) = \mathrm{Exp}(u_1[x_1]); \cdots ; \mathrm{Exp}(u_n[x_n])$. For each $1 \leq k \leq n$, if $u_k \in A$ then $|\mathrm{Exp}(u_k[x_k])| = 1$, else $|\mathrm{Exp}(u_k[x_k])| \leq (\mathcal{S} \cdot Ar)^{h(M)}$ by hypothesis of induction since $h(u_k) < h(M)$. It follows that $|\mathrm{Exp}(M[x])| \leq (\mathcal{S} \cdot Ar)^{1+h(M)}$ since $n \leq \mathcal{S} \cdot Ar$. □

An *automaton plan* is a tuple $\rho = \langle \Sigma, A, Au, r, x \rangle$ where $\langle \Sigma, A, Au, r \rangle$ is an automaton hierarchy and $x \in \Sigma^{ar(r)}$ is an input string to the root automaton r. While automaton hierarchies represent families of plans, an automaton plan represents a unique plan given by $\pi = \mathrm{Exp}(r[x]) \in A_\Sigma^+$.

In subsequent sections we exploit the notion of *uniform expansion*, defined as follows:

Definition 7. *An automaton hierarchy* $\langle \Sigma, A, Au, r \rangle$ *has* uniform expansion *if and only if for each automaton* $M \in Au$ *there exists a number* ℓ_M *such that* $|\mathrm{Exp}(M[x])| = \ell_M$ *for each* $x \in \Sigma^{ar(M)}$*.*

In other words, expanding an automaton call $M[x]$ always results in an operator sequence of length exactly ℓ_M, regardless of the input x.

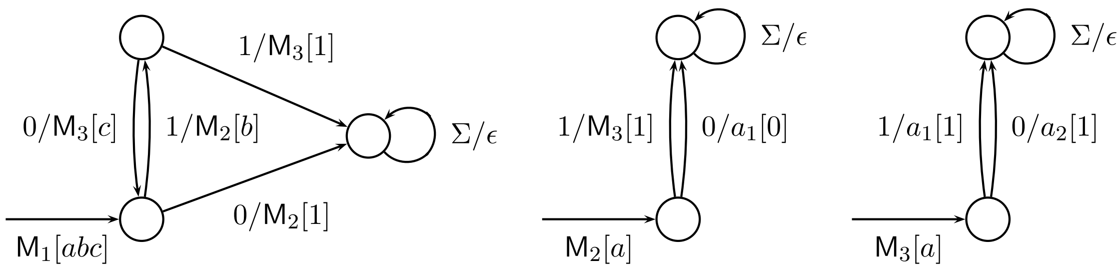

Figure 2: The three automata in the simple example.

4. Examples of Automaton Plans

In this section we present several examples of automaton plans. Our aim is first and foremost to illustrate the concept of automaton plans. However, we also use the examples to illuminate several interesting properties of automaton plans. For example, we can use small automaton plans to represent exponentially long plans.

We begin by showing an example of a relatively simple automaton plan on two symbols, two actions, and three automata, defined as $\rho = \langle\{0,1\}, \{a_1, a_2\}, \{\mathsf{M}_1, \mathsf{M}_2, \mathsf{M}_3\}, \mathsf{M}_1, 100\rangle$. Actions a_1 and a_2 both have arity 1, and Figure 2 shows the three automata M_1, M_2, and M_3 (with arity 3, 1, and 1, respectively). In the figure, the edge without origin points to the initial state of the automaton, and the label on this edge contains the name and input string of the automaton.

To simplify the description of index strings and argument maps we assign explicit names (a, b, and c) to each symbol of the input string of an automaton. Each argument map is described as a string of input symbol names and symbols from $\Sigma = \{0,1\}$. For example, the label $\mathsf{M}_2[b]$ in automaton M_1 corresponds to the output symbol $(\mathsf{M}_2, 2)$, i.e. the index string from M_1 to M_2 assigns the second input symbol (b) of M_1 to the lone input symbol of M_2. Recall that the symbols of the input string have two separate functions: to decide which edges of the automaton to transition along, and to propagate information by copying symbols onto actions and other automata.

The plan π represented by ρ is given by

$$\begin{aligned}
\pi &= \mathrm{Exp}(\mathsf{M}_1[100]) = \mathrm{Exp}(\mathrm{Apply}(\mathsf{M}_1[100])) = \mathrm{Exp}(\langle \mathsf{M}_2[0], \mathsf{M}_3[0], \mathsf{M}_2[1]\rangle) = \\
&= \mathrm{Exp}(\mathsf{M}_2[0]); \mathrm{Exp}(\mathsf{M}_3[0]); \mathrm{Exp}(\mathsf{M}_2[1]) = \\
&= \mathrm{Exp}(\mathrm{Apply}(\mathsf{M}_2[0])); \mathrm{Exp}(\mathrm{Apply}(\mathsf{M}_3[0])); \mathrm{Exp}(\mathrm{Apply}(\mathsf{M}_2[1])) = \\
&= \mathrm{Exp}(\langle a_1[0]\rangle); \mathrm{Exp}(\langle a_2[1]\rangle); \mathrm{Exp}(\langle \mathsf{M}_3[1]\rangle) = \mathrm{Exp}(a_1[0]); \mathrm{Exp}(a_2[1]); \mathrm{Exp}(\mathsf{M}_3[1]) = \\
&= \langle a_1[0]\rangle; \langle a_2[1]\rangle; \mathrm{Exp}(\mathrm{Apply}(\mathsf{M}_3[1])) = \langle a_1[0]\rangle; \langle a_2[1]\rangle; \mathrm{Exp}(\langle a_1[1]\rangle) = \\
&= \langle a_1[0]\rangle; \langle a_2[1]\rangle; \mathrm{Exp}(a_1[1]) = \langle a_1[0]\rangle; \langle a_2[1]\rangle; \langle a_1[1]\rangle = \langle a_1[0], a_2[1], a_1[1]\rangle.
\end{aligned}$$

Selecting another root automaton call would result in a different operator sequence. For example, the root $\mathsf{M}_1[000]$ would result in the sequence $\mathrm{Exp}(\mathsf{M}_1[000]) = \langle a_1[1]\rangle$, and the root $\mathsf{M}_1[101]$ would result in $\mathrm{Exp}(\mathsf{M}_1[101]) = \langle a_1[0], a_1[1], a_1[0]\rangle$.

We next show that just like macro plans, automaton plans can compactly represent plans that are exponentially long. Figure 3 shows an automaton M_n for moving n discs from peg a to peg b via peg c in Towers of Hanoi. In the figure, $\mathsf{A}_n[ab]$ is the action for moving disc n from a to b. For $n = 1$ the edge label should be $\epsilon/\langle \mathsf{A}_1[ab]\rangle$. It is not hard to show that the automaton plan $\mu = \langle\{1,2,3\}, \{\mathsf{A}_1, \ldots, \mathsf{A}_N\}, \{\mathsf{M}_1, \ldots, \mathsf{M}_N\}, \mathsf{M}_N, 132\rangle$ is a plan for the Towers of Hanoi instance

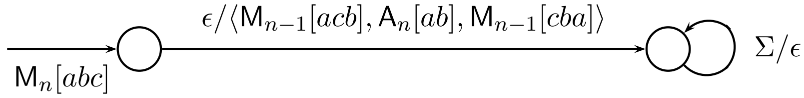

Figure 3: The automaton M_n in the automaton plan for Towers of Hanoi.

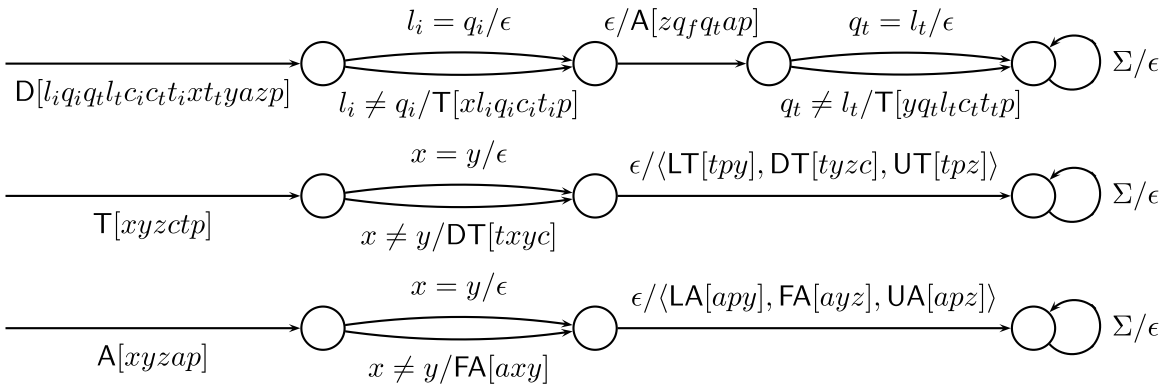

Figure 4: The automata D for delivering a package and T, A for moving a package using a truck/airplane.

with N discs. Unlike macro solutions for Towers of Hanoi (Jonsson, 2009), the automaton plan has a single automaton per disc, which is possible because of parameterization.

The ability to parameterize automata also makes it possible to represent other types of plans compactly. Figure 4 shows three automata D, T, and A that can be combined to construct an automaton plan for any instance of the LOGISTICS domain. The set of symbols Σ contains the objects of the instance: packages, airplanes, trucks, cities, and locations. The automaton T moves a package using a truck, and the input string $xyzctp$ consists of three locations x, y, and z, a city c, a truck t, and a package p. Initially, truck t is at x, package p is at y, and the destination of package p is z. The actions DT, LT, and UT stand for DriveTruck, LoadTruck, and UnloadTruck, respectively.

Automaton T assumes that locations y and z are different, else there is nothing to be done and the automaton outputs the empty string, violating the definition of automaton plans. On the other hand, locations x and y may be the same, and the automaton checks whether they are equal. Only when x and y are different is it necessary to first drive truck t from x to y. We use the notation $x = y$ and $x \neq y$ as shorthand to denote that there are $|L|$ intermediate notes, one per location, such that the automaton transitions to a distinct intermediate node for each assignment of a location to x. From each intermediate node there are $|L|$ edges to the next node: $|L| - 1$ of these edges correspond to $x \neq y$ and only one edge corresponds to $x = y$.

Once the truck is at location y, the operator sequence output by T loads package p in t, drives t to the destination, and unloads p from t. Automaton A for moving a package using an airplane is similarly defined on actions FA (FlyAirplane), LA (LoadAirplane), and UA (UnloadAirplane). Automaton D delivers a package from its current location to its destination, and the input string $l_iq_iq_tl_tc_ic_tt_ixt_tyazp$ consists of the initial location l_i of the package, intermediate airports q_i and

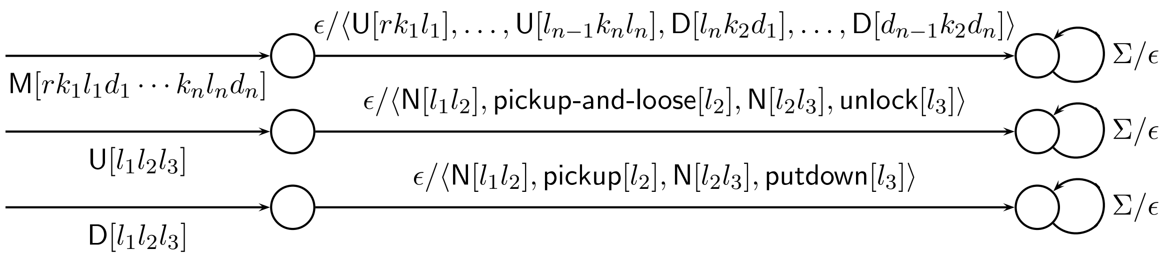

Figure 5: The automata in an automaton plan for the GRID domain.

q_t, the target location l_t, the initial and target cities c_i and c_t, a truck t_i in city c_i initially at x, a truck t_t in city c_t initially at y, an airplane a initially at z, and the package p itself. Automaton D assumes that cities c_i and c_t are different, else we could use automaton T to transport the package using a truck within a city. However, locations l_i and q_i may be equal, as well as q_t and l_t, and the automaton only moves the package using a truck whenever necessary.

We also show an example automaton plan for the GRID domain, in which a robot has to deliver keys to locations, some of which are locked. The keys are distributed at initial locations, and the robot can only carry one key at a time. The actions are to move the robot to a neighboring location, pick up a key (possibly loosing the key currently held), put down a key, and unlock a location.

Figure 5 shows an automaton plan for an instance of GRID. The root automaton M takes as input the current location of the robot (r) and three locations k_i, l_i, and d_i for each key, where k_i is the current location of the key, l_i is the associated locked location, and d_i is the destination. The plan works by first unlocking all locations in a prespecified order (which must exist for the problem to be solvable) and then delivering all keys to their destination.

The automaton U takes three locations: the location of the robot (l_1), the location of the key (l_2), and the location to be unlocked (l_3). The decomposition navigates to the key, picks it up, navigates to the location, and unlocks it. Delivering a key works in a similar way. For simplicity some parameters of actions have been omitted, and a few modifications are necessary: the first time U is applied there is no key to loose, and the first time D is applied there is a key to loose.

The automaton N (not shown) navigates between pairs of locations l_1 and l_2. Since automaton plans cannot keep track of the state, N has to include one automaton state for each possible input (l_1, l_2) (alternatively we can define a separate automaton for each destination l_2). Note that the automaton M can be used to represent solutions to different instances on the same set of locations.

5. Relationship to Other Compact Plan Representations

In this section we compare and contrast automaton plans to other compact plan representations from the literature: macro plans, HTNs (Erol et al., 1996), as well as CRARs and CSARs (Bäckström & Jonsson, 2012). Informally, CRARs and CSARs are theoretical concepts describing any compact representation of a plan that admits efficient random access (CRAR) or sequential access (CSAR) to the operators of the plan. To compare plan representations we use the following subsumption relation (Bäckström et al., 2012b):

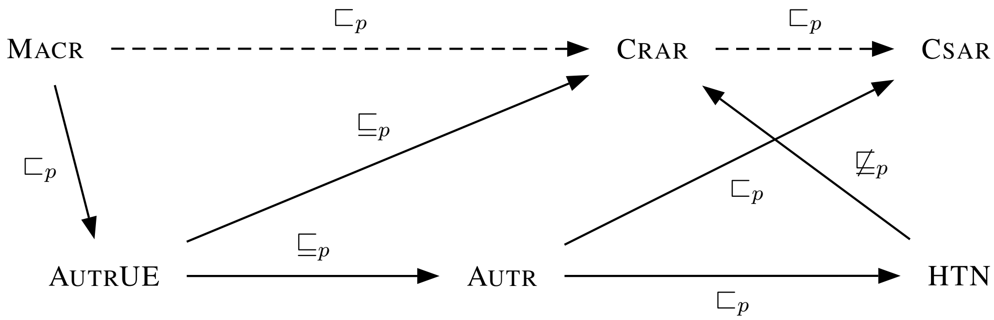

Figure 6: Summary of subsumption results, with dashed edges marking previously known results.

Definition 8. *Let X and Y be two plan representations. Then Y is at least as expressive as X, which we denote $X \sqsubseteq_p Y$, if there is a polynomial-time function g such that for each* STRIPS *planning instance **p** and each plan π for **p**, if ρ is an X representation of π, $g(\rho)$ is a Y representation of π.*

Figure 6 summarizes the subsumption results for the different plan representations considered in this paper. MACR and AUTR refer to macro plans and automaton plans, respectively, while AUTRUE refer to automaton plans with uniform expansion. Previously known results are shown using dashed edges; the remaining results are proven in this section. We use the notation $X \sqsubset_p Y$ to indicate that $X \sqsubseteq_p Y$ and $Y \not\sqsubseteq_p X$. From the figure we see that automaton plans are strictly more expressive than macro plans, but strictly less expressive than CSARs and HTNs. In the case of CRARs, we only prove a partial result: that automaton plans with uniform expansion can be translated to CRARs.

In the rest of this section, we first show that automaton plans with uniform expansion can be efficiently transformed to CRARs. We then prove that plan verification for automaton plans is $\mathbf{\Pi}_2^p$-complete. We use this latter result to prove separation between macro plans and automaton plans, and between automaton plans and CSARs/HTNs. When proving that $X \sqsubseteq_p Y$ holds for two representations X and Y, we assume that the size of the X representation is polynomial in $||\boldsymbol{p}||$, the description size of the planning instance. Trivially, automaton plans with uniform expansion are also automaton plans, while it is unknown whether general automaton plans can be efficiently transformed such that they have uniform expansion.

5.1 Automaton Plans and CRARs

In this section we show that automaton plans with uniform expansion can be efficiently translated to CRARs, i.e. compact plan representations that admit efficient random access. We first define CRARs and then describe an algorithm for transforming each automaton plan with uniform expansion to a corresponding CRAR.

Definition 9. *Let p be a polynomial function. Given a* STRIPS *planning instance $\boldsymbol{p} = \langle P, A, \Sigma, I, G\rangle$ with associated plan π, a p-*CRAR *of π is a representation ρ such that $||\rho|| \leq p(||\boldsymbol{p}||)$ and ρ outputs the k-th element of π in time and space $p(||\boldsymbol{p}||)$ for each $1 \leq k \leq |\pi|$.*

Note that for ρ to be a p-CRAR, the polynomial function p has to be independent of the planning instance $\boldsymbol{p}$, else we can always find a constant for each individual instance $\boldsymbol{p}$ such that the size of any representation for $\boldsymbol{p}$ is bounded by this constant.

```
1  function Find(k, u[x])
2      if u[x] ∈ A_Σ return u[x]
3      else
4          ⟨u_1[x_1], ..., u_n[x_n]⟩ ← Apply(u[x])
5          s ← 0, j ← 1
6          while s + ℓ(u_j) < k do
7              s ← s + ℓ(u_j), j ← j + 1
8          return Find(k − s, u_j[x_j])
```

Figure 7: Algorithm for using an automaton plan as a CRAR.

Theorem 1. AUTRUE $\sqsubseteq_p$ CRAR.

Proof. To prove the theorem we show that for each STRIPS planning instance $\boldsymbol{p}$ and each automaton plan $\rho = \langle \Sigma, A, Au, r, x\rangle$ with uniform expansion representing a plan π for $\boldsymbol{p}$, we can efficiently construct a corresponding p-CRAR for π with $p(||\boldsymbol{p}||) = (Ar + (\log \mathcal{S} + \log Ar)|Au|) \cdot \mathcal{S} \cdot Ar \cdot |Au|$.

Since ρ has uniform expansion there exist numbers ℓ_M, $M \in Au$, such that $|\text{Exp}(M[x])| = \ell_M$ for each $x \in \Sigma^{ar(M)}$. The numbers ℓ_M can be computed bottom up as follows. Traverse the automata of the expansion graph G_{Au} in reverse topological order. For each $M \in Au$, pick an input string $x \in \Sigma^{ar(M)}$ at random and compute $\text{Apply}(M[x]) = \langle u_1[y_1], \ldots, u_n[y_n]\rangle$. The number ℓ_M is given by $\ell_M = \ell_{u_1} + \ldots + \ell_{u_n}$ where, for each $1 \leq k \leq n$, $\ell_{u_k} = 1$ if $u_k \in A$ and ℓ_{u_k} has already been computed if $u_k \in Au$ since by definition, u_k comes after M in any topological ordering.

Because of Lemma 3, the total complexity of computing $\text{Apply}(M[x])$ for all $M \in Au$ is $\mathcal{S} \cdot Ar^2 \cdot |Au|$. Due to Lemma 6, $\ell_M \leq (\mathcal{S} \cdot Ar)^{1+h(M)} \leq (\mathcal{S} \cdot Ar)^{|Au|}$ for each $M \in Au$. Since $(\mathcal{S} \cdot Ar)^{|Au|} = 2^{(\log \mathcal{S} + \log Ar)|Au|}$, we need at most $(\log \mathcal{S} + \log Ar)|Au|$ bits to represent ℓ_M, and computing ℓ_M requires at most $(\log \mathcal{S} + \log Ar) \cdot \mathcal{S} \cdot Ar \cdot |Au|$ operations. Repeating this computation for each $M \in Au$ gives us a complexity bound of $(Ar + (\log \mathcal{S} + \log Ar)|Au|) \cdot \mathcal{S} \cdot Ar \cdot |Au|$.

We prove that the recursive algorithm Find in Figure 7 has the following properties, by induction over the number of recursive calls:

1. for each $M[x] \in Au_\Sigma$ such that $\text{Exp}(M[x]) = \langle a_1[x_1], \ldots, a_n[x_n]\rangle$, Find($k$, $M[x]$) returns operator $a_k[x_k]$ for $1 \leq k \leq n$, and
2. for each $a[x] \in A_\Sigma$, Find(k, $a[x]$) returns $a[x]$.

Basis: If Find(k, $u[x]$) does not call itself recursively, then $u[x]$ must be an operator. By definition, $\text{Exp}(u[x]) = u[x]$ since $u[x] \in A_\Sigma$.

Induction step: Suppose the claim holds when Find makes at most m recursive calls for some $m \geq 0$. Assume Find(k, $u[x]$) makes $m+1$ recursive calls. Let $\langle u_1[x_1], \ldots, u_n[x_n]\rangle = \text{Apply}(u[x])$ and, for each $1 \leq k \leq n$, $\ell(u_k) = 1$ if $u_k[x_k] \in A_\Sigma$ and $\ell(u_k) = \ell_{u_k}$ if $u_k[x_k] \in Au_\Sigma$. Lines 5–7 computes s and j such that either

1. $j = 1$, $s = 0$ and $k \leq \ell(u_1)$ or
2. $j > 1$, $s = \ell(u_1) + \ldots + \ell(u_{j-1}) < k \leq \ell(u_1) + \ldots + \ell(u_j)$.

By definition, $\text{Exp}(u[x]) = \text{Exp}(u_1[x_1]); \ldots; \text{Exp}(u_n[x_n])$, implying that operator k in $\text{Exp}(u[x])$ is operator $k - s$ in $\text{Exp}(u_j[x_j])$. It follows from the induction hypothesis that the recursive call Find($k - s$, $u_j[x_j]$) returns this operator.

To prove the complexity of Find, note that Find calls itself recursively at most once for each $M \in Au$ since G_{Au} is acyclic. Moreover, the complexity of computing $\text{Apply}(M[x])$ is bounded by $\mathcal{S} \cdot Ar^2$, and the while loop on lines 6–7 runs at most $n \leq \mathcal{S} \cdot Ar$ times, each time performing at most $(\log \mathcal{S} + \log Ar)|Au|$ operations to update the value of s. We have thus showed that the automaton plan ρ together with the procedure Find and the values ℓ_M, $M \in Au$, constitute a p-CRAR for π with $p(||\boldsymbol{p}||) = (Ar + (\log \mathcal{S} + \log Ar)|Au|) \cdot \mathcal{S} \cdot Ar \cdot |Au|$. □

5.2 Verification of Automaton Plans

In this section we show that the problem of plan verification for automaton plans is $\mathbf{\Pi}_2^p$-complete. We first prove membership by reducing plan verification for automaton plans to plan verification for CRARs, which is known to be $\mathbf{\Pi}_2^p$-complete (Bäckström et al., 2012b). We then prove hardness by reducing $\forall\exists$-SAT, which is also $\mathbf{\Pi}_2^p$-complete, to plan verification for automaton plans with uniform expansion. The complexity result for plan verification is later used to separate automaton plans from macro plans, CSARs, and HTNs, but we do not obtain a similar separation result between automaton plans and CRARs since the complexity of plan verification is the same.

To prove membership we first define an alternative expansion function Exp' that pads the original plan with dummy operators until the expansion of each automaton has the same length for each accepting input. Intuitively, even though the original automaton plan need not have uniform expansion, the alternative expansion function Exp' emulates an automaton plan that does. Note that this is not sufficient to prove that we can transform any automaton plan to a p-CRAR, since operators have different indices in the plans represented by the two expansion functions Exp and Exp'.

Let $\boldsymbol{p}$ be a planning instance, and let $\rho = \langle \Sigma, A, Au, r, x \rangle$ be an automaton plan representing a solution π to $\boldsymbol{p}$. For each automaton $M \in Au$, let $I_M = (\mathcal{S} \cdot Ar)^{1+h(M)}$ be the upper bound on $|\text{Exp}(M[x])|$ from Lemma 6. Let $\delta = \langle \varnothing, \varnothing \rangle$ be a parameter-free dummy operator with empty pre- and postcondition, and add δ to A_Σ. Define δ^k, $k > 0$, as a sequence containing k copies of δ. We define an alternative expansion function Exp' on $(A_\Sigma \cup Au_\Sigma)^+$ as follows:

1. $\text{Exp}'(a[x]) = \langle a[x] \rangle$ if $a[x] \in A_\Sigma$,
2. $\text{Exp}'(M[x]) = \text{Exp}'(\text{Apply}(M[x])); \delta^L$ if $M[x] \in Au_\Sigma$, where the length of δ^L is $L = I_M - |\text{Exp}'(\text{Apply}(M[x]))|$,
3. $\text{Exp}'(\langle u_1[y_1], \ldots, u_n[y_n] \rangle) = \text{Exp}'(u_1[y_1]); \ldots ; \text{Exp}'(u_n[y_n])$.

The only difference with respect to the original expansion function Exp is that the alternative expansion function Exp' appends a sequence δ^L of dummy operators to the result of $\text{Exp}'(\text{Apply}(M[x]))$, causing $\text{Exp}'(M[x])$ to have length exactly I_M.

In the following lemma we prove that the operator sequence output by the alternative expansion function Exp' is equivalent to the operator sequence output by the original expansion function Exp.

Lemma 10. *For each automaton call* $M[x] \in Au_\Sigma$, $\text{Exp}'(M[x]) \in A_\Sigma^{I_M}$, *and applying* $\text{Exp}(M[x])$ *and* $\text{Exp}'(M[x])$ *to any state* s *either is not possible or results in the same state.*

Proof. We prove the lemma by induction over $|Au|$. The base case is given by $|Au| = 1$. In this case, since G_{Au} is acyclic, $\text{Apply}(M[x])$ is a sequence of operators $\langle a_1[x_1], \ldots, a_n[x_n] \rangle \in A_\Sigma^+$, implying

$$\begin{aligned}\mathrm{Exp}(M[x]) &= \langle a_1[x_1], \ldots, a_n[x_n]\rangle,\\ \mathrm{Exp}'(M[x]) &= \langle a_1[x_1], \ldots, a_n[x_n]\rangle; \delta^L,\end{aligned}$$

where $L = I_M - n$. Thus $\mathrm{Exp}'(M[x]) \in A_\Sigma^{I_M}$, and applying $\mathrm{Exp}'(M[x])$ in a state s has the same effect as applying $\mathrm{Exp}(M[x])$ since the dummy operator δ is always applicable and has no effect.

We next prove the inductive step $|Au| > 1$. In this case, $\mathrm{Apply}(M[x])$ is a sequence of operators and automaton calls $\langle u_1[y_1], \ldots, u_n[y_n]\rangle \in (A_\Sigma \cup Au_\Sigma)^+$, implying

$$\begin{aligned}\mathrm{Exp}(M[x]) &= \mathrm{Exp}(u_1[x_1]); \cdots; \mathrm{Exp}(u_n[x_n]),\\ \mathrm{Exp}'(M[x]) &= \mathrm{Exp}'(u_1[x_1]); \cdots; \mathrm{Exp}'(u_n[x_n]); \delta^L,\end{aligned}$$

where δ^L contains enough copies of δ to make $|\mathrm{Exp}'(M[x])| = I_M$. For each $1 \le k \le n$, if $u_k[x_k]$ is an operator we have $\mathrm{Exp}'(u_k[x_k]) = \mathrm{Exp}(u_k[x_k]) = \langle u_k[x_k]\rangle$, which clearly has identical effects. On the other hand, if $u_k[x_k]$ is an automaton call, since G_{Au} is acyclic we have $u_k[x_k] \in (Au \setminus \{M\})_\Sigma$, implying that $\mathrm{Exp}'(u_k[x_k])$ has the same effect as $\mathrm{Exp}(u_k[x_k])$ by hypothesis of induction since $|Au \setminus \{M\}| < |Au|$. Thus $\mathrm{Exp}'(M[x]) \in A_\Sigma^{I_M}$, and applying $\mathrm{Exp}'(M[x])$ in a state s has the same effect as applying $\mathrm{Exp}(M[x])$ since the dummy operator δ is always applicable and has no effect. □

We are now ready to prove membership in $\mathbf{\Pi}_2^p$. Because of Lemma 10, instead of verifying the plan $\mathrm{Exp}(r[x])$, we can verify the plan $\mathrm{Exp}'(r[x])$ given by the alternative expansion function Exp'.

Lemma 11. *Plan verification for* AUTR *is in* $\mathbf{\Pi}_2^p$.

Proof. We prove the lemma by reducing plan verification for automaton plans to plan verification for CRARs, which is known to be $\mathbf{\Pi}_2^p$-complete (Bäckström et al., 2012b). Consider any automaton plan $\rho = \langle \Sigma, A, Au, r, x\rangle$ associated with a STRIPS planning instance $\boldsymbol{p}$. Instead of constructing a p-CRAR for the operator sequence $\mathrm{Exp}(r[x])$ represented by ρ, we construct a p-CRAR for the operator sequence $\mathrm{Exp}'(r[x])$. Due to Lemma 10, $\mathrm{Exp}(r[x])$ is a plan for $\boldsymbol{p}$ if and only if $\mathrm{Exp}'(r[x])$ is a plan for $\boldsymbol{p}$.

A p-CRAR for $\mathrm{Exp}'(r[x])$ can be constructed by modifying the algorithm Find in Figure 7. Instead of using the numbers ℓ_M associated with an automaton plan with uniform expansion, we use the upper bounds I_M on the length of any operator sequence output by each automaton. The only other modification we need to make is add a condition $j \le n$ to the while loop, and if the while loop terminates with $j = n+1$, we should return δ, since this means that $I_{u_1} + \cdots + I_{u_n} < k \le I_u$. The complexity of the resulting p-CRAR is identical to that in the proof of Theorem 1 since the numbers I_M are within the bounds used in that proof, i.e. $I_M \le (\mathcal{S} \cdot Ar)^{|Au|}$ for each $M \in Au$. □

As an example, consider the automaton plan $\rho = \langle \{0,1\}, \{a_1, a_2\}, \{\mathsf{M}_1, \mathsf{M}_2, \mathsf{M}_3\}, \mathsf{M}_1, 100\rangle$ with M_1, M_2, and M_3 defined in Figure 2. Applying the definitions we obtain $\mathcal{S} = 3$, $Ar = 4$, $h(\mathsf{M}_1) = 2$, $h(\mathsf{M}_2) = 1$, and $h(\mathsf{M}_3) = 0$, which yields

$$\begin{aligned}I_{\mathsf{M}_1} &= (\mathcal{S} \cdot Ar)^{1+h(\mathsf{M}_1)} = 12^3 = 1728,\\ I_{\mathsf{M}_2} &= (\mathcal{S} \cdot Ar)^{1+h(\mathsf{M}_2)} = 12^2 = 144,\\ I_{\mathsf{M}_3} &= (\mathcal{S} \cdot Ar)^{1+h(\mathsf{M}_3)} = 12^1 = 12.\end{aligned}$$

Although $I_{\mathsf{M}_1} = 1728$ is a gross overestimate on the length of any operator sequence output by M_1, the number of bits needed to represent I_{M_1} is polynomial in $||\rho||$. Applying the alternative expansion function Exp′ to ρ yields $\mathrm{Exp}'(\mathsf{M}_1[100]) = \langle a_1[0]\rangle; \delta^{143}; \langle a_2[1]\rangle; \delta^{11}; \langle a_1[1]\rangle; \delta^{1571}$.

To prove $\mathbf{\Pi}_2^p$-completeness, it remains to show that plan verification for automaton plans is $\mathbf{\Pi}_2^p$-hard. The proof of the following lemma is quite lengthy, so we defer it to Appendix A.

Lemma 12. *Plan verification for* AUTRUE *is* $\mathbf{\Pi}_2^p$*-hard.*

The main theorem of this section now follows immediately from Lemmas 11 and 12.

Theorem 2. *Plan verification for* AUTRUE *and* AUTR *is* $\mathbf{\Pi}_2^p$*-complete.*

Proof. Since AUTRUE $\sqsubseteq$ AUTR, Lemma 11 implies that plan verification for AUTRUE is in $\mathbf{\Pi}_2^p$, while Lemma 12 implies that plan verification for AUTR is $\mathbf{\Pi}_2^p$-hard. Thus plan verification for both AUTRUE and AUTR is $\mathbf{\Pi}_2^p$-complete. □

5.3 Automaton Plans and Macro Plans

In this section we show that automaton plans are strictly more expressive than macros. To do this, we first define macro plans and show that any macro plan can trivially be converted into an equivalent automaton plan with uniform expansion. We then show that there are automaton plans that cannot be efficiently translated to macro plans.

A *macro plan* $\mu = \langle \mathcal{M}, m_r \rangle$ for a STRIPS instance ***p*** consists of a set $\mathcal{M}$ of macros and a root macro $m_r \in \mathcal{M}$. Each macro $m \in \mathcal{M}$ consists of a sequence $m = \langle u_1, \ldots, u_n \rangle$ where, for each $1 \leq k \leq n$, u_k is either an operator in A_Σ or another macro in $\mathcal{M}$. The *expansion* of m is a sequence of operators in A_Σ^* obtained by recursively replacing each macro in $\langle u_1, \ldots, u_n \rangle$ with its expansion. This process is well-defined as long as no macro appears in its own expansion. The plan π represented by μ is given by the expansion of the root macro m_r.

Lemma 13. MACR $\sqsubseteq_p$ AUTRUE.

Proof. To prove the lemma we show that there exists a polynomial p such that for each STRIPS planning instance ***p*** and each macro plan μ representing a solution π to ***p***, there exists an automaton plan ρ for π with uniform expansion such that $||\rho|| = O(p(||\mu||))$.

Each macro of μ is a parameter-free sequence $m = \langle u_1, \ldots, u_l \rangle$ of operators and other macros. We can construct an automaton plan ρ by replacing each macro m with an automaton M_m such that $ar(M_m) = 0$. The automaton M_m has two states s_0 and s, and two edges: one from s_0 to s with label $\epsilon / \langle (w_1, \nu_1), \ldots, (w_l, \nu_l) \rangle$, and one from s to itself with label Σ/ϵ. For each $1 \leq j \leq l$, if u_j is an operator, then w_j is the associated action and the index string $\nu_j \in \Sigma^{ar(u_j)}$ contains the arguments of u_j in the sequence m, which have to be explicitly stated since m is parameter-free. If u_j is a macro, then $w_j = M_{u_j}$ and $\nu_j = \varnothing$ since $ar(M_m) = ar(M_{u_j}) = 0$. The root of ρ is given by $r = M_{m_r}[\epsilon]$, where m_r is the root macro of μ.

We show by induction that, for each macro $m = \langle u_1, \ldots, u_l \rangle$ of μ, $\mathrm{Exp}(M_m[\epsilon])$ equals the expansion of m. The base case is given by $|Au| = 1$. Then m is a sequence of operators in A_Σ^+, and $\mathrm{Apply}(M_m[\epsilon])$ returns m, implying $\mathrm{Exp}(M_m[\epsilon]) = \mathrm{Apply}(M_m[\epsilon]) = m$. If $|Au| > 1$, $\mathrm{Apply}(M_m[\epsilon])$ contains the same operators as m, but each macro u_j in m, where $1 \leq j \leq l$, is replaced with the automaton call $M_{u_j}[\epsilon]$. By hypothesis of induction, $\mathrm{Exp}(M_{u_j}[\epsilon])$ equals the expansion of u_j. Then $\mathrm{Exp}(M_m[\epsilon])$ equals the expansion of m since both are concatenations of identical sequences. It is easy to see that the size of each automaton M_m is polynomial in m. We have shown that each macro plan μ can be transformed into an equivalent automaton plan ρ whose size is polynomial in $||\mu||$, implying the existence of a polynomial p such that $||\rho|| = O(p(||\mu||))$. The automaton plan trivially has uniform expansion since each automaton is always called on the empty input string. □

We next show that automaton plans with uniform expansion are strictly more expressive than macro plans.

Theorem 3. MACR $\sqsubset_p$ AUTRUE *unless* $\mathbf{P} = \mathbf{\Pi}_2^p$.

Proof. Due to Lemma 13 it remains to show that AUTRUE $\not\sqsubseteq_p$ MACR, i.e. that we cannot efficiently translate arbitrary automaton plans with uniform expansion to equivalent macro plans. Bäckström et al. (2012b) showed that plan verification for macro plans is in **P**. Assume that there exists a polynomial-time algorithm that translates any automaton plan with uniform expansion to an equivalent macro plan. Then we could verify automaton plans with uniform expansion in polynomial time, by first applying the given algorithm to produce an equivalent macro plan and then verifying the macro plan in polynomial time. However, due to Theorem 2, no such algorithm can exist unless $\mathbf{P} = \mathbf{\Pi}_2^p$. □

5.4 Automaton Plans and CSARs

In this section we show that automaton plans are strictly less expressive than CSARs, defined as follows:

Definition 14. *Let p be a polynomial function. Given a* STRIPS *planning instance $\boldsymbol{p} = \langle P, A, \Sigma, I, G\rangle$ with associated plan π, a p-*CSAR *of π is a representation ρ such that $||\rho|| \leq p(||\boldsymbol{p}||)$ and ρ outputs the elements of π sequentially, with the time needed to output each element bounded by $p(||\boldsymbol{p}||)$.*

Just as for p-CRARs, the polynomial function p of a p-CSAR has to be independent of the planning instance $\boldsymbol{p}$. We first show that any automaton plan can be transformed into an equivalent p-CSAR in polynomial time. We then show that there are p-CSARs that cannot be efficiently translated to automaton plans.

Lemma 15. AUTR $\sqsubseteq_p$ CSAR.

Proof. To prove the lemma we show that for each STRIPS planning instance $\boldsymbol{p}$ and each automaton plan ρ representing a solution π to $\boldsymbol{p}$, we can efficiently construct a corresponding p-CSAR for π with $p(||\boldsymbol{p}||) = \mathcal{S} \cdot Ar^2 \cdot |Au|$. We claim that the algorithm Next in Figure 8 always outputs the next operator of π in polynomial time. The algorithm maintains the following global variables:

- A call stack $S = [M_1[x_1], \ldots, M_k[x_k]]$ where $M_1[x_1] = r[x]$ is the root of ρ and, for each $1 < i \leq k$, $M_i[x_i]$ is an automaton call that appears in $\mathrm{Apply}(M_{i-1}[x_{i-1}])$.
- An integer k representing the current number of elements in S.
- For each $1 \leq i \leq k$, a sequence θ_i that stores the result of $\mathrm{Apply}(M_i[x_i])$.
- For each $1 \leq i \leq k$, an integer z_i which is an index of θ_i.

```
function Next()
    while z_k = |θ_k| do
        if k = 1 return ⊥
        else
            pop M_k[x_k] from S
            k ← k − 1
    repeat
        z_k ← z_k + 1
        u[x] ← θ_k[z_k]
        if u[x] ∈ A_Σ return u[x]
        else
            push u[x] onto S
            k ← k + 1
            θ_k ← Apply(u[x])
            z_k ← 0
```

Figure 8: Algorithm for finding the next operator of an automaton plan.

Prior to the first call to Next, the global variables are initialized to $S = [r[x]]$, $k = 1$, $\theta_1 = \text{Apply}(r[x])$, and $z_1 = 0$.

The algorithm Next works as follows. As long as there are no more elements in $\text{Apply}(M_k[x_k])$, the automaton call $M_k[x_k]$ is popped from the stack S and k is decremented. If, as a result, $k = 1$ and $\text{Apply}(M_1[x_1])$ contains no more elements, Next returns $\perp$, correctly indicating that the plan π has no further operators.

Once we have found an automaton call $M_k[x_k]$ on the stack such that $\text{Apply}(M_k[x_k])$ contains more elements, we increment z_k and retrieve the element $u[x]$ at index z_k of θ_k. If $u[x] \in A_\Sigma$, $u[x]$ is the next operator of the plan and is therefore returned by Next. Otherwise $u[x]$ is pushed onto the stack S, k is incremented, θ_k is set to $\text{Apply}(u[x])$, z_k is initialized to 0, and the process is repeated for the new automaton call $M_k[x_k] = u[x]$.

Since the expansion graph G_{Au} is acyclic, the number of elements k on the stack is bounded by $|Au|$. Thus the complexity of the while loop is bounded by $|Au|$ since all operations in the loop have constant complexity. Since $\text{Exp}(M_1[x_1]) = \text{Exp}(r[x]) \in A_\Sigma^+$, the repeat loop is guaranteed to find k and z_k such that $u[x] = \theta_k[z_k]$ is an operator, proving the correctness of the algorithm. The only operation in the repeat loop that does not have constant complexity is $\text{Apply}(u[x])$; from Lemma 3 we know that this complexity is bounded by $\mathcal{S} \cdot Ar^2$, and we might have to repeat this operation at most $|Au|$ times. The space required to store the global variables is bounded by $Ar \cdot |Au|$. We have shown that the global variables together with the algorithm Next constitute a p-CSAR with $p(||\boldsymbol{p}||) = O(\mathcal{S} \cdot Ar^2 \cdot |Au|)$. □

We next show that automaton plans are strictly less expressive than p-CSARs. Let $\mathbf{P}_{\leq 1}$ be the subclass of STRIPS planning instances such that at most one operator is applicable in each reachable state. The following lemma is due to Bylander (1994):

Lemma 16. *Plan existence for $\mathbf{P}_{\leq 1}$ is **PSPACE**-hard.*

Proof. Bylander presented a polynomial-time reduction from polynomial-space deterministic Turing machine (DTM) acceptance, a **PSPACE**-complete problem, to STRIPS plan existence. Given any DTM, the planning instance $\boldsymbol{p}$ constructed by Bylander belongs to $\mathbf{P}_{\leq 1}$ and has a solution if and only if the DTM accepts. □

Theorem 4. AUTR $\sqsubset_p$ CSAR *unless* $\boldsymbol{PSPACE} = \boldsymbol{\Sigma}_3^p$.

Proof. Due to Lemma 15 it remains to show that CSAR $\not\sqsubseteq_p$ AUTR. We first show that plan verification for CSARs is **PSPACE**-hard. Given a planning instance $\boldsymbol{p}$ in $\mathbf{P}_{\leq 1}$, let π be the unique plan obtained by always selecting the only applicable operator, starting from the initial state. Without loss of generality, we assume that no operators are applicable in the goal state. Hence π either solves $\boldsymbol{p}$, terminates in a dead-end state, or enters a cycle. It is trivial to construct a p-CSAR ρ for π: in each state, loop through all operators and select the one whose precondition is satisfied. Critically, the construction of ρ is independent of π. Due to Lemma 16, it is **PSPACE**-hard to determine whether $\boldsymbol{p}$ has a solution, i.e. whether the plan represented by ρ solves $\boldsymbol{p}$.

On the other hand, assume that there exists an automaton plan ρ for π such that $||\rho|| = O(p(||\boldsymbol{p}||))$ for some fixed polynomial p. Then we can solve plan existence for $\boldsymbol{p}$ in $\mathbf{NP}^{\mathbf{\Pi}_2^p} = \mathbf{\Sigma}_3^p$ by non-deterministically guessing an automaton plan and verifying that it represents a solution to $\boldsymbol{p}$. This implies that $\mathbf{PSPACE} = \mathbf{\Sigma}_3^p$. □

5.5 Automaton Plans and HTNs

In this section we show that automaton plans are strictly less expressive than HTNs. We begin by defining the class of HTNs that we compare to automaton plans. We then show that we can efficiently transform automaton plans to HTNs, but not the other way around.

Just like planning instances, an HTN involves a set of fluents, a set of operators, and an initial state. Unlike planning instances, however, in which the aim is to reach a goal state, the aim of an HTN is to produce a sequence of operators that perform a given set of tasks. Each task has one or more associated methods that specify how to decompose the task into subtasks, which can either be operators or other tasks. Planning proceeds by recursively decomposing each task using an associated method until only primitive operators remain. While planning, an HTN has to keep track of the current state, and operators and methods are only applicable if their preconditions are satisfied in the current state.

In general, the solution to an HTN is not unique: there may be more than one applicable method for decomposing a task, and each method may allow the subtasks in the decomposition to appear in different order. In contrast, our subsumption relation $\sqsubseteq_p$ is only defined for compact representations of unique solutions. For this reason, we consider a restricted class of HTNs in which the methods associated with a task are mutually exclusive and the subtasks in the decomposition of each method are totally ordered. This class of HTNs does indeed have unique solutions, since each task can only be decomposed in one way. Since this class of HTNs is a strict subclass of that of HTNs in general, our results hold for general HTNs if we remove the requirement on the uniqueness of the solution.

Our definition of HTNs is largely based on SHOP2 (Nau, Ilghami, Kuter, Murdock, Wu, & Yaman, 2003), the state-of-the-art algorithm for solving HTNs. We formally define an HTN domain as a tuple $H = \langle P, A, T, \Theta \rangle$ where $\langle P, A \rangle$ is a planning domain, T a set of function symbols called *tasks*, and Θ a set of function symbols called *methods*. Each method $\theta \in \Theta$ is of the form $\langle t, \mathrm{pre}(\theta), \Lambda \rangle$, where $t \in T$ is the associated task, $\mathrm{pre}(\theta)$ is a precondition, and $\Lambda =$

$\langle(t_1, \varphi_1), \ldots, (t_k, \varphi_k)\rangle$ is a task list where, for each $1 \leq i \leq k$, $t_i \in A \cup T$ is an action or a task and φ_i is an argument map from θ to t_i. The arity of θ satisfies $ar(\theta) \geq ar(t)$, and the arguments of t are always copied onto θ. If $ar(\theta) > ar(t)$, the arguments with indices $ar(t) + 1, \ldots, ar(\theta)$ are free parameters of θ that can take on any value. The precondition $\mathrm{pre}(\theta)$ has the same form as the precondition of an action in A, i.e. $\mathrm{pre}(\theta) = \{(p_1, \varphi_1, b_1), \ldots, (p_l, \varphi_l, b_l)\}$ where, for each $1 \leq j \leq l$, $p_l \in P$ is a predicate, φ_l is an argument map from θ to p_l, and b_l is a Boolean. Each task t may have multiple associated methods.

An HTN instance is a tuple $\boldsymbol{h} = \langle P, A, T, \Theta, \Sigma, I, L\rangle$ where $\langle P, A, T, \Theta\rangle$ is an HTN domain, Σ a set of objects, $I \subseteq P_\Sigma$ an initial state, and L a task list. An HTN instance implicitly defines a set of grounded tasks T_Σ and a set of grounded methods Θ_Σ, and the task list $L \in T_\Sigma^+$ is a sequence of grounded tasks. The precondition $\mathrm{pre}(\theta[xy])$ of a grounded method $\theta[xy] \in \Theta_\Sigma$, where x is the parameters copied from t and y is an assignment to the free parameters of θ, is derived from $\mathrm{pre}(\theta)$ in the same way as the precondition $\mathrm{pre}(a[x])$ of an operator $a[x]$ is derived from $\mathrm{pre}(a)$.

Unlike STRIPS planning, the aim of an HTN instance is to recursively expand each grounded task $t[x] \in T_\Sigma$ in L by applying an associated grounded method $\theta[xy]$ until only primitive operators remain. The grounded method $\theta[xy]$ is applicable if the precondition $\mathrm{pre}(\theta[xy])$ is satisfied, and applying $\theta[xy]$ replaces task $t[x]$ with the sequence of grounded operators or tasks obtained by applying the sequence of argument maps in Λ to $\theta[xy]$. The problem of plan existence for HTNs is to determine if such an expansion is possible.

Lemma 17. AUTR $\sqsubseteq_p$ HTN.

The proof of Lemma 17 appears in Appendix B. Intuitively, the idea is to construct an HTN in which tasks are associated with states in the graphs of the automata, and methods with the edges of these graphs. The HTN emulates an execution model for ρ: each grounded task corresponds to an automaton M, a current state s in the graph of M, an input string $x \in \Sigma^{ar(M)}$, and an index k of x. The associated grounded methods indicate the possible ways to transition from s to another state. Given an edge with label σ/u, the corresponding method is only applicable if $x_k = \sigma$, and applying the method recursively applies all operators and tasks in the sequence u, followed by the task associated with the next state, incrementing k if necessary.

Theorem 5. AUTR $\sqsubset_p$ HTN *unless* ***PSPACE*** $= \boldsymbol{\Sigma}_3^p$.

Proof. Due to Lemma 17 it remains to show that HTN $\not\sqsubseteq_p$ AUTR. Erol et al. (1996) showed that the problem of plan existence for propositional HTNs with totally ordered task lists is **PSPACE**-hard. Their proof is by reduction from propositional STRIPS planning, and the number of applicable methods for each task equals the number of applicable STRIPS operators in the original planning instance, which can in general be larger than one. However, due to Lemma 16, we can instead reduce from the class $\mathbf{P}_{\leq 1}$, resulting in HTNs with at most one applicable method for each task.

If there exists a polynomial-time algorithm that translates HTNs to equivalent automaton plans, we can solve plan existence for HTNs in $\mathbf{NP}^{\boldsymbol{\Pi}_2^p} = \boldsymbol{\Sigma}_3^p$ by non-deterministically guessing an automaton plan and verifying that the automaton plan is a solution. This implies that $\mathbf{PSPACE} = \boldsymbol{\Sigma}_3^p$. $\square$

The reasoning used in the proof of Theorem 5 can also be used to show that HTN $\not\sqsubseteq_p$ CRAR, implying that random access is bounded away from polynomial for HTNs. However, it is unknown whether CRARs can be efficiently translated to HTNs.

function Lowest($\langle P, A, \Sigma, I, G\rangle$)
    $s \leftarrow I$
    while $G \not\subseteq s$ **do**
        $O \leftarrow \{o_i \in A_\Sigma : o_i$ is applicable in $s\}$
        $m \leftarrow \min_i o_i \in O$
        $s \leftarrow (s \setminus \mathrm{pre}(o_m)_-) \cup \mathrm{pre}(o_m)_+$

Figure 9: Algorithm that always selects the applicable operator with lowest index.

One important difference between automaton plans and HTNs is that the latter keeps track of the state. We conjecture that such state-based compact representations are hard to verify in general. Consider the algorithm in Figure 9 that always selects the applicable operator in A_Σ with lowest index. This algorithm is a compact representation of a well-defined operator sequence. Plan verification for this compact representation is **PSPACE**-hard due to Lemma 16, since for planning instances in $\mathbf{P}_{\leq 1}$, the algorithm will always choose the only applicable operator. Arguably, our algorithm is the simplest state-based compact representation one can think of, but plan verification is still harder than for automaton plans.

6. Related Work

The three main sources of inspiration for automaton plans are macro planning, finite-state automata, and string compression. Below, we briefly discuss these three topics and their connections with automaton plans.

6.1 Macro Planning

The connection between macros and automaton plans should be clear at this point: the basic mechanism in automaton plans for recursively defining plans is a direct generalization of macros. In the context of automated planning, macros were first proposed by Fikes et al. (1972) as a tool for plan execution and analysis. The idea did not immediately become widespread and even though it was used in some planners, it was mainly viewed as an interesting idea with few obvious applications. Despite this, advantageous ways of exploiting macros were identified by, for instance, Minton and Korf: Minton (1985) proposed storing useful macros and adding them to the set of operators in order to speed up search while Korf (1987) showed that the search space over macros can be exponentially smaller than the search space over the original planning operators.

During the last decade, the popularity of macros has increased significantly. This is, for example, witnessed by the fact that several planners that exploit macros have participated in the International Planning Competition. MARVIN (Coles & Smith, 2007) generates macros online that escape search plateaus, and offline from a reduced version of the planning instance. MACRO-FF (Botea, Enzenberger, Müller, & Schaeffer, 2005) extracts macros from the domain description as well as solutions to previous instances solved. WIZARD (Newton, Levine, Fox, & Long, 2007) uses a genetic algorithm to generate macros. Researchers have also studied how macros influence the computational complexity of solving different classes of planning instances. Giménez and Jonsson (2008) showed that plan generation is provably polynomial for the class 3S of planning instances if the solution can be expressed using macros. Jonsson (2009) presented a similar algorithm for optimally solving a subclass of planning instances with tree-reducible causal graphs. In both cases, planning instances in the respective class can have exponentially long optimal solutions, making it impossible to generate a solution in polynomial time without the use of macros.

6.2 Finite State Automata

When we started working on new plan representations, it soon become evident that automata are a convenient way of organizing the computations needed "inside" a compactly represented plan. This thought was not particularly original since automata and automaton-like representations are quite common in planning. In order to avoid confusion, we want to emphasize that our automaton hierarchies are not equivalent to the concept of *hierarchical automata*. The term hierarchical automata is used in the literature as a somewhat loose collective term for a large number of different approaches to automaton-based hierarchical modelling of systems; notable examples can be found in control theory (Zhong & Wonham, 1990) and model checking (Alur & Yannakakis, 1998).

There are many examples where solutions to planning problems are represented by automata but these examples are, unlike automaton plans, typically in the context of non-deterministic planning. Cimatti, Roveri, and Traverso (1998) presented an algorithm that, when successful, returns strong cyclic solutions to non-deterministic planning instances. Winner and Veloso (2003) used examples to learn generalized plans. Bonet, Palacios, and Geffner (2010) used a transformation to classical planning to generate finite-state controllers for non-deterministic planning. Finally, Hu and De Giacomo (2013) presented a general algorithm for synthesizing finite state controllers based on a behavior specification of the domain.

Automata have also been used for representing other objects in planning. Hickmott, Rintanen, Thiébaux, and White (2007) and LaValle (2006) used automata to represent the entire planning instance. In contrast, Toropila and Barták (2010) used automata to represent the domains of individual variables of the instance. Baier and McIlraith (2006) showed how to convert an LTL representation of *temporally extended goals*, i.e. conditions that must hold over the intermediate states of a plan, into a non-deterministic finite automaton.

6.3 String Compression

The ideas behind automaton plans and macro plans are closely related to string compression. Most algorithms for string compression are variants of the pioneering work by Lempel and Ziv (1976). Normally, the compressed representation of a string is a *straight line program (SLP)* which is a context free grammar that can only generate one single string. One might say that this is precisely what a hierarchical macro plan is. Although not widely used, there have also been attempts to use automaton representations of strings in order to achieve more compact representations (cf., see Zipstein, 1992). One might say that such approaches generalize SLPs in a way that is very similar to the way automaton plans generalize macro plans. It is important to note that string compression algorithms *per se* have limited interest when it comes to representing plans. The basic reason is that a complete plan first needs to be generated and then compressed by the compression algorithm, and this excludes the utilization of compact plans in the planning process. For instance, the previously mentioned polynomial-time macro planning algorithms (Giménez & Jonsson, 2008; Jonsson, 2009) cannot be replaced (with preserved computational properties) by a planner combined with a string compression algorithm since the planner may need to produce an exponentially long plan.

String compression usually makes no assumptions at all about the content of the string to represent. This makes the methods very general although often not optimal for a particular application. There are examples, though, of more specialised representations. For instance, Subramanian and Shankar (2005) present a method for compressing XML documents by using automata that are based on the XML syntax. Our automaton plans, just like macro plans, do not make any particular assumptions either about the sequence (or string) to be represented. However, it should be evident from our examples that we primarily intend the automata of a plan representation to have some functional correspondence to the plan structure.

7. Discussion

We have introduced the novel concept of automaton plans, i.e. plans represented by hierarchies of finite state automata. Automaton plans extend macro plans by allowing parameterization and branching, and can be used to represent solutions to a variety of planning instances. We have showed that automaton plans are strictly more expressive than macro plans, but strictly less expressive than HTNs and CSARs, i.e. compact representations allowing polynomial-time sequential access. The precise relationship between automaton plans and CRARs is still an open question, but we have presented a partial result: that automaton plans with uniform expansion can be transformed into CRARs.

In our definition of automaton plans we have restricted ourselves to STRIPS planning, and a possible extension would be to consider more general planning formalisms. Below we describe how such an extension would affect the complexity results in Section 5. Most of our transformations are valid for any string, and hence independent of the planning formalism. In particular, macro plans can always be translated to automaton plans with uniform expansion, the latter can always be transformed to CRARs, and automaton plans can always be transformed to CSARs. However, we are not aware of any HTN formalism that extends the action definition to allow for more complex actions. Hence our transformation from automaton plans to HTNs involve actions with STRIPS-style preconditions and effects. All separation results for STRIPS also carry over to more general planning formalisms since there cannot exist a polynomial function for translating all instances.

Although we have mainly studied the computational properties of automaton plans in the context of compact plan representation, we believe that automaton plans may find other uses. Probably the most interesting question from a practical perspective is how to construct automaton plans. We do not have a definite answer to this question, but there are at least two ideas that we believe are worth exploring. One possible way to generate automaton plans is to first construct an HTN for a given domain, and then use the HTN to solve instances of the domain. Instead of flattening the solution as is typically done, the idea would be to keep its hierarchical structure and transform it into an automaton plan. It does not matter which HTN representation we consider as long as we can verify whether a solution to an instance is valid; once the solution has been verified the transformation to an automaton plan might become tractable, at least in practical cases. This approach would likely require more sophisticated techniques for generating HTNs than those currently available, unless the HTN is already provided.

Another interesting extension of automaton plans is to allow recursive automata that include calls to themselves. Consider the automaton M_n from Section 4 for moving n discs in Towers of Hanoi. If we introduced symbols $j_1, \ldots, j_N$ representing the number of discs, we could define a single recursive automaton M that includes a fourth parameter j_n representing the number of discs

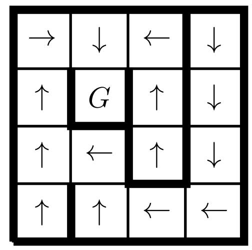

Figure 10: An example contingent planning instance.

to be moved. The recursive calls to M on input j_n would include the symbol j_{n-1}, effectively decrementing the number of discs. For the recursive mechanism to work there would need to exist a base case for which no more recursive calls are made (in the case of Towers of Hanoi, the base case typically consists in moving 0 or 1 discs). We remark that the computational properties of automaton plans from this paper would no longer apply since the expansion graph would no longer be acyclic.

Finally, modified automaton plans could be used to represent contingent plans compactly. Solutions to contingent planning instances are typically in the form of directed graphs in which each node is a *belief state*, i.e. a subset of states. Edges correspond to actions that are applicable in each belief state, as well as *observations* about the current state. The outcome of an observation is a single bit indicating whether the current value of a given fluent is true or false. Since the outcome of an observation is uncertain, each observation *splits* a belief state into one belief state containing states for which the fluent is true, and one containing states for which the fluent is false. The solution represents a *policy*, indicating which action should be applied in each belief state.

Figure 10 shows an example of a contingent planning instance. Each location is described by an (x, y)-coordinate. At each location there is an arrow indicating which way to go, and a flag indicating whether we have reached the target destination G. Two fluents are sufficient to represent the direction in which the arrow is pointing:

00: up
01: down
10: left
11: right

There are four actions U, D, L, and R for moving up, down, left, and right, respectively. The (x, y)-coordinate is not observable, and the initial state is unknown. In each state we can only observe whether or not we have reached the destination and in which direction the arrow is pointing.

Figure 11 shows an automaton that represents a solution to the example contingent planning instance. The set of symbols is $\Sigma = \{0, 1\}$, i.e. symbols are outcomes of observations and are used to branch on edges of the automaton. In each state we make three consecutive observations: whether or not we have reached the goal (0 or 1), and in which direction the arrow is pointing (two bits). If we reach the goal we move to a state where we simply consume the remaining observations (if any). If not, we use the direction of the arrow to decide which action to apply next.

The automaton solution is independent of the size of the contingent planning instance as long as the arrows point in the right direction. In contrast, the number of belief states is doubly exponential in the number of fluents, i.e. $2^{2^{|P_\Sigma|}}$. If we add locations to the example contingent planning problem, the number of belief states increases exponentially, and so does the size of a solution in the form

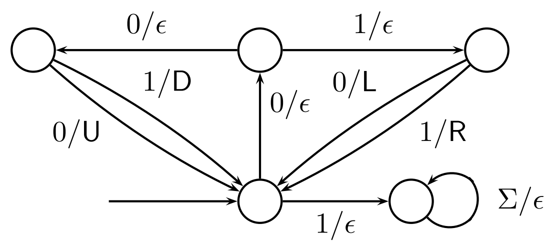

Figure 11: An automaton representing a contingent plan.

of a directed graph. Although the example automaton plan is not hierarchical, it is not difficult to imagine that navigation is a subtask of a larger problem, in which case we could call the automaton from other automata.

Although the semantics of contingent planning is different from that of classical planning, the automata in Figure 11 can be used to construct a perfectly valid automaton plan. However, our definition imposes two restrictions on contingent plans. First, since automata in automaton plans have fixed arity, we can only represent contingent plans with a constant number of observations. Second, the entire sequence of observations has to be passed as input to the automaton beforehand. It may therefore make sense to consider a relaxation of the definition: allowing input to be passed to an automaton online, thus allowing the arity of an automaton to vary.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions.

Appendix A. Proof of Lemma 12.

In this section we prove Lemma 12, which states that plan verification for automaton plans with uniform expansion is $\mathbf{\Pi}_2^p$-hard. We prove the lemma by reduction from $\forall\exists$-SAT to plan verification for automaton plans with uniform expansion. Our proof proceeds in three steps. We first show how to construct a STRIPS planning instance $\boldsymbol{p}_F$ for any given $\forall\exists$-SAT formula F. We then prove that there exists an operator sequence π_F such that $\boldsymbol{p}_F$ has unique solution π_F if and only if F is satisfiable. Finally, we construct an automaton plan ρ_F with uniform expansion that represents the sequence π_F, i.e. ρ_F represents a valid plan for $\boldsymbol{p}_F$ if and only if F is satisfiable.

Construction 18. *Let $F = \forall x_1 \cdots \forall x_m \exists y_1 \cdots \exists y_n \cdot \phi$ be a $\forall\exists$-SAT formula where ϕ is a 3SAT formula, and let $L_F = \{\ell_1, \ldots, \ell_{2(m+n)}\}$ be a set of literals, where $\ell_{2i-1} = \overline{x_i}$ and $\ell_{2i} = x_i$ for each $1 \leq i \leq m$ and $\ell_{2(m+j)-1} = \overline{y_j}$ and $\ell_{2(m+j)} = y_j$ for each $1 \leq j \leq n$. Also define a total order $<$ on L_F such that $\ell_i < \ell_j$ if and only if $i < j$. The formula $\phi = (c_1 \wedge \cdots \wedge c_h)$ is a conjunction of 3-literal clauses $c_k = \ell_k^1 \vee \ell_k^2 \vee \ell_k^3$ such that $\ell_k^1, \ell_k^2, \ell_k^3 \in L_F$. Assume without loss of generality that $\ell_k^1 \leq \ell_k^2 \leq \ell_k^3$.*

Given the formula F, construct a STRIPS planning instance $\boldsymbol{p}_F = \langle P_{\Sigma_F}, A_{\Sigma_F}, \Sigma_F, I_F, G_F\rangle$ where $P_{\Sigma_F} = \{fx, fy, fs, sat, x_1, \ldots, x_m, y_1, \ldots, y_n, v_0, \ldots, v_h\}$, $\Sigma_F = \{0, 1\}$, $I_F = \varnothing$, $G_F =$

$\{fx, sat, x_1, \ldots, x_m\}$, *and* A_{Σ_F} *contains the following operators, each described on the form* $\mathrm{pre}(a) \Rightarrow \mathrm{post}(a)$*:*

$$
\begin{aligned}
&os: && \{\overline{fx}, \overline{fy}, \overline{v_0}\} \Rightarrow \{v_0, fs\}\\
&ol_k^1: && \{v_{k-1}, \overline{v_k}, \ell_k^1\} \Rightarrow \{v_k\}\\
&ol_k^2: && \{v_{k-1}, \overline{v_k}, \overline{\ell_k^1}, \ell_k^2\} \Rightarrow \{v_k\}\\
&ol_k^3: && \{v_{k-1}, \overline{v_k}, \overline{\ell_k^1}, \overline{\ell_k^2}, \ell_k^3\} \Rightarrow \{v_k\}\\
&on_k: && \{v_{k-1}, \overline{v_k}, \overline{\ell_k^1}, \overline{\ell_k^2}, \overline{\ell_k^3}\} \Rightarrow \{v_k, \overline{fs}\}\\
&ot: && \{v_h, fs\} \Rightarrow \{fy, \overline{v_0}, \ldots, \overline{v_h}, sat\}\\
&of: && \{v_h, \overline{fs}\} \Rightarrow \{fy, \overline{v_0}, \ldots, \overline{v_h}\}\\
&oy_j: && \{fy, \overline{y_j}, y_{j+1}, \ldots, y_n\} \Rightarrow \{\overline{fy}, y_j, \overline{y_{j+1}}, \ldots, \overline{y_n}\}\\
&od: && \{fy, y_1, \ldots, y_n\} \Rightarrow \{fx, \overline{fy}, \overline{y_1}, \ldots, \overline{y_n}\}\\
&ox_i: && \{fx, sat, \overline{x_i}, x_{i+1}, \ldots, x_m\} \Rightarrow \{\overline{fx}, \overline{sat}, x_i, \overline{x_{i+1}}, \ldots, \overline{x_m}\}
\end{aligned}
$$

We explain the intuition behind the planning instance $\boldsymbol{p}_F$. First note that all predicates and actions are parameter-free, so the set of fluents equals the set of predicates and the set of operators equals the set of actions. No function map is thus necessary to describe the pre- or postcondition of an operator. The indices used for operators are in the ranges $1 \leq i \leq m$, $1 \leq j \leq n$, and $1 \leq k \leq h$.

A plan for $\boldsymbol{p}_F$ takes the form of three nested loops. The outer loop uses operators of type ox_i to iterate over all assignments to $x_1, \ldots, x_m$, the universal variables of the formula F. The middle loop uses operators of type oy_j to iterate over all assignments to $y_1, \ldots, y_n$, the existential variables of F. The inner loop uses operators of type $ol_k^1, ol_k^2, ol_k^3, on_k$ to iterate over all clauses of the 3SAT formula ϕ, at the same time verifying whether ϕ is satisfied given the current assignment to $x_1, \ldots, x_m, y_1, \ldots, y_n$. The remaining fluents have the following functions:

fx: control the applicability of the operators ox_i used to iterate over all assignments to $x_1, \ldots, x_m$.

fy: control the applicability of the operators oy_j used to iterate over all assignments to $y_1, \ldots, y_n$.

v_0: control the applicability of the operators $ol_k^1, ol_k^2, ol_k^3, on_k$ used to iterate over all clauses.

fs: remember whether ϕ is satisfied for the current assignment to $x_1, \ldots, x_m, y_1, \ldots, y_n$.

sat: remember whether $\exists y_1, \ldots, y_n \phi$ is satisfied for the current assignment to $x_1, \ldots, x_m$.

During each inner loop, we first have to apply operator os to add fluent v_0. For each clause c_k, we then have to apply one of the operators $ol_k^1, ol_k^2, ol_k^3, on_k$ to add v_k. Finally, we have to apply one of ot and of to delete $v_0, \ldots, v_h$. During this process, fluent fs is added by os and deleted only if on_k is applied for some clause c_k.

Operators ot and of also add fy, causing an operator of type oy_j to become applicable. If fs is true, operator ot also adds sat. Applying an operator of type oy_j has the effect of moving to the next assignment to $y_1, \ldots, y_n$. When $y_1, \ldots, y_n$ are all true, operator od is applicable instead, adding fluent fx and resetting $y_1, \ldots, y_n$ to false. When fx is true, we can apply an operator of type ox_i to move to the next assignment to $x_1, \ldots, x_m$. These operators all require sat as a precondition. When $x_1, \ldots, x_m$ are all true, the goal state G_F ensures that we iterate one last time over the middle loop to make fx and sat true.

Lemma 19. *The* $\forall\exists$*-SAT formula* F *is satisfiable if and only if the planning instance* $\mathbf{p}_F$ *has a unique solution* π_F *of the form*

$$\begin{aligned} \pi_F &= E_0, ox, E_1, ox, \ldots, ox, E_{2^m-1}, \\ E_i &= V_i^0, oy, V_i^1, oy, \ldots, oy, V_i^{2^n-1}, od, \\ V_i^j &= os, oz_1, oz_2, \ldots, oz_h, ow, \end{aligned}$$

where each ox *is an operator among* $ox_1, \ldots, ox_m$*, each* oy *is an operator among* $oy_1, \ldots, oy_n$*, each* oz_k*, for* $1 \le k \le h$*, is an operator among* ol_k^1*,* ol_k^2*,* ol_k^3*,* on_k*, and* ow *is an operator among* ot*,* of*.*

Proof. We prove the lemma by showing the following:

1. In each state reachable from the initial state I_F, at most one operator is applicable.
2. Repeatedly selecting the only applicable operator results in the sequence π_F given above.
3. The sequence π_F is applicable in the initial state I_F and the goal state G_F holds in the resulting state if and only if F is satisfiable.

We first show that in each reachable state, there exists $0 \le k \le h+1$ such that a) when $k = 0$, all variables among $v_0, \ldots, v_h$ are false; b) when $1 \le k \le h$, $v_0, \ldots, v_{k-1}$ are true and $v_k, \ldots, v_h$ are false; and c) when $k = h+1$ all variables among $v_0, \ldots, v_h$ are true. While doing so we ignore operators oy_j, od, and ox_i since they have no effect on $v_0, \ldots, v_h$. In the initial state all fluents are false so the statement holds for $k = 0$. If $k = 0$, the only applicable operator is os which sets v_0 to true, effectively incrementing the current value of k. If $1 \le k \le h$, the possible operators are $ol_k^1, ol_k^2, ol_k^3, on_k$. However, the preconditions of these operators are mutually exclusive such that exactly one of them is applicable. Each of these operators sets v_k to true, incrementing the value of k. Finally, if $k = h+1$, the possible operators are ot and of. The preconditions of these operators are also mutually exclusive, and both operators set $v_0, \ldots, v_h$ to false, effectively resetting the value of k to 0.

The only operators affecting fluents $y_1, \ldots, y_n$ are $oy_1, \ldots, oy_n$. Each of these requires fy as a precondition and sets fy to false. The only operators setting fy to true are ot and of, which both reset k to 0. This implies that each time we want to apply an operator of type oy_j, fluents $v_0, \ldots, v_h$ need to go through a complete cycle from $k = 0$ to $k = h+1$ and finish with ot or of. This cycle corresponds exactly to the sequence V_i^j of the lemma.

We next show that $y_1, \ldots, y_n$ act as a binary counter from 0 to $2^n - 1$, enumerating all assignments to the existential variables of the formula F. Note that the preconditions of $oy_1, \ldots, oy_n$ are mutually exclusive, since oy_n requires $\overline{y_n}$, oy_{n-1} requires $\overline{y_{n-1}}, y_n$, and so on. Specifically, the only applicable operator is oy_j, where $1 \le j \le n$ is the largest index such that y_j is false. Repeatedly applying the only available operator results in the following series of values for $y_1, \ldots, y_n$:

$$
\begin{array}{c}
0\cdots000 \\
0\cdots001 \\
0\cdots010 \\
0\cdots011 \\
0\cdots100 \\
0\cdots101 \\
\vdots
\end{array}
$$

If $y_1, \ldots, y_n$ are all true, there exists no applicable operator of type oy_j, but operator od is applicable instead.

The only operators affecting fluents $x_1, \ldots, x_m$ are $ox_1, \ldots, ox_m$. Each of these requires fx as a precondition and sets fx to false. The only operator setting fx to true is od, which also resets $y_1, \ldots, y_n$ to false. This implies that each time we want to apply an operator of type ox_i, fluents $y_1, \ldots, y_n$ need to go through a complete cycle from 0 to $2^n - 1$ and finish with od. This cycle corresponds exactly to the sequence E_i of the lemma.

Since the operators $ox_1, \ldots, ox_m$ have the same form as $oy_1, \ldots, oy_n$, at most one of them is applicable, and repeatedly applying the only available operator among $ox_1, \ldots, ox_m$ causes the fluents $x_1, \ldots, x_m$ to act as a binary counter from 0 to $2^m - 1$, enumerating all assignments to the universal variables of the formula F. The precondition $\{\overline{fx}, \overline{fy}\}$ of operator os ensures that os is not applicable whenever we are about to apply an operator of type oy_j or ox_i. To achieve the goal state $G_F = \{fx, sat, x_1, \ldots, x_m\}$, the fluents $x_1, \ldots, x_m$ have to complete a full cycle from 0 to $2^m - 1$, and to set fx to true the fluents $y_1, \ldots, y_n$ have to complete one last cycle after setting $x_1, \ldots, x_m$ to true. This corresponds exactly to the sequence π_F of the lemma.

Finally, we need to show that the sequence π_F is applicable in the initial state I_F and that the goal state G_F holds in the resulting state if and only if the formula F is satisfiable. In the initial state I_F and after using an operator of type ox_i to iterate over $x_1, \ldots, x_m$, the fluent sat is false. Each time we apply os, fluent fs is added. While iterating over the clauses, fs is deleted if and only if we apply an operator of type on_k, i.e. there exists a clause c_k that is not satisfied by the current assignment to $x_1, \ldots, x_m, y_1, \ldots, y_n$. If fs is true at the end of this loop, operator ot adds sat.

If the formula F is not satisfied, there exists an assignment to $x_1, \ldots, x_m$ such that ϕ is unsatisfied for each assignment to $y_1, \ldots, y_n$. Consequently, sat is false when applying the operator od making fx true for that assignment to $x_1, \ldots, x_m$. Then either no operator of type ox_i is applicable (if at least one fluent among $x_1, \ldots, x_m$ is false) or the goal state G_F does not hold in the resulting state (if $x_1, \ldots, x_m$ are all true). Conversely, if F is satisfied, sat is always true when applying od, causing π_F to be applicable and G_F to hold in the resulting state. □

We now proceed to construct an automaton plan that represents the operator sequence π_F described in Lemma 19.

Construction 20. *Let* $\boldsymbol{p}_F = \langle P_{\Sigma_F}, A_{\Sigma_F}, \Sigma_F, I_F, G_F \rangle$ *be the planning instance defined in Construction 18. Construct an automaton plan* $\rho_F = \langle \Sigma_F, A_{\Sigma_F}, Au_F, r, x \rangle$ *with the following properties:*

- $Au_F = \{\mathsf{X}_1, \ldots, \mathsf{X}_m, \mathsf{Y}_1, \ldots, \mathsf{Y}_n, \mathsf{S}_1, \ldots, \mathsf{S}_{h+1}, \mathsf{U}_1, \ldots, \mathsf{U}_{h+1}\}$,

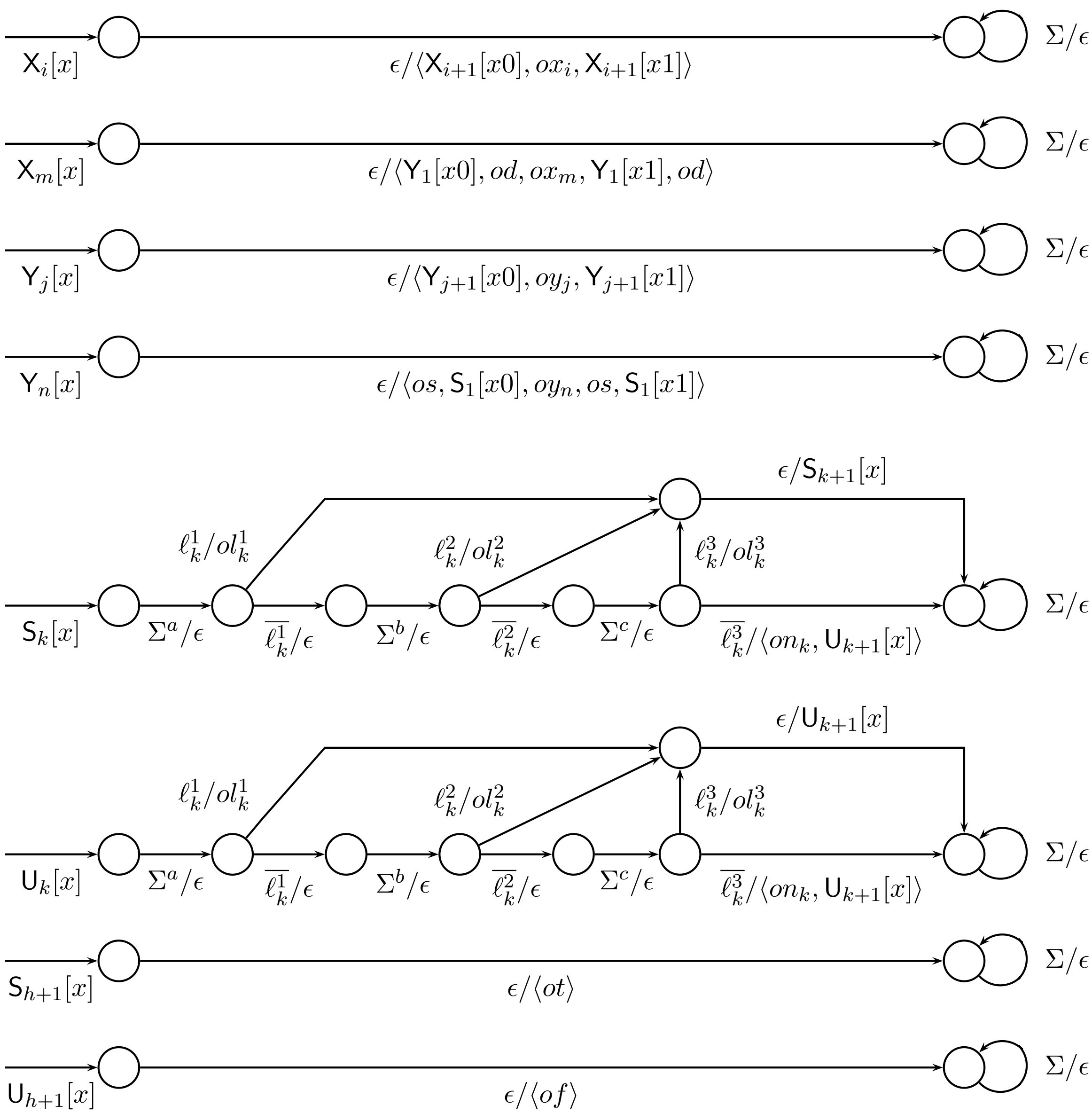

Figure 12: Graphs of the automata defined in Construction 20.

- *for each* $1 \leq i \leq m$, $ar(\mathsf{X}_i) = i - 1$,
- *for each* $1 \leq j \leq n$, $ar(\mathsf{Y}_j) = m + j - 1$,
- *for each* $1 \leq k \leq h + 1$, $ar(\mathsf{S}_k) = ar(\mathsf{U}_k) = m + n$,
- $r = \mathsf{X}_1$ *and* $x = \epsilon$.

The graphs associated with the automata in Au_n *are shown in Figure 12. The indices used to describe automata are in the ranges* $1 \leq i < m$, $1 \leq j < n$, *and* $1 \leq k \leq h$. *For each automaton* $M[x]$, *the argument maps are described as* $u[x]$, $u[x0]$, *or* $u[x1]$, *indicating that the input string* x *of* $M[x]$ *is copied onto* u, *possibly appending* 0 *or* 1 *at the end.*

Intuitively, the input string x of automata S_k and U_k represents an assignment to the fluents $x_1, \ldots, x_m, y_1, \ldots, y_n$. The edge with label Σ^a consumes a input symbols, where a is the number of variables that precede the variable corresponding to the literal ℓ_k^1. Likewise, b is the number of variables between ℓ_k^1 and ℓ_k^2, and c is the number of variables between ℓ_k^2 and ℓ_k^3. The assignment to ℓ_k^1 determines whether to output operator ol_k^1 or continue checking whether c_k is satisfied.

Note that all automata defined in Construction 20 have uniform expansion. For each $1 \leq k \leq h$ and $x \in \Sigma^{m+n}$, $\text{Apply}(\mathsf{S}_k[x])$ contains exactly one operator among $ol_k^1, ol_k^2, ol_k^3, on_k$, followed by either $\mathsf{S}_{k+1}[x]$ or $\mathsf{U}_{k+1}[x]$. The same is true for U_k. We show that the automaton plan ρ_F defined in Construction 20 indeed represents the operator sequence π_F from Lemma 19.

Lemma 21. *The automaton plan* ρ_F *represents the operator sequence* π_F.

Proof. For each $1 \leq i \leq m$, the input string x to the automaton X_i represents an assignment of values to the fluents $x_1, \ldots, x_{i-1}$ of the planning instance $\boldsymbol{p}_F$. For each $1 \leq j \leq n$, the input string to Y_j represents an assignment of values to $x_1, \ldots, x_m, y_1, \ldots, y_{j-1}$, and for each $1 \leq k \leq h+1$, the input string to S_k and U_k represents a complete assignment of values to $x_1, \ldots, x_m, y_1, \ldots, y_n$. Consequently, the symbol that X_i appends to x represents the current value of x_i, and the symbol that Y_j appends represents the current value of y_j.

Since each automaton X_i first sets x_i to 0 and then to 1, the series of assignments to $x_1, \ldots, x_m$ becomes

$$\begin{array}{c} 0 \cdots 000 \\ 0 \cdots 001 \\ 0 \cdots 010 \\ 0 \cdots 011 \\ 0 \cdots 100 \\ 0 \cdots 101 \\ \vdots \end{array}$$

which is identical to the binary counter on $x_1, \ldots, x_m$ induced by the plan π_F. Likewise, for each assignment to $x_1, \ldots, x_m$, the assignments to $y_1, \ldots, y_n$ describe the same binary counter as π_F.

Before changing the value of fluent x_i from 0 to 1, the automaton X_i inserts operator ox_i. The last assignment to $x_{i+1}, \ldots, x_m$ before appending ox_i is $1, \ldots, 1$, and the first assignment to $x_{i+1}, \ldots, x_m$ after appending ox_i is $0, \ldots, 0$. Consequently, the automata perfectly emulate the pre- and postcondition of ox_i. The same is true of the operator oy_j inserted by Y_j. Automaton X_m inserts operator od after each cycle of assignments to $y_1, \ldots, y_n$, and automaton Y_n inserts operator os at the beginning of each iteration over the fluents $v_0, \ldots, v_h$.

For each $1 \leq k \leq h$, the purpose of automata S_k and U_k is to decide which operator to append among $ol_k^1, ol_k^2, ol_k^3, on_k$. To do this, they first access the value of the variable associated with the literal ℓ_k^1. If ℓ_k^1 is satisfied, operator ol_k^1 is appended, else literal ℓ_k^2 is checked, then ℓ_k^3 if necessary. Only if all three literals are unsatisfied by the current variable assignment is operator on_k appended.

The only difference between automata S_k and U_k is that S_{h+1} appends operator ot, while U_{h+1} appends of. Being in automaton S_k indicates that the first $k-1$ clauses of the current 3SAT instance are satisfied by the current variable assignment. Only if clause c_k is unsatisfied does S_k call U_{k+1}, in which case all subsequent calls will be to automata of type U. In other words, another purpose of automata S_k and U_k is to remember the current value of the variable fs. This way, the correct operator among ot and of is appended at the end of each iteration over fluents $v_0, \ldots, v_h$, which concludes the proof. $\square$

To show that plan verification is $\mathbf{\Pi}_2^p$-hard for automaton plans with uniform expansion, given any $\forall\exists$-SAT formula F, construct (in polynomial time) the planning instance $\boldsymbol{p}_F$ in Construction 18 and the automaton plan ρ_F in Construction 20. Lemma 21 states that ρ_F represents the operator sequence π_F defined in Lemma 19. Due to Lemma 19, π_F is a plan for $\boldsymbol{p}_F$ if and only if F is satisfiable. We have thus reduced $\forall\exists$-satisfiability (a $\mathbf{\Pi}_2^p$-complete problem) to plan verification for automaton plans with uniform expansion.

Appendix B. Proof of Lemma 17

In this section we prove Lemma 17, which states that any automaton plan ρ can be efficiently transformed to an equivalent HTN instance $\boldsymbol{h}$. Let $\boldsymbol{p} = \langle P, A, \Sigma, I, G\rangle$ be a STRIPS instance and let $\rho = \langle \Sigma, A, Au, r\rangle$ be an automaton plan representing a solution π to $\boldsymbol{p}$. We define an HTN instance $\boldsymbol{h} = \langle P', A', T, \Theta, \Sigma', I', L\rangle$ as follows:

- $P' = P \cup \{\mathsf{consec}\} \cup \{\mathsf{precedes}\} \cup \{\mathsf{isset}\text{-}M\}_{M \in Au}$ with $ar(\mathsf{consec}) = ar(\mathsf{precedes}) = ar(\mathsf{isset}\text{-}M) = 2$ for each $M \in Au$,
- $A' = A \cup \{\mathsf{set}\text{-}M, \mathsf{unset}\text{-}M\}_{M \in Au}$ with $ar(\mathsf{set}\text{-}M) = ar(\mathsf{unset}\text{-}M) = 2$ for each $M \in Au$,
- $\Sigma' = \Sigma \cup J$, where $J = \{j_0, \ldots, j_K\}$ is a set of indices and $K = \max_{M \in Au} ar(M)$,
- $I' = I \cup \{\mathsf{consec}[j_{i-1}j_i] : 1 \leq i \leq K\} \cup \{\mathsf{precedes}[j_i j_k] : 0 \leq i < k \leq K\}$.

In the induced set of fluents $P'_{\Sigma'}$, the static fluent $\mathsf{consec}[jk]$ is true if and only if j and k are consecutive indices in J, and $\mathsf{precedes}[jk]$ is true if and only if j precedes k in J. For each automaton $M \in Au$, each symbol $\sigma \in \Sigma$, and each index $1 \leq i \leq ar(M)$, the fluent $\mathsf{isset}\text{-}M[\sigma j_i]$ indicates whether the i-th symbol of the input string $x \in \Sigma^{ar(M)}$ of M equals σ. In the induced set of operators $A'_{\Sigma'}$, the operators $\mathsf{set}\text{-}M[\sigma j]$ and $\mathsf{unset}\text{-}M[\sigma j]$ add and delete fluent $\mathsf{isset}\text{-}M[\sigma j]$, respectively.

For each automaton $M \in Au$, we also add the following tasks and methods to the sets T and Θ:

- A task $\mathsf{setall}\text{-}M$ with arity $ar(M)$,
- A method $\mathsf{dosetall}\text{-}M$ with arity $ar(M)$ and associated task $\mathsf{setall}\text{-}M$,

- For each state $s \in S$ of M, a task visit-M-s with arity $ar(M) + 1$,
- For each edge (s, t) with label ϵ/u, a method traverse-M-s-t with arity $ar(M) + 1$ and associated task visit-M-s,
- For each edge (s, t) with label σ/u, $\sigma \in \Sigma$, a method consume-M-s-t with arity $ar(M) + 2$ and associated task visit-M-s,
- For each state $s \in S$ with $|\Sigma|$ outgoing edges, a method finish-M-s with arity $ar(M) + 1$ and associated task visit-M-s.

Formally, the precondition and task list of each method $\theta \in \Theta$ should contain pairs (u, φ) of an action or task u and an associated argument map φ from θ to u. However, to simplify notation we instead describe grounded preconditions and task lists of grounded methods $\theta[xy]$. In the induced set of grounded tasks $T_{\Sigma'}$, the grounded task setall-$M[x]$ sets the current input string of M to x. The lone associated grounded method dosetall-$M[x] \in \Theta_{\Sigma'}$ has empty precondition and the following task list:

$$\begin{aligned}\Lambda = \langle &\mathsf{unset}\text{-}M[\sigma_1 j_1], \ldots, \mathsf{unset}\text{-}M[\sigma_n j_1], \mathsf{set}\text{-}M[x_1 j_1],\\ &\vdots\\ &\mathsf{unset}\text{-}M[\sigma_1 j_{ar(M)}], \ldots, \mathsf{unset}\text{-}M[\sigma_n j_{ar(M)}], \mathsf{set}\text{-}M[x_{ar(M)} j_{ar(M)}]\rangle,\end{aligned}$$

where $\sigma_1, \ldots, \sigma_n$ are the symbols in the set Σ. In other words, dosetall-$M[x]$ first unsets all symbols at each index of the input string, then sets the symbol according to x.

The grounded task visit-M-$s[xj_k]$ indicates that we are currently at state s of automaton M, that the input string is x and that the current index of x is k. If s has a single outgoing edge (s, t) with label $\epsilon/(u, \nu)$, the only associated grounded method is traverse-M-s-$t[xj_k]$ with empty precondition. Let $u[\varphi(x)]$ be the result of applying the argument map φ induced by the index string ν to the input string x of M. If $u \in A$, the grounded task list Λ of traverse-M-s-$t[xj_k]$ equals $\Lambda = \langle u[\varphi(x)], \mathsf{visit}\text{-}M\text{-}t[xj_k]\rangle$, effectively applying operator $u[\varphi(x)]$. On the other hand, if $u \in Au$, the task list is $\Lambda = \langle \mathsf{setall}\text{-}u[\varphi(x)], \mathsf{visit}\text{-}u\text{-}s_0[\varphi(x)j_0], \mathsf{visit}\text{-}M\text{-}t[xj_k]\rangle$, first setting the input string of u to $\varphi(x)$ and then visiting the initial state s_0 of u with index j_0. In either case, the grounded task visit-M-$t[xj_k]$ at the end of Λ ensures that we next visit state t of M without incrementing k.

If, instead, s has multiple outgoing edges, then for each outgoing edge (s, t) with label $\sigma/(u, \nu)$ for some $\sigma \in \Sigma$, there is an associated grounded method consume-M-s-$t[xj_k j_{k+1}]$ with precondition $\{\mathsf{consec}[j_k j_{k+1}], \mathsf{precedes}[j_k j_{ar(M)}], \mathsf{isset}\text{-}M[\sigma j_{k+1}]\}$. The index j_{k+1} is a free parameter of consume-M-s-t, but the precondition consec$[j_k j_{k+1}]$ ensures that j_k and j_{k+1} are consecutive indices in J. The precondition precedes$[j_k j_{ar(M)}]$ ensures that $k < ar(M)$, i.e. that there are input symbols left in x to process. Note that indices start at j_0, so it is the symbol at index j_{k+1} of the input string x that should be set to σ. The task list is identical to that of traverse-M-s-$t[xj_k]$, except that the last task visit-M-$t[xj_{k+1}]$ is associated with the next index j_{k+1}, indicating that we have consumed a symbol of the input string.

For each $s \in S$ with $|\Sigma|$ outgoing edges, the grounded method finish-M-$s[xj_k]$ has precondition $j_k = j_{ar(M)}$, i.e. the method is only applicable if we have consumed all symbols of the input string. We assume that $j_k = j_{ar(M)}$ can be checked without introducing an additional predicate in P'. The task list Λ is empty, indicating that we have finished traversing the states of M. The method is not applicable for states with a single outgoing edge since we should fire all applicable ϵ-transitions before terminating.

The task list of the HTN instance $\boldsymbol{h}$ is given by $L = \langle \mathsf{setall}\text{-}r[x], \mathsf{visit}\text{-}r\text{-}s_0[xj_0]\rangle$ where $r[x]$ is the root of the automaton plan ρ. Because of the way tasks and methods are defined, expanding $\mathsf{visit}\text{-}r\text{-}s_0[xj_0]$ corresponds exactly to executing the automaton r with input string x starting from s_0. Thus the expansion of L corresponds exactly to the solution π of $\boldsymbol{p}$ represented by ρ if we remove all instances of operators $\mathsf{set}\text{-}M$ and $\mathsf{unset}\text{-}M$. Since π is a solution to $\boldsymbol{p}$, each operator in the sequence is guaranteed to be applicable.

The only type of task with multiple associated methods is $\mathsf{visit}\text{-}M\text{-}s$. The methods $\mathsf{consume}\text{-}M\text{-}s\text{-}t$ associated with $\mathsf{visit}\text{-}M\text{-}s$ are mutually exclusive since at any moment, $\mathsf{isset}\text{-}M[\sigma j_k]$ is true for at most one symbol $\sigma \in \Sigma$ at each index $1 \leq k \leq ar(M)$ of the input string x of M. If $j_k = j_{ar(M)}$, the method $\mathsf{finish}\text{-}M\text{-}s$ is applicable instead. The task list of each method is totally ordered, implying that the instance $\boldsymbol{h}$ belongs to our restricted class of HTNs with mutually exclusive methods and totally ordered task lists.

References

Alur, R., & Yannakakis, M. (1998). Model Checking of Hierarchical State Machines. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 175–188.

Bäckström, C., Jonsson, A., & Jonsson, P. (2012a). From Macro Plans to Automata Plans. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI)*, pp. 91–96.

Bäckström, C., Jonsson, A., & Jonsson, P. (2012b). Macros, Reactive Plans and Compact Representations. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI)*, pp. 85–90.

Bäckström, C., & Jonsson, P. (2012). Algorithms and Limits for Compact Plan Representation. *Journal of Artificial Intelligence Research*, *44*, 141–177.

Baier, J., & McIlraith, S. (2006). Planning with Temporally Extended Goals Using Heuristic Search. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 342–345.

Bonet, B., Palacios, H., & Geffner, H. (2010). Automatic Derivation of Finite-State Machines for Behavior Control. In *Proceedings of the 24th National Conference on Artificial Intelligence (AAAI)*.

Botea, A., Enzenberger, M., Müller, M., & Schaeffer, J. (2005). Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *Journal of Artificial Intelligence Research*, *24*, 581–621.

Bylander, T. (1994). The Computational Complexity of Propositional STRIPS Planning. *Artificial Intelligence*, *69*, 165–204.

Cimatti, A., Roveri, M., & Traverso, P. (1998). Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI)*, pp. 875–881.

Coles, A., & Smith, A. (2007). MARVIN: A Heuristic Search Planner with Online Macro-Action Learning. *Journal of Artificial Intelligence Research*, *28*, 119–156.

Erol, K., Hendler, J., & Nau, D. (1996). Complexity Results for HTN Planning. *Annals of Mathematics and Artificial Intelligence*, *18*, 69–93.

Fikes, R., Hart, P., & Nilsson, N. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, *3*(4), 251–288.

Fikes, R., & Nilsson, N. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, *2*(3/4), 189–208.

Giménez, O., & Jonsson, A. (2008). The Complexity of Planning Problems With Simple Causal Graphs. *Journal of Artificial Intelligence Research*, *31*, 319–351.

Hickmott, S., Rintanen, J., Thiébaux, S., & White, L. (2007). Planning via Petri Net Unfolding. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1904–1911.

Hu, Y., & De Giacomo, G. (2013). A Generic Technique for Synthesizing Bounded Finite-State Controllers. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS)*.

Jonsson, A. (2009). The Role of Macros in Tractable Planning. *Journal of Artificial Intelligence Research*, *36*, 471–511.

Korf, R. (1987). Planning as Search: A Quantitative Approach. *Artificial Intelligence*, *33(1)*, 65–88.

LaValle, S. (2006). *Planning Algorithms*. Cambridge Press.

Lempel, A., & Ziv, J. (1976). On the Complexity of Finite Sequences. *IEEE Transactions on Information Theory*, *22*(1), 75–81.

McAllester, D., & Rosenblitt, D. (1991). Systematic Nonlinear Planning. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI)*, pp. 634–639.

Mealy, G. (1955). A Method to Synthesizing Sequential Circuits. *Bell System Technical Journal*, *34*, 1045–1079.

Minton, S. (1985). Selectively Generalizing Plans for Problem-Solving. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 596–599.

Nau, D., Ilghami, O., Kuter, U., Murdock, J., Wu, D., & Yaman, F. (2003). SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, *20*, 379–404.

Newton, M., Levine, J., Fox, M., & Long, D. (2007). Learning Macro-Actions for Arbitrary Planners and Domains. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 256–263.

Subramanian, H., & Shankar, P. (2005). Compressing XML Documents Using Recursive Finite State Automata. In *Proceedings of the 10th International Conference on Implementation and Application of Automata (CIAA)*, pp. 282–293.

Toropila, D., & Barták, R. (2010). Using Finite-State Automata to Model and Solve Planning Problems. In *Proceedings of the 11th Italian AI Symposium on Artificial Intelligence (AI*IA)*, pp. 183–189.

Winner, E., & Veloso, M. (2003). DISTILL: Towards Learning Domain-Specific Planners by Example. In *Proceedings of the 20th International Conference on Machine Learning (ICML)*, pp. 800–807.

Zhong, H., & Wonham, M. (1990). On the Consistency of Hierarchical Supervision in Discrete-Event Systems. *IEEE Transactions on Automatic Control*, *35*(10), 1125–1134.

Zipstein, M. (1992). Data Compression with Factor Automata. *Theoretical Computer Science*, *92*(1), 213–221.