Journal of Artificial Intelligence Research 62 (2018) 755-797 Submitted 05/18; published 08/18

Computing Hierarchical Finite State Controllers with Classical Planning

Javier Segovia-Aguas JAVIER.SEGOVIA@UPF.EDU
Dept. Information and Communication Technologies
Universitat Pompeu Fabra
Roc Boronat 138, 08018 Barcelona, Spain

Sergio Jiménez SERJICE@DSIC.UPV.ES
Dept. Sistemas Informáticos y Computación
Universitat Politècnica de València
Camino de Vera s/n. 46022 Valencia, Spain

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

Abstract

Finite State Controllers (FSCs) are an effective way to compactly represent *sequential plans*. By imposing appropriate conditions on transitions, FSCs can also represent *generalized plans* (plans that solve a range of planning problems from a given domain). In this paper we introduce the concept of *hierarchical FSCs* for planning by allowing controllers to call other controllers. This call mechanism allows hierarchical FSCs to represent generalized plans more compactly than individual FSCs, to compute controllers in a modular fashion or even more, to compute recursive controllers. The paper introduces a classical planning compilation for computing hierarchical FSCs that solve challenging generalized planning tasks. The compilation takes as input a finite set of classical planning problems from a given domain. The output of the compilation is a single classical planning problem whose solution induces: (1) a hierarchical FSC and (2), the corresponding validation of that controller on the input classical planning problems.

1. Introduction

Finite State Controllers (FSCs) are a compact and effective representation commonly used in AI; prominent examples include robotics (Brooks, 1989) and video-games (Buckland, 2004). In planning, FSCs offer two main benefits: solution *compactness* (Bäckström, Jonsson, & Jonsson, 2014); and solution *generalization*. FSCs can represent *generalized plans* that solve a range of different planning problems with a common structure, arbitrarily large problems, as well as problems with partial observability and non-deterministic actions (Bonet, Palacios, & Geffner, 2010; Hu & Levesque, 2011; Srivastava, Immerman, Zilberstein, & Zhang, 2011; Hu & De Giacomo, 2013).

Even FSCs have limitations, however. Consider the problem of traversing all nodes of a binary tree (as in Figure 1). A classical plan for this task is an action sequence whose length is linear in the number of nodes, and hence exponential in the depth of the tree. In contrast, the recursive definition of the Depth-First Search (DFS) algorithm only requires a few lines of code and it is able to traverse any tree, no matter the tree size. Standard FSCs cannot however implement recursion, and the iterative definition of the DFS algorithm is considerably more complicated, involving an external data structure.

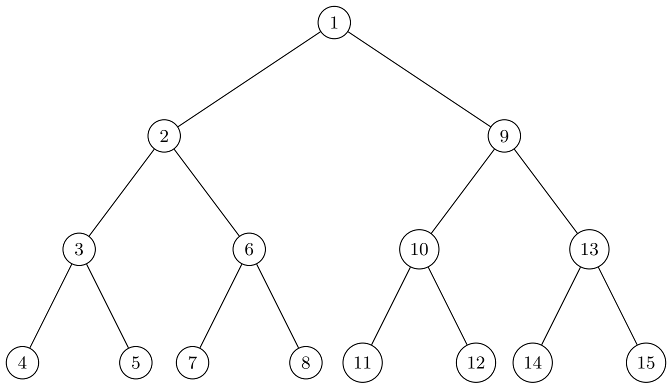


Figure 1: Binary tree with fifteen nodes. Nodes in the tree are labeled following a DFS order.

In this paper we introduce *hierarchical FSCs*, a novel formalism for representing and computing compact and generalized planning solutions. Our formalism extends standard FSCs for planning in three ways. First, a hierarchical FSC can involve multiple individual FSCs. Second, each FSC can call other FSCs. Third, each FSC has a parameter list, and when an FSC is called, it is necessary to specify the arguments assigned to the parameters. As a special case, our formalism makes it possible to implement recursion by allowing an FSC to call itself with different arguments.

To illustrate this idea Figure 2 shows a *hierarchical FSC*, named $C[n]$, that implements a recursive DFS traversal of binary trees. The controller $C[n]$ is pictured as a directed graph whose nodes mount to the different controller states. Edges in the graph represent the transitions of the controller states. Each edge is tagged with a *condition/action* label, that denotes the condition under which the action is taken. $C[n]$ has a lone parameter that represents the current node of the binary tree. Condition $\mathsf{isNull}(n)$ tests whether n has no assigned value, $\mathsf{isVisited}(n)$ tests whether n is an already visited node, while a hyphen '$-$' indicates that the corresponding transition fires no matter what. Action $\mathsf{visit}(n)$ visits node n, while $\mathsf{copyL}(n, m)$ and $\mathsf{copyR}(n, m)$ assign the left and right child of node n to m, respectively. Action $\mathsf{call}(m)$ is a recursive call to the controller itself, assigning argument m to the only parameter of the controller and restarting execution from the initial controller state q_0.

Intuitively the controller $C[n]$ works as follows. The controller state q_3 is a *terminal state* and the action $\mathsf{visit}(n)$ (on the transition to q_3) is in fact not needed and could be removed. The hierarchical FSC of Figure 2 is automatically generated by our approach, so we present conditions and actions exactly as they appear. By repeatedly assigning the right child of n to n itself (following the transition $\mathsf{isVisited}(n)/\mathsf{copyR}(n, n)$) the controller visits all nodes on the rightmost branch of the tree until n has no value (or after visiting a leaf). Moreover, by assigning the left child of n to *child* (following the transition $-/\mathsf{copyL}(n, child)$) and

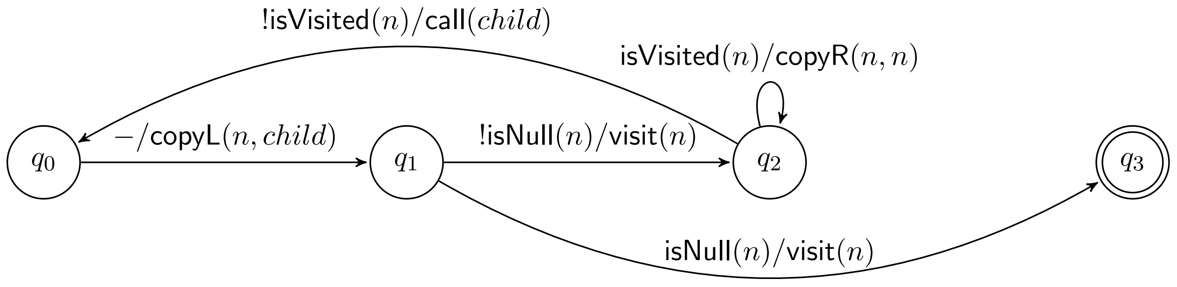


Figure 2: Hierarchical FSC, named $C[n]$, that implements a recursive DFS traversal of a binary tree. Its lone parameter represents the current node of the binary tree.

making a recursive call (following the transition !isVisited(n)/call($child$)) the controller is recursively executed on all the left sub-trees.

Compared to previous work on the computation of FSCs for planning, the contributions of our approach are:

1. A reformulation of *FSCs for planning* that allows us to synthesize the observation function of a FSC in addition to its transition and output functions.

2. A formal definition of *hierarchical FSCs for planning* that allows controllers to call other controllers and that includes recursion as a special case.

3. A novel method for computing hierarchical FSCs for planning. The method is a compilation that takes as input a set of classical planning problems from a given domain. The output of the compilation is a single classical planning problem whose solution induces, a hierarchical FSC and the corresponding validation of that controller on the input planning problems.

A first description of the *hierarchical FSC* formalism previously appeared in our conference paper (Segovia-Aguas, Jiménez, & Jonsson, 2016b). Compared to that work, this paper includes the following novel material:

- An study of the relation of *Mealy machines* and FSCs for classical and generalized planning.

- Formal and empirical comparisons of *hierarchical FSCs* with the *planning program* formalism for generalized planning (Segovia-Aguas, Jiménez, & Jonsson, 2016a).

- An exhaustive empirical evaluation of *hierarchical FSCs* with additional results in new benchmarks and with various planners.

The rest of the paper is organized as follows. Section 2 introduces the planning model we follow in this work and presents our definition of FSCs for planning, which is slightly adapted from previous work. Section 3 describes our compilation for computing FSCs that solve classical planning problems and generalized planning problems. Section 4 formalizes *hierarchical FSCs* and extends our compilation to compute controllers of this kind. Section 5 evaluates our approach and reports the empirical performance of our compilation. Finally, Section 6 describes related work and we conclude with a discussion in Section 7.

2. Background

This section defines the planning model we follow in the rest of the paper and presents previous formalisms for FSCs in the context of planning.

2.1 Classical Planning with Conditional Effects

We use the model of *classical planning with conditional effects* because it can compactly define actions whose precise effects depend on the state where the action is executed. The support of conditional effects is now a requirement of the *International Planning Competition* (Vallati et al., 2015) and current classical planners cope with conditional effects without compiling them away (Röger, Pommerening, & Helmert, 2014).

We use F to denote the set of *fluents* (propositional variables) describing a state. A literal l is a valuation of a fluent $f \in F$, i.e. $l = f$ or $l = \neg f$. A set of literals L on F represents a partial assignment of values to fluents (WLOG we assume that L does not assign conflicting values to any fluent). Given L, $\neg L = \{\neg l : l \in L\}$ is the complement of L. Finally, we use $\mathcal{L}(F)$ to denote the set of all literal sets on F, i.e. all partial assignments of values to fluents.

A *state* s is a set of literals such that $|s| = |F|$, i.e. a total assignment of values to fluents. The number of states is then $2^{|F|}$. Explicitly including negative literals $\neg f$ in states simplifies subsequent definitions, but we often abuse notation by defining a state s only in terms of the fluents that are true in s, as is common in Strips planning. In that case, since s is a total assignment, all fluents not explicitly mentioned as true are assumed to be false.

A *classical planning problem* with conditional effects is a tuple $P = \langle F, A, I, G\rangle$, where F is a set of fluents, A is a set of actions, I is an initial state and G is a goal condition, i.e. a set of literals. Each action $a \in A$ has a set of literals $\mathsf{pre}(a)$, called the *precondition*, and a set of *conditional effects*, $\mathsf{cond}(a)$. Each conditional effect $C \rhd E \in \mathsf{cond}(a)$ is composed of sets of literals C (the condition) and E (the effect). Action a is *applicable* in state s if and only if $\mathsf{pre}(a) \subseteq s$, and the resulting set of *triggered effects* is

$$\mathsf{eff}(s,a) = \bigcup_{C \rhd E \in \mathsf{cond}(a), C \subseteq s} E,$$

i.e. effects whose conditions hold in s. The result of applying a in s is the *successor state* $\theta(s,a) = (s \setminus \neg\mathsf{eff}(s,a)) \cup \mathsf{eff}(s,a)$. We often assume that fluent set F of a planning problem P is induced by a set of predicates Ψ and a set of objects Ω, as specified in PDDL (McDermott et al., 1998). Likewise, the action set A is induced by a set of action schemas $\mathcal{A}$ and Ω.

A *solution* for P can be specified using different representation formalisms, e.g. a sequence of actions, a partially ordered plan, a policy, an FSC, etc. Each solution model has its own syntax and semantics, and defines the space of solutions that can be computed as well as the worst case computation complexity. Here we define a *sequential plan* for P as an action sequence $\pi = \langle a_1, \ldots, a_n\rangle$ that induces a state sequence $\langle s_0, s_1, \ldots, s_n\rangle$ such that $s_0 = I$ and, for each i such that $1 \leq i \leq n$, a_i is applicable in s_{i-1} and generates the successor state $s_i = \theta(s_{i-1}, a_i)$. The plan π *solves* P if and only if $G \subseteq s_n$, i.e. if the goal condition is satisfied following the application of π in I.

Next we define our formalism for FSCs that is suitable for representing solutions to both classical and generalized planning.

2.2 Finite State Controllers for Planning

FSCs for planning in the literature (Bonet et al., 2010; Hu & De Giacomo, 2013) are similar to *transducers* that, in addition to processing input, produce output. We first define transducers and then explain how FSCs for planning are derived from transducers.

Formally, a finite state transducer or *Mealy machine* is a tuple $\Delta = \langle Q, q_0, Q_\perp, \Sigma, \Lambda, T, \Gamma \rangle$:

- Q is a finite set of controller states where $q_0 \in Q$ is the *initial controller state* and $Q_\perp \subseteq Q$ is the subset of *terminal controller states*,
- Σ is a finite set of input symbols called the *input alphabet*,
- Λ is a finite set of output symbols called the *output alphabet*,
- $T : Q \times \Sigma \rightarrow Q$ is a *transition function* mapping pairs of a controller state and an input symbol to the corresponding next state,
- $\Gamma : Q \times \Sigma \rightarrow \Lambda$ is an *output function* mapping pairs of a controller state and an input symbol to the corresponding output symbol,

When in $q \in Q$, on receiving input $\sigma \in \Sigma$, the transducer Δ transitions to $q' = T(q, \sigma)$ and outputs symbol $\lambda = \Gamma(q, \sigma)$. This process starts with $q = q_0$ and it is repeated for $q = q'$ until $q' \in Q_\perp$, is a terminal controller state, or until the sequence of input symbols is exhausted.

Given a classical planning problem with conditional effects $P = \langle F, A, I, G \rangle$, an FSC for P is a pair $C = (\Delta, O)$ of a transducer $\Delta = \langle Q, q_0, Q_\perp, \Sigma, A, T, \Gamma \rangle$ and an *observation function* $O : 2^F \rightarrow \Sigma$. The *observation function* maps the current planning state to an *observation* i.e., a symbol in the input alphabet of the transducer. The transitions of the transducer Δ do not rely then on an external input, but rather on the current planning state. On the other hand, the output alphabet of this particular transducer is given by the set of actions A in the classical planning problem P.

The *world state* of an FSC for P is a pair (q, s) that consists of a controller state $q \in Q$ and a planning state s, with the initial world state given by (q_0, I). Given C, an FSC for P, and a world state (q, s), then C transitions to a new world state (q', s') that is computed as follows:

1. First we retrieve the observation $O(s) \in \Sigma$ symbol that is associated with the current planning state s.
2. Then we compute the new controller state $q' = T(q, O(s))$ and the planning action $a \in A$ to apply next, $a = \Gamma(q, O(s))$.
3. Finally we compute the new planning state $s' = \theta(s, a)$ that results from applying action a in the planning state s.

We use $(q, s) \rightarrow_C (q', s')$ to denote the transition from (q, s) to (q', s'), and we use $(q_0, s_0) \rightarrow_C^k (q_k, s_k)$ to denote a sequence of k such transitions. For $(q_0, s_0) \rightarrow_C^k (q_k, s_k)$ to be *well-defined*, all intermediate actions have to be applicable, i.e. $\mathsf{pre}(a_i) \subseteq s_{i-1}$ for each $a_i = \Gamma(q_{i-1}, O(s_{i-1}))$, $1 \leq i \leq k$. Each transition $T(q, O(s))$ is associated with a single observation of the current planning state however, FSCs for planning can represent expressive state queries including the `no-op` action in A. The `no-op` action does not modify the planning state and affects only to the next controller state which allows to concatenate state queries like in *decision trees*.

In previous work (Bonet et al., 2010; Hu & De Giacomo, 2013), the set $Q_\perp$ of terminal controller states is empty, and termination is implicitly defined as $G \subseteq s$, i.e. the goal condition G has to hold in the planning state s of the current world state (q, s). Hence a given controller C solves P if and only if there exists a well-defined transition sequence $(q_0, I) \rightarrow_C^k (q_k, s_k)$, $k \geq 0$, such that $G \subseteq s_k$. This means that goal achievement has to be tested after executing every action for checking termination. Further, the authors assume that the *observation function* is given as input to synthesize a FSC that solves a given contingent planning problem.

Hu and De Giacomo (2013) define a *generalized planning problem* $\mathcal{P} = \{P_1, \ldots, P_T\}$ as a finite set of classical planning problems $P_t = \langle F_t, A, I_t, G_t \rangle$, $1 \leq t \leq T$, that share the same action set A. An FSC for a generalized planning problem $\mathcal{P}$ is a pair $C = (\Delta, \mathcal{O})$ of a transducer $\Delta = \langle Q, q_0, Q_\perp, \Sigma, A, T, \Gamma \rangle$ and a set $\mathcal{O}$ of observation functions $O_t : 2^{F_t} \rightarrow \Sigma$, $1 \leq t \leq T$. The preconditions and conditional effects of actions in A could have different definitions for different planning problems in $\mathcal{P}$. An FSC $C = (\Delta, \mathcal{O})$ solves $\mathcal{P}$ if and only if $C_t = (\Delta, O_t)$ solves each P_t, $1 \leq t \leq T$. The authors show how to synthesize the transition function T and output function Γ given all other elements of C (including the observation functions $O_1, \ldots, O_T$) such that C solves $\mathcal{P}$.

3. Computing Finite State Controllers for Planning

This section details our novel formalism of *FSCs for planning* and presents our compilation for computing FSCs with off-the-shelf planners. The compilation takes as input a classical planning instance, and outputs another classical planning instance whose solution induces (1), an FSC plus (2), the validation of that FSC on the input planning instance. The output of the compilation is formalized in the standard PDDL language, so off-the-shelf classical planners can be used to compute the FSCs.

3.1 A Novel Definition of FSCs for Planning

We reformulate the previous definition of *FSCs for planning*. Given a classical planning problem with conditional effects $P = \langle F, A, I, G \rangle$, an FSC for P is a pair $C = (\Delta, \Phi)$ of a transducer $\Delta = \langle Q, q_0, q_\perp, \{0, 1\}, A, T, \Gamma \rangle$ and a function $\Phi : Q \rightarrow F$ that maps a controller state into a fluent from the given planning problem.

Compared to the previous definition of FSCs for planning, this novel formalism introduced the following three modifications:

1. There is a single *terminal controller state* $q_\perp$. The reason for including an explicit terminal controller state $q_\perp$ is that we will later extend our definition to hierarchies

of FSCs in which the goal G is not necessarily satisfied when an FSC terminates its execution. In addition, including an explicit terminal state allows us to use FSC as *acceptors* (Segovia-Aguas, Jiménez, & Jonsson, 2017b, 2017a).

2. The *input alphabet* is simply $\{0,1\}$ so Φ induces an observation function $O : Q \times 2^F \to \{0,1\}$ that maps pairs of a controller state and a planning state, into either 0 or 1. Formally, $O(q,s) = \Phi(q) \in s$, where $\Phi(q) \in s$ is interpreted as a test whose outcome:
    - Equals 1, iff the fluent $\Phi(q)$ is *true* in the planning state s.
    - Equals 0, iff the fluent $\Phi(q)$ is *false* in the planning state s.

   When $\Phi(q)$ is *static* (i.e. the value of this fluent in F is not changed by any action in A) then the outcome of $O(q,s)$, either 0 or 1, is constant.

3. *The observation function O is defined on controller states in addition to planning states.* We aim to synthesize the observation function O in addition to T and Γ (in contrast to previous work in which O is given). To keep the space of observation functions tractable, we restricted ourselves to the simplest observation set $\{0,1\}$. Further, we only consider observation functions that use the Φ mapping to test the truth value of a single fluent. Therefore synthesizing O is equivalent to synthesizing Φ (there is nothing however, that prevents $\Phi(q)$ to be a *derived fluent* that is, a fluent that holds when an arbitrary formula over primitive fluents holds).

The execution model of FSCs for planning is the same as before but now, we say that an FSC $C = (\Delta, \Phi)$ solves P if and only if there exists a well-defined transition sequence $(q_0, I) \to_C^k (q_\perp, s_k)$, $k \geq 0$, such that $G \subseteq s_k$, where $q_\perp$ is the terminal controller state. The execution of an FSC on a planning problem $P = \langle F, A, I, G\rangle$ can fail for any of the following three reasons:

1. The execution terminates in a world state $(q_\perp, s_k)$ but the goal condition does not hold, i.e. $G \not\subseteq s_k$.

2. For some world state (q_i, s_i) with $0 \leq i \leq k$, the action $a_i = \Gamma(q_i, O(q_i, s_i))$ cannot be applied because the precondition of a_i does not hold in s_i, i.e. $\mathsf{pre}(a_i) \not\subseteq s_i$.

3. The execution enters an infinite loop that never reaches the terminal state $q_\perp$.

To illustrate our new definition of FSCs for planning, we show an example of a controller for traversing a *linked list*. We model this task as a classical planning problem $P = \langle F, A, I, G\rangle$, where F contains the following fluents:

- For each pair of list nodes x, y, a fluent $\mathsf{succ}(x,y)$ identifying y as the successor node of x in the linked list.

- For each list node x fluents $\mathsf{visited}(x)$, denoting that x has been visited and $\mathsf{end}(x)$, denoting that node x is the end of the list.

- Fluents $\mathsf{assign}(n,x)$, denoting that variable n points to the list node x, and the derived fluent $\mathsf{isEnd}(n) \equiv \exists x\ \mathsf{assign}(n,x) \wedge \mathsf{end}(x)$, denoting that variable n points to the end of the linked list.

The action set A contains a `no-op` action, named $a_\perp$, such that $\mathsf{pre}(a_\perp) = \mathsf{cond}(a_\perp) = \emptyset$ as well as the following two kinds of actions:

- $\mathsf{visit}(n)$, that mark the list node assigned to n as visited:

$$\begin{aligned}\mathsf{pre}(\mathsf{visit}(n)) &= \emptyset,\\ \mathsf{cond}(\mathsf{visit}(n)) &= \{\{\mathsf{assign}(n,x)\} \rhd \{\mathsf{visited}(x)\} : \forall x\}.\end{aligned}$$

- $\mathsf{step}(n)$, that move n to the next node in the linked list:

$$\begin{aligned}\mathsf{pre}(\mathsf{step}(n)) &= \emptyset,\\ \mathsf{cond}(\mathsf{step}(n)) &= \{\{\mathsf{assign}(n,x), \mathsf{succ}(x,y)\} \rhd \{\neg\mathsf{assign}(n,x), \mathsf{assign}(n,y)\} : \forall x,y\}.\end{aligned}$$

For a linked list of length k, the initial state I and goal condition G are defined as follows:

$$\begin{aligned}I &= \{\mathsf{assign}(n,x_0), \mathsf{succ}(x_0,x_1), \ldots, \mathsf{succ}(x_{k-1},x_k), \mathsf{end}(x_k)\},\\ G &= \{\mathsf{visited}(x_0), \ldots, \mathsf{visited}(x_{k-1})\}.\end{aligned} \tag{1}$$

Figure 3 shows a three-state FSC that solves a traversing list planning problem P. The edge (q_0, q_2) with label $\mathsf{isEnd}(n)/a_\perp$ encodes that $\Phi(q_0) = \mathsf{isEnd}(n)$, $T(q_0, 1) = q_2$, $\Gamma(q_0, 1) = a_\perp$, i.e. encodes the transition and associated action when $\mathsf{isEnd}(n)$ holds in the current planning state. The edge (q_0, q_1) with label $!\mathsf{isEnd}(n)/\mathsf{visit}(n)$ encodes the transition and action when $\mathsf{isEnd}(n)$ does not hold, i.e. $T(q_0, 0) = q_1$ and $\Gamma(q_0, 0) = \mathsf{visit}(n)$. The edge (q_1, q_0) with label $-/\mathsf{step}(n)$ denotes that, when in q_1, the transition and action are always the same no matter the current planning state.

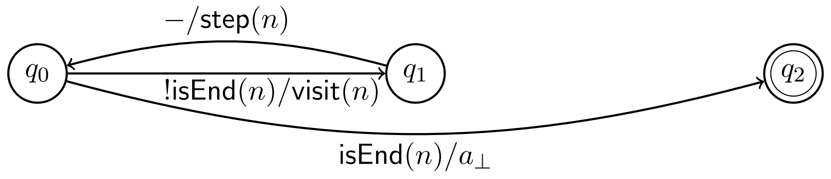

$-/\mathsf{step}(n)$
q_0
q_1
q_2
$!\mathsf{isEnd}(n)/\mathsf{visit}(n)$
$\mathsf{isEnd}(n)/a_\perp$

Figure 3: FSC for the task of traversing a linked list.

3.2 Computing FSCs for Classical Planning

This section describes our compilation for computing an FSC that solves a given classical planning problem. The idea behind the compilation is to include the current controller state, as part of the planning state, and to define actions of two types: *program actions* (that program the three functions Φ, T and Γ of the FSC) and *execute actions*, that simulate the execution of the FSC evaluating its programmed functions Φ, T and Γ.

Formally, the compilation takes as input a classical planning problem $P = \langle F, A, I, G\rangle$ and a bound n on the number of controller states, and outputs another classical planning problem $P_n = \{F_n, A_n, I_n, G_n\}$. Any plan that solves P_n generates an FSC $C = (\langle Q, q_0, q_\perp, \{0,1\}, A, T, \Gamma\rangle, \Phi)$ and validates that C solves P.

We first set $Q = \{q_0, \ldots, q_n\}$ and $q_\perp \equiv q_n$. The functions Φ, T and Γ are not defined on $q_\perp$, so we say that C has n controller states (even though $|Q| = n+1$). Now we proceed to define the compilation.

The set of fluents $F_n = F \cup F_{fun} \cup F_{aux}$, where F_{fun} contains the fluents needed to encode the functions Φ, T and Γ:

- For each $q \in Q$ and $f \in F$, a fluent cond_q^f that holds iff f is the condition associated with q, i.e. if $\Phi(q) = f$.
- For each $q, q' \in Q$ and $b \in \{0,1\}$, a fluent $\mathsf{succ}_{q,q'}^b$ that holds iff $T(q,b) = q'$.
- For each $q \in Q$, $b \in \{0,1\}$ and $a \in A$, a fluent $\mathsf{act}_{q,a}^b$ that holds iff $\Gamma(q,b) = a$.
- For each $q \in Q$ and $b \in \{0,1\}$, fluents nocond_q, nosucc_q^b and noact_q^b that hold iff we have yet to program the functions Φ, T and Γ, respectively, in q and b.

Moreover, F_{aux} contains the following fluents:

- For each $q \in Q$, a fluent cs_q that holds iff q is the current controller state.
- Fluents evl and app that hold iff we are done evaluating the condition or applying the action corresponding to the current controller state, and fluents o^0 and o^1 representing the outcome of the evaluation (0 or 1).

The initial state and goal condition equal $I_n = I \cup \{\mathsf{cs}_{q_0}\} \cup \{\mathsf{nocond}_q, \mathsf{noact}_q^b, \mathsf{nosucc}_q^b : q \in Q, b \in \{0,1\}\}$ and $G_n = G \cup \{\mathsf{cs}_{q_n}\}$. Finally, the set of actions A_n replaces the actions in A with the following actions:

- For each $q \in Q$ and $f \in F$, an action pcond_q^f for programming $\Phi(q) = f$:

$$\begin{aligned}\mathsf{pre}(\mathsf{pcond}_q^f) &= \{\mathsf{cs}_q, \mathsf{nocond}_q\},\\ \mathsf{cond}(\mathsf{pcond}_q^f) &= \{\emptyset \rhd \{\neg\mathsf{nocond}_q, \mathsf{cond}_q^f\}\}.\end{aligned}$$

- For each $q \in Q$ and $f \in F$, an action econd_q^f that evaluates the condition of the current controller state:

$$\begin{aligned}\mathsf{pre}(\mathsf{econd}_q^f) &= \{\mathsf{cs}_q, \mathsf{cond}_q^f, \neg\mathsf{evl}\},\\ \mathsf{cond}(\mathsf{econd}_q^f) &= \{\emptyset \rhd \{\mathsf{evl}\}, \{\neg f\} \rhd \{\mathsf{o}^0\}, \{f\} \rhd \{\mathsf{o}^1\}\}.\end{aligned}$$

- For each $q \in Q$, $b \in \{0,1\}$ and $a \in A$, an action $\mathsf{pact}_{q,a}^b$ for programming $\Gamma(q,b) = a$:

$$\begin{aligned}\mathsf{pre}(\mathsf{pact}_{q,a}^b) &= \mathsf{pre}(a) \cup \{\mathsf{cs}_q, \mathsf{evl}, \mathsf{o}^b, \mathsf{noact}_q^b\},\\ \mathsf{cond}(\mathsf{pact}_{q,a}^b) &= \{\emptyset \rhd \{\neg\mathsf{noact}_q^b, \mathsf{act}_{q,a}^b\}\}.\end{aligned}$$

- For each $q \in Q$, $b \in \{0,1\}$ and $a \in A$, an action $\mathsf{eact}_{q,a}^b$ that applies the action of the current controller state:

$$\begin{aligned}\mathsf{pre}(\mathsf{eact}_{q,a}^b) &= \mathsf{pre}(a) \cup \{\mathsf{cs}_q, \mathsf{evl}, \mathsf{o}^b, \mathsf{act}_{q,a}^b, \neg\mathsf{app}\},\\ \mathsf{cond}(\mathsf{eact}_{q,a}^b) &= \mathsf{cond}(a) \cup \{\emptyset \rhd \{\mathsf{app}\}\}.\end{aligned}$$

- For each $q, q' \in Q$ and $b \in \{0,1\}$, an action $\mathsf{psucc}^b_{q,q'}$ for programming $T(q,b) = q'$:

$$\mathsf{pre}(\mathsf{psucc}^b_{q,q'}) = \{\mathsf{cs}_q, \mathsf{evl}, \mathsf{o}^b, \mathsf{app}, \mathsf{nosucc}^b_q\},$$
$$\mathsf{cond}(\mathsf{psucc}^b_{q,q'}) = \{\emptyset \rhd \{\neg\mathsf{nosucc}^b_q, \mathsf{succ}^b_{q,q'}\}\}.$$

- For each $q, q' \in Q$ and $b \in \{0,1\}$, an action $\mathsf{esucc}^b_{q,q'}$ that transitions to the next controller state:

$$\mathsf{pre}(\mathsf{esucc}^b_{q,q'}) = \{\mathsf{cs}_q, \mathsf{evl}, \mathsf{o}^b, \mathsf{app}, \mathsf{succ}^b_{q,q'}\},$$
$$\mathsf{cond}(\mathsf{esucc}^b_{q,q'}) = \{\emptyset \rhd \{\neg\mathsf{cs}_q, \neg\mathsf{evl}, \neg\mathsf{o}^b, \neg\mathsf{app}, \mathsf{cs}_{q'}\}\}.$$

Actions pcond^f_q, $\mathsf{pact}^b_{q,a}$ and $\mathsf{psucc}^b_{q,q'}$ program the three functions Φ, T and Γ, respectively that encode the possible transitions of the controller, while econd^f_q, $\mathsf{eact}^b_{q,a}$ and $\mathsf{esucc}^b_{q,q'}$ execute the corresponding function. Fluents evl and app control the order of the execution such that Φ is always executed first, then Γ, and finally T.

We remark that the functions Φ, T and Γ are programmed *online*: actions pcond^f_q, $\mathsf{pact}^b_{q,a}$ and $\mathsf{psucc}^b_{q,q'}$ are only applicable when the current controller state is q and the current observation is b, respectively. As a consequence, a plan that solves P_n may not always program Φ, T and Γ for all the controller states in Q, in which case the resulting FSC ignores the unprogrammed controller states. One benefit of this *online* approach is that we can immediately check whether the precondition of a given action $a \in A$ holds (note that $\mathsf{pre}(a)$ is part of the precondition of the action $\mathsf{pact}^b_{q,a}$ for programming a).

The compilation P_n also includes a mechanism for symmetry breaking, which we proceed to describe. For simplicity we excluded this mechanism from the above definition of P_n. To program a transition to a controller state q' using an action $\mathsf{psucc}^b_{q,q'}$, q' has to be *available*. Initially, only q_1 and q_n are available. When we visit q_1 for the first time, q_2 becomes available, etc. It is straightforward to implement this mechanism using fluents $\mathsf{available}_q$ and conditional effects of $\mathsf{esucc}^b_{q,q'}$. With this mechanism in place, we avoid generating multiple permutations of the same FSC that only rename the controller states.

3.3 Example

We show how our P_n compilation computes the three-state controller of Figure 3. Recall that the initial state output by our compilation is $I_n = I \cup \{\mathsf{cs}_{q_0}\} \cup \{\mathsf{nocond}_q, \mathsf{noact}^b_q, \mathsf{nosucc}^b_q : q \in Q, b \in \{0,1\}\}$. In this example, where I is given by the expression (1), the only applicable actions at I_n are $\mathsf{pcond}^f_{q_0}$ for programming the value of $\Phi(q_0)$. To produce the FSC in Figure 3, the planner chooses $f = \mathsf{isEnd}(\mathsf{n})$. The effect of action $\mathsf{pcond}^f_{q_0}$ is $\{\neg\mathsf{nocond}_{q_0}, \mathsf{cond}^f_{q_0}\}$.

In the resulting state, the only applicable action is $\mathsf{econd}^f_{q_0}$. Since $\mathsf{isEnd}(n)$ is not true in the current state, the effect of $\mathsf{econd}^f_{q_0}$ is $\{\mathsf{evl}, \mathsf{o}^0\}$. At this state, the only applicable actions are $\mathsf{pact}^0_{q_0,a}$, $a \in A$, for programming the value of $\Gamma(q_0, 0)$. To produce the FSC in Figure 3, the planner chooses $a = \mathsf{visit}(n)$. The effect of $\mathsf{pact}^0_{q_0,a}$ is to add fluent $\mathsf{act}^0_{q_0,a}$, causing $\mathsf{eact}^0_{q_0,a}$ to be the only applicable action. In turn, the effect of $\mathsf{eact}^0_{q_0,a}$ is $\{\mathsf{visited}(x_0), \mathsf{app}\}$, where $\mathsf{visited}(x_0)$ is the effect of a and app indicates that we have applied action $a = \Gamma(q_0, 0)$.

Now the only applicable actions are $\mathsf{psucc}^0_{q_0,q}$, $q \in Q$, for programming the value of $T(q_0, 0)$. In the case of computing the FSC of Figure 3, the planner chooses $q = q_1$. The effect of $\mathsf{psucc}^0_{q_0,q}$ is to add fluent $\mathsf{succ}^0_{q_0,q}$, causing $\mathsf{esucc}^0_{q_0,q}$ to be the only applicable action. The effect of $\mathsf{esucc}^0_{q_0,q}$ is $\{\neg\mathsf{cs}_{q_0}, \neg\mathsf{evl}, \neg\mathsf{o}^0, \neg\mathsf{app}, \mathsf{cs}_{q_1}\}$, representing a transition from q_0 to q_1 and making actions $\mathsf{pcond}^f_{q_1}$ applicable. So far, the applied action sequence is:

$$\langle \mathsf{pcond}^{\mathsf{isEnd}(n)}_{q_0}, \mathsf{econd}^{\mathsf{isEnd}(n)}_{q_0}, \mathsf{pact}^0_{q_0,\mathsf{visit}(n)}, \mathsf{eact}^0_{q_0,\mathsf{visit}(n)}, \mathsf{psucc}^0_{q_0,q_1}, \mathsf{esucc}^0_{q_0,q_1} \rangle.$$

Next, to program and simulate the transition from q_1 to q_0, the planner chooses a similar action sequence. Here, any static fluent can be used to produce the FSC in Figure 3, e.g. $\Phi(q_1) = \mathsf{succ}(x_0, x_2)$, since this transition fires no matter what.

$$\langle \mathsf{pcond}^{\mathsf{succ}(x_0,x_2)}_{q_1}, \mathsf{econd}^{\mathsf{succ}(x_0,x_2)}_{q_1}, \mathsf{pact}^0_{q_1,\mathsf{step}(n)}, \mathsf{eact}^0_{q_1,\mathsf{step}(n)}, \mathsf{psucc}^0_{q_1,q_0}, \mathsf{esucc}^0_{q_1,q_0} \rangle.$$

Since $\Phi(q)$, $\Gamma(q, 0)$ and $T(q, 0)$ are already programmed for $q \in \{q_0, q_1\}$, to simulate now the transition from q_0 to q_1 and back to q_0 we only need *execute* actions (no *programming* actions are necessary here):

$$\langle \mathsf{econd}^{\mathsf{isEnd}(n)}_{q_0}, \mathsf{eact}^0_{q_0,\mathsf{visit}(n)}, \mathsf{esucc}^0_{q_0,q_1}, \mathsf{econd}^{\mathsf{succ}(x_0,x_2)}_{q_1}, \mathsf{eact}^0_{q_1,\mathsf{step}(n)}, \mathsf{esucc}^0_{q_1,q_0} \rangle.$$

The result is to visit x_1 and move n from pointing to x_1 to pointing to x_2. This cycle repeats until the effect of $\mathsf{econd}^{\mathsf{isEnd}(n)}_{q_0}$ is $\{\mathsf{evl}, \mathsf{o}^1\}$, indicating that fluent $\mathsf{isEnd}(n)$ is true. When this happens we can program and simulate the transition from q_0 to q_2 using the following action sequence:

$$\langle \mathsf{pact}^1_{q_0,a_\perp}, \mathsf{eact}^1_{q_0,a_\perp}, \mathsf{psucc}^1_{q_0,q_2}, \mathsf{esucc}^1_{q_0,q_2} \rangle.$$

Since q_2 is the terminal controller state and all list nodes have been visited, the goal condition G_n is satisfied. Note that G_n is only satisfied after the execution of the programmed FSC guarantees to solve the input planning problem P.

3.4 Properties

Here we analyze the theoretical properties of our P_n compilation for synthesizing an FSC that solves a classical planning task P.

Theorem 1 (Soundness)**.** *Any plan π that solves P_n induces an FSC that solves P.*

Proof. Once π programs the functions Φ, T and Γ of the FSC they cannot be altered. Programming actions pcond^f_q, $\mathsf{pact}^b_{q,a}$ and $\mathsf{psucc}^b_{q,q'}$ delete fluents nocond_q, noact^b_q and nosucc^b_q respectively, and there are no actions in A_n for adding these fluents, which bans the later application of actions pcond^f_q, $\mathsf{pact}^b_{q,a}$ or $\mathsf{psucc}^b_{q,q'}$ for the same values of q and b.

The plan π deterministically executes the FSC (the programmed Φ, T and Γ functions) on the input planning problem P. The only way to change the current controller state from q to q' is to apply a partial action sequence $\langle \mathsf{econd}^f_q, \mathsf{eact}^b_{q,a}, \mathsf{esucc}^b_{q,q'} \rangle$ for some $f \in F$, $a \in A$, $b \in \{0, 1\}$. Because of the corresponding preconditions cond^f_q, $\mathsf{act}^b_{q,a}$ and $\mathsf{succ}^b_{q,q'}$ of these actions, this means that programming actions pcond^f_q, $\mathsf{pact}^b_{q,a}$ and $\mathsf{psucc}^b_{q,q'}$ have to be applied in advance. Further, since cond_q^f, $\mathsf{act}_{q,a}^b$ and $\mathsf{succ}_{q,q'}^b$ are true for at most a single combination of values of f, a and q', this uniquely determines the value of the functions Φ, T and Γ in q and b.

Eventually π executes the programmed FSC on P until it solves P. The subset of fluents $\{\mathsf{cs}_q : q \in Q\} \cup F \subseteq F_n$ represents the current world state (q, s) of an FSC C with controller states $Q = \{q_0, \dots, q_n\}$. The definition of I_n claims that the initial world state is (q_0, I). To satisfy the goal condition G_n, π has to simulate a *well-defined* transition sequence $(q_0, I) \rightarrow_C^k (q_n, s)$, $0 \leq k = n$, such that G holds in s, i.e. $G \subseteq s$. The partial action sequence $\langle \mathsf{econd}_q^f, \mathsf{eact}_{q,a}^b, \mathsf{esucc}_{q,q'}^b \rangle$ precisely simulates a well-defined transition $(q, s) \rightarrow_C (q', s')$ of C. Action econd_q^f adds o^b where $b \in \{0, 1\}$ is the truth value of f in s. Action $\mathsf{eact}_{q,a}^b$ applies the action a in s to obtain a new state $s' = \theta(s, a)$. Finally, action $\mathsf{esucc}_{q,q'}^b$ transitions to controller state q'. This deterministic execution continues until we reach a terminal state (q_n, s) or revisit a world state. If π solves P_n, execution finishes in (q_n, s) and the goal condition G holds in s, which is the definition of the FSC solving P. □

Theorem 2 (Completeness)**.** *If there exists an FSC $C = (\langle Q, q_0, q_\perp, \{0, 1\}, A, T, \Gamma \rangle, \Phi)$ that solves P, there exists a corresponding plan π that solves P_n for each $n \geq |Q| - 1$.*

Proof. By definition, fluents cond_q^f, $\mathsf{act}_{q,a}^b$ and $\mathsf{succ}_{q,q'}^b$ can altogether determine the Φ, T and Γ functions of any FSC (provided that there is an enough amount n of controller states). Therefore, a plan π can be built that programs the functions Φ, Γ and T of any FSC making the appropriate fluents true among cond_q^f, $\mathsf{act}_{q,a}^b$ and $\mathsf{succ}_{q,q'}^b$ using the corresponding actions pcond_q^f, $\mathsf{pact}_{q,a}^b$ and $\mathsf{psucc}_{q,q'}^b$.

The fact that C solves P implies that there exists a sequence of well-defined world transitions $(q_0, I) \rightarrow_C^k (q_n, s)$, $0 \leq k = n$, such that $G \subseteq s$. The execution of the sequence of world transitions of any FSC can be simulated using action sequences of type $\langle \mathsf{econd}_q^f, \mathsf{eact}_{q,a}^b, \mathsf{esucc}_{q,q'}^b \rangle$. Therefore, the plan π can be extended simulating the execution of the functions Φ, Γ and T, starting from (q_0, I) and until reaching a world state (q_n, s) s.t. $G \subseteq s$. This is the definition of π satisfying the goal condition G_n to solve P_n.

□

Our compilation is not complete in the sense that the bound n on the number of controller states may be too small to accommodate an FSC that solves P. For instance, the FSC in Figure 2 cannot be computed if $n < 3$. Larger values of n do not formally affect the completeness of our approach but they do affect its practical performance since the sets F_n and A_n grow with the parameter n and classical planners are sensitive to the input size.

3.5 Computing FSCs for Generalized Planning

This section presents an extension of our compilation to compute *FSCs for generalized planning.* The input to the compilation is no longer a single classical planning problem, but a finite set of classical planning problems that define the generalized planning problem $\mathcal{P} = \{P_1, \dots, P_T\}$. The output of the extended compilation is a classical planning problem whose solution induces an FSC C and validates that C solves every classical planning problem P_t, $1 \leq t \leq T$.

Let us modify the definition of a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ such that now the planning problems $P_t = \langle F, A, I_t, G_t \rangle$, $1 \leq t \leq T$, share the fluent set F in addition to the action set A. Despite the action set is shared, the precise effects of an action is determined by the state where the action is applied due to conditional effects. An FSC for $\mathcal{P}$ is a pair $C = (\Delta, \Phi)$ of a transducer Δ and a mapping Φ, inducing an observation function O which is shared among the planning problems in $\mathcal{P}$. As before, the FSC C solves $\mathcal{P}$ iff C solves each P_t, $1 \leq t \leq T$.

This definition of a generalized planning problem is not as restrictive as it first may appear. We can define a large fluent set F and action set A, and use the initial state I_t of each planning problem P_t to *"switch on/off"* certain elements. This way, we can address planning problems of various sizes in $\mathcal{P}$, as long as F and A are sufficiently large to accommodate the largest planning problem in $\mathcal{P}$. For example, in our list traversal example, we can define a set of list nodes $x_0, \ldots, x_{20}$, and use the fluent $\mathsf{end}(x_5)$ to indicate that the given list has length 5. Even though there are fluents in F and actions in A associated with the list nodes $x_6, \ldots, x_{20}$, these fluents and action are not used in this particular planning problem.

Since the extension of the compilation, $P'_n = \langle F'_n, A'_n, I'_n, G'_n \rangle$, is similar to the original compilation $P_n = \langle F_n, A_n, I_n, G_n \rangle$, we define P'_n in terms of P_n:

- The set of fluents is $F'_n = F_n \cup F_{test}$, where $F_{test} = \{\mathsf{test}_t : 1 \leq t \leq T\}$ models the current classical planning problem in $\mathcal{P}$ that is being solved.

- The set of actions is $A'_n = A_n \cup A_{end}$, where A_{end} includes actions end_t, $1 \leq t < T$, for ending the execution on the current classical planning problem in $\mathcal{P}$ and enabling the next one:

$$\begin{aligned}
\mathsf{pre}(\mathsf{end}_t) &= G_t \cup \{\mathsf{cs}_{q_n}, \mathsf{test}_t\}, \\
\mathsf{cond}(\mathsf{end}_t) &= \{\emptyset \rhd \{\neg\mathsf{cs}_{q_n}, \mathsf{cs}_{q_0}, \neg\mathsf{test}_t, \mathsf{test}_{t+1}\}\} \\
&\qquad \cup\ \{\{\neg f\} \rhd \{f\} : f \in I_{t+1}\} \cup\ \{\{f\} \rhd \{\neg f\} : \neg f \in I_{t+1}\}\}.
\end{aligned}$$

  The precondition tests that we have reached the goals G_t of the current problem P_t in the terminal controller state q_n, while the effect resets the world state to (q_0, I_{t+1}), i.e. the initial state of the next problem P_{t+1}, which becomes the current problem.

- The initial state is $I'_n = I_1 \cup \{\mathsf{cs}_{q_0}, \mathsf{test}_1\} \cup \{\mathsf{nocond}_q, \mathsf{noact}^b_q, \mathsf{nosucc}^b_q : q \in Q, b \in \{0,1\}\}$, while the goal condition is $G'_n = G_T \cup \{\mathsf{cs}_{q_n}, \mathsf{test}_T\}$.

A plan solving P'_n induces an FSC C and iterates over the classical planning problems $P_t \in \mathcal{P}$, $1 \leq t \leq T$ using actions of type end_t and hence, validating that C solves every $P_1, \ldots, P_T$.

4. Hierarchical Finite State Controllers

This section extends our FSCs formalism to *hierarchical FSCs.* The extension allows a controller to call other controllers, forming hierarchies of FSCs and enabling the reuse of existing controllers. In addition, a *hierarchical FSC* have a list of parameters. This makes it possible to implement recursive solutions by allowing an given controller to call itself with different arguments.

4.1 Parameter Passing

To explain the intuition behind hierarchical FSCs, we borrow several concepts from programming. An FSC is similar to a *procedure* in programming, i.e. an independent program unit with an associated set of program instructions. A procedure can be *called* an arbitrary number of times, which consists in executing the associated program instructions. Procedure calls are organized in a *call stack* that keeps track of where execution should continue once the execution of a given procedure ends.

A procedure may contain *local variables* whose values are different for each call to the procedure. Some of these local variables may be designated as *parameters* of the procedure. When a procedure is called, it is necessary to specify the values of its parameters. Since local variables have different values for different procedure calls, each level of the call stack maintains a *copy* of each local variable, storing its value for the current procedure call.

The first step necessary for defining *hierarchical FSCs* is to introduce the concept of *local variable*. Given a classical planning problem $P = \langle F, A, I, G\rangle$ we assume that the set of fluents F of a planning problem are instantiated from a set of predicates Ψ and a set of objects Ω. Our approach is to designate a subset of objects, $\Omega_v = \{v_1, \ldots, v_q\} \subseteq \Omega$, as *local variables*.

To represent the assignment of values to local variables, we define a predicate $\mathsf{assign} \in \Psi$. Given a local variable object $v \in \Omega_v$ and another object $x \in \Omega \setminus \Omega_v$, the fluent $\mathsf{assign}(v, x)$ denotes that x is the current value assigned to v. Since the local variable v can only have one assigned value at a time, the set of fluents $\{\mathsf{assign}(v, x) : x \in \Omega \setminus \Omega_v\}$ is *exactly-1 invariant*, i.e. precisely one fluent in this set is true at any given moment. All other predicates in Ψ are instantiated exclusively on objects in $\Omega \setminus \Omega_v$.

In programming, a procedure call can either assign a concrete value to a parameter, or pass a variable as argument such that the current value of the variable is assigned to the parameter. In this work we are assuming that *parameter passing* is always of the second type, i.e. the arguments passed to the parameters of a controller are also local variable objects in Ω_v. With this regard, the same local variables can be reused for all the controllers in a *hierarchical FSC*, even if these local variables have different uses in the different controllers. The reason is that each level of the stack maintains a *copy* of each local variable.

4.2 Hierarchical FSCs for Planning

We are now ready to define a hierarchical FSC for a classical planning problem P as a pair $\mathcal{H} = (\mathcal{C}, C_1)$, where $\mathcal{C} = \{C_1, \ldots, C_m\}$ is a set of FSCs for P and $C_1 \in \mathcal{C}$ is the root FSC. Each FSC $C_i = (\langle Q, q_0, q_\perp, \{0, 1\}, \Lambda, T_i, \Gamma_i\rangle, \Phi_i)$, $1 \le i \le m$, shares the same set of controller states Q and output set Λ, and differs only in the three functions Φ_i, T_i and Γ_i that govern the particular transitions of C_i. In addition, each FSC C_i has $k_i \le |\Omega_v|$ associated parameters. Because of symmetry breaking and WLOG, we define the parameter list of C_i as $[v_1, \ldots, v_{k_i}]$, i.e. the k_i first local variable objects in Ω_v.

The shared output set $\Lambda = A \cup \mathcal{Z}$ extends the set of primitive planning actions A to include also the set of possible *calls* $\mathcal{Z} = \{C_i[p] : C_i \in \mathcal{C}, p \in \Omega_v^{k_i}\}$. That is, each transition of a controller C_i either (1), applies an action in A or (2), calls an FSC $C_j[p]$ with the specific arguments $p \in \Omega_v^{k_i}$. We remark that a call action in $\mathcal{Z}$ can be used to implement recursion making a controller to call itself.

4.2.1 The Execution Model of Hierarchical FSCs for Planning

To model the execution of a *hierarchical FSC* $\mathcal{H}$ on a classical planning problem P, we introduce the concept of a *call stack* for FSCs. Because of the local fluents $\mathsf{assign}(v, x)$, a planning state $s = s_l \cup s_g$ can be decomposed into a local state s_l and a global state s_g. Each level of the call stack maintains its own copy of the local state s_l, while the global state s_g is shared among all levels of the call stack. The execution of a hierarchical FSC on P starts at the root controller C_1 in state (q_0, I) and on level 0 of the call stack.

For a given controller C_i and world state (q, s) with $s = s_l \cup s_g$, if $\Gamma_i(q, O(q, s)) = a$ returns an action $a \in A$, the execution semantics is the same as for single FSCs. However, when $\Gamma_i(q, O(q, s)) = C_j[p]$ returns an FSC call in $\mathcal{Z}$, we push call $C_j[p]$ onto the call stack, setting the world state to $(q_0, s'_l \cup s_g)$, i.e. the initial controller state q_0 and a new local state s'_l obtained from s_l by copying the value of each local variable in p to the corresponding parameter in the parameter list $[v_1, \ldots, v_{k_i}]$ of C_j. Execution then proceeds on the next stack level following the definition of C_j.

When we reach a terminal state $(q_\perp, s')$ of C_j with $s' = s'_l \cup s'_g$, control is returned to the parent controller C_i by popping the procedure call $C_j[p]$ from the call stack. Specifically, the state of C_i becomes $(q', s_l \cup s'_g)$ where $q' = T_i(q, O(q, s))$ is the next controller state according to the transition function T_i, and s_l is the assignment to the local variables already stored in the call stack. The execution of a *hierarchical FSC* $\mathcal{H}$ on P terminates when it reaches a terminal state $(q_\perp, s)$ on stack level 0 and, $\mathcal{H}$ solves P iff its execution on that problem terminates and $G \subseteq s$.

To ensure that the execution model remains finite, we define an upper bound ℓ on the size of the call stack. As a consequence, the execution of a *hierarchical FSC* $\mathcal{H}$ on a classical planning problem P has a fourth failure condition:

4. Execution does not terminate because, when executing an FSC call $C_j[p] \in \mathcal{Z}$ the size of the stack equals ℓ. Executing such a call would result in a call stack whose size exceeds the upper bound ℓ, i.e. a *stack overflow*.

4.2.2 An Example of Hierarchical FSC for Planning

To illustrate our definition of *hierarchical FSCs* for planning, we use binary tree traversal as an example (Figure 1). In addition to the tree nodes, we introduce two local variable objects $\Omega_v = \{n, \mathit{child}\}$, and define the actions on that variable objects. We can model this task as a planning problem $P = \langle F, A, I, G \rangle$, where F contains the following fluents:

- For each tree node x, a fluent $\mathsf{visited}(x)$ denoting that x has been visited.
- For each pair of tree nodes x, y, two fluents $\mathsf{left}(x, y)$ and $\mathsf{right}(x, y)$ denoting that y is the left (or right) child of x, respectively.
- For each variable $v \in \{n, \mathit{child}\}$ and tree node x, a fluent $\mathsf{assign}(v, x)$ denoting that x is assigned to the variable v, and a fluent $\mathsf{isNull}(v)$ indicating that v is empty.
- For each $v \in \{n, \mathit{child}\}$, a fluent $\mathsf{isVisited}(v)$ denoting that the node assigned to v has been visited, modelled as a derived predicate $\mathsf{isVisited}(v) \equiv \exists x\ \mathsf{assign}(v, x) \wedge \mathsf{visited}(x)$.

The action set A contains the following actions:

- For each $v \in \{n, child\}$, an action $\mathsf{visit}(v)$ that marks the node assigned to v as visited:

$$\begin{aligned}\mathsf{pre}(\mathsf{visit}(v)) &= \emptyset,\\ \mathsf{cond}(\mathsf{visit}(v)) &= \{\{\mathsf{assign}(v,x)\} \rhd \{\mathsf{visited}(x)\} : \forall x\}.\end{aligned}$$

- For each $u, v \in \{n, child\}$, an action $\mathsf{copyL}(u,v)$ that copies the left child of u onto v:

$$\begin{aligned}\mathsf{pre}(\mathsf{copyL}(u,v)) &= \emptyset,\\ \mathsf{cond}(\mathsf{copyL}(u,v)) &= \{\emptyset \rhd \{\mathsf{isNull}(v), \neg\mathsf{assign}(v,x) : \forall x\}\}\\ &\quad \cup\ \{\{\mathsf{assign}(u,x), \mathsf{left}(x,y)\} \rhd \{\neg\mathsf{isNull}(v), \mathsf{assign}(v,y)\} : \forall x,y\}.\end{aligned}$$

  By default, the effect is $\mathsf{isNull}(v)$, but if u has a left child that node is assigned to v.

- For each $u, v \in \{n, child\}$, an action $\mathsf{copyR}(u,v)$ that copies the right child of u onto v, with a definition analogous to $\mathsf{copyL}(u,v)$.

For the binary tree in Figure 1, the initial state I and goal condition G are defined as

$$\begin{aligned}I &= \{\mathsf{assign}(n,x_0), \mathsf{left}(x_0,x_1), \mathsf{right}(x_0,x_9), \ldots, \mathsf{right}(x_{13},x_{15})\},\\ G &= \{\mathsf{visited}(x_1), \ldots, \mathsf{visited}(x_{15})\}.\end{aligned} \tag{2}$$

Figure 2 shows a hierarchical FSC $\mathcal{H} = (\{C\}, C)$ that solves P. Even though $\mathcal{H}$ contains a single FSC C, it is still hierarchical in the sense that C includes a call to itself, represented by the edge (q_2, q_0) with label $!\mathsf{isVisited}(n)/\mathsf{call}(child)$. Note that C has a single parameter, which we define as the first variable object in Ω_v, namely n.

4.3 Computing Hierarchical Finite State Controllers

We now describe a compilation from a classical planning problem $P = \langle F, A, I, G\rangle$ into another classical planning problem $P^\ell_{n,m} = \langle F^\ell_{n,m}, A^\ell_{n,m}, I^\ell_{n,m}, G^\ell_{n,m}\rangle$, such that solving $P^\ell_{n,m}$ amounts to programming a *hierarchical FSC* $\mathcal{H} = \langle \mathcal{C}, C_1\rangle$ and simulating its execution on P. The parameters of the compilation are n, a bound on the number of controller states, m that is a bound on the number of FSCs and ℓ, a bound on the stack size. For each $C_i \in \mathcal{C}$, the compilation also needs to specify a bound k_i on the number of parameters of the corresponding controller. Note that the actions in A can now be defined on the variable objects to update the value of local variables (e.g. as shown in the previous example with the actions $\mathsf{visit}(v)$, $\mathsf{copyL}(u,v)$ and $\mathsf{copyR}(u,v)$).

The set of fluents is given by $F^\ell_{n,m} = F_r \cup F^\ell_a \cup F^m_{fun} \cup F^\ell_{aux} \cup F_H$ where

- F_r is the set of fluents instantiated from predicates different from assign.
- $F^\ell_a = \{f^l : 0 \leq l \leq \ell, f \in F_a\}$, where $F_a = \{\mathsf{assign}_{v,x} : v \in \Omega_v, x \in \Omega \setminus \Omega_v\}$ is the set of fluents instantiated from assign. By parameterizing on the stack level l, all fluents in F_a are evaluated with respect to the current stack level.
- $F^m_{fun} = \{f^i : f \in F_{fun}, 1 \leq i \leq m\}$ where, as before, F_{fun} is the set of fluents modelling the functions Φ, T and Γ, which we parameterize on the FSC C_i, $1 \leq i \leq m$.

- $F_{aux}^{\ell} = \{f^l : f \in F_{aux}, 0 \leq l \leq \ell\}$ where, as before, F_{aux} is the set of fluents representing the execution model, which we parameterize now on the stack level l.

Moreover, F_H contains the following additional fluents:

- For each l, $0 \leq l \leq \ell$, a fluent lvl^l denoting that l is the current level of the call stack.
- For each $C_i \in \mathcal{C}$ and l, $0 \leq l \leq \ell$, a fluent $\mathsf{fsc}^{i,l}$ denoting that C_i is the FSC being executed on stack level l.
- For each $q \in Q$, $b \in \{0,1\}$, $C_i, C_j \in \mathcal{C}$ and $p \in \Omega_v^{k_j}$, a fluent $\mathsf{call}_{q,j}^{b,i}(p)$ denoting that $\Gamma_i(q,b) = C_j[p]$.

The initial state and goal condition are defined as $I_{n,m}^{\ell} = (I \cap F_r) \cup \{f^0 : f \in I \cap F_a\} \cup \{\mathsf{cs}_{q_0}^0, \mathsf{lvl}^0, \mathsf{fsc}^{1,0}\} \cup \{\mathsf{nocond}_q^i, \mathsf{noact}_q^{b,i}, \mathsf{nosucc}_q^{b,i} : q \in Q, b \in \{0,1\}, C_i \in \mathcal{C}\}$ and $G_{n,m}^{\ell} = G \cup \{\mathsf{cs}_{q_n}^0\}$. In other words, fluents of type $\mathsf{assign}_{v,x} \in F_a$ are initially marked with stack level 0, the controller state on level 0 is q_0, the current stack level is 0, the FSC on level 0 is C_1, and the functions Φ_i, T_i and Γ_i are yet to be programmed for any FSC $C_i \in \mathcal{C}$. To satisfy the goal we have to reach the terminal state q_n on level 0 of the stack.

To establish the actions in the set $A_{n,m}^{\ell}$, we first adapt all actions in A_n by parameterizing on the FSC $C_i \in \mathcal{C}$ and stack level l, $0 \leq l \leq \ell$, adding preconditions lvl^l and $\mathsf{fsc}^{i,l}$, and modifying the remaining preconditions and effects accordingly. As an illustration we provide the definition of the resulting action $\mathsf{pcond}_q^{f,i,l}$:

$$\mathsf{pre}(\mathsf{pcond}_q^{f,i,l}) = \{\mathsf{lvl}^l, \mathsf{fsc}^{i,l}, \mathsf{cs}_q^l, \mathsf{nocond}_q^i\},$$
$$\mathsf{cond}(\mathsf{pcond}_q^{f,i,l}) = \{\emptyset \rhd \{\neg\mathsf{nocond}_q^i, \mathsf{cond}_q^{f,i}\}\}.$$

Compared to the old version of pcond_q^f, the current controller state $\mathsf{cs}_q^l \in F_{aux}^{\ell}$ refers to the stack level l, and fluents nocond_q^i and $\mathsf{cond}_q^{f,i}$ in F_{fun}^m refer to the FSC C_i. The precondition models the fact that we can only program the function Φ_i of C_i in controller state q on stack level l when l is the current stack level, C_i is being executed on level l, the current controller state on level l is q, and Φ_i has not been previously programmed in q.

In addition to the actions adapted from A_n, the set $A_{n,m}^{\ell}$ also contains the following new actions:

- For each $q \in Q$, $b \in \{0,1\}$, $C_i, C_j \in \mathcal{C}$, $p \in \Omega_v^{k_j}$ and l, $0 \leq l < \ell$, an action $\mathsf{pcall}_{q,j}^{b,i,l}(p)$ to program a call from the current FSC C_i to FSC C_j:

$$\mathsf{pre}(\mathsf{pcall}_{q,j}^{b,i,l}(p)) = \{\mathsf{lvl}^l, \mathsf{fsc}^{i,l}, \mathsf{cs}_q^l, \mathsf{evl}^l, \mathsf{o}^{b,l}, \mathsf{noact}_q^{b,i}\},$$
$$\mathsf{cond}(\mathsf{pcall}_{q,j}^{b,i,l}(p)) = \{\emptyset \rhd \{\neg\mathsf{noact}_q^{b,i}, \mathsf{call}_{q,j}^{b,i}(p)\}\}.$$

- For each $q \in Q$, $b \in \{0,1\}$, $C_i, C_j \in \mathcal{C}$, $p \in \Omega_v^{k_j}$ and l, $0 \leq l < \ell$, an action $\mathsf{ecall}_{q,j}^{b,i,l}(p)$ that executes an FSC call:

$$\mathsf{pre}(\mathsf{ecall}_{q,j}^{b,i,l}(p)) = \{\mathsf{lvl}^l, \mathsf{fsc}^{i,l}, \mathsf{cs}_q^l, \mathsf{evl}^l, \mathsf{o}^{b,l}, \mathsf{call}_{q,j}^{b,i}(p), \neg\mathsf{app}^l\},$$
$$\mathsf{cond}(\mathsf{ecall}_{q,j}^{b,i,l}(p)) = \{\emptyset \rhd \{\neg\mathsf{lvl}^l, \mathsf{lvl}^{l+1}, \mathsf{fsc}^{j,l+1}, \mathsf{cs}_{q_0}^{l+1}, \mathsf{app}^l\}\}$$
$$\cup \; \{\{\mathsf{assign}_{p_k,x}^l\} \rhd \{\mathsf{assign}_{v_k,x}^{l+1}\} : 1 \leq k \leq k_j, x \in \Omega_x\}.$$

- For each $C_i \in \mathcal{C}$ and l, $0 < l \leq \ell$, an action $\mathsf{term}^{i,l}$:

$$\mathsf{pre}(\mathsf{term}^{i,l}) = \{\mathsf{lvl}^l, \mathsf{fsc}^{i,l}, \mathsf{cs}^l_{q_n}\},$$
$$\mathsf{cond}(\mathsf{term}^{i,l}) = \{\emptyset \rhd \{\neg\mathsf{lvl}^l, \neg\mathsf{fsc}^{i,l}, \neg\mathsf{cs}^l_{q_n}, \mathsf{lvl}^{l-1}\}\} \cup \{\emptyset \rhd \{\neg\mathsf{assign}^l_{v,x} : v \in \Omega_v, x \in \Omega_x\}\}.$$

As an alternative to $\mathsf{pact}^{b,i,l}_{q,a}$, the action $\mathsf{pcall}^{b,i,l}_{q,j}(p)$ programs an FSC call $C_j[p]$, i.e. defines the output function as $\Gamma_i(q,b) = C_j[p]$. Action $\mathsf{ecall}^{b,i,l}_{q,j}(p)$ executes this FSC call by incrementing the current stack level to $l+1$ and setting the controller state on level $l+1$ to q_0. The conditional effect $\{\mathsf{assign}^l_{p^k,x}\} \rhd \{\mathsf{assign}^{l+1}_{L^k_j,x}\}$ effectively copies the value of the argument p^k on level l to the corresponding parameter L^k_j of C_j on level $l+1$. When in the terminal state q_n, the termination action $\mathsf{term}^{i,l}$ decrements the stack level to $l-1$ and deletes all temporary information about stack level l.

Besides computing a *hierarchical FSC* starting from scratch, the $P^\ell_{n,m}$ compilation is flexible to reuse existing FSCs. In this regard, our compilation can also partially specify the functions Γ_i, Λ_i and Φ_i of an existing FSC C_i by setting to `True` the corresponding fluents of type $\mathsf{cond}^{f,i}_q$, $\mathsf{act}^{b,i}_{q,a}$, $\mathsf{succ}^{b,i}_{q,q'}$ and $\mathsf{call}^{b,i}_{q,j}(p)$ in the initial state $I^\ell_{n,m}$. This way, we can incorporate prior knowledge regarding the configuration of some previously existing FSCs in $\mathcal{C}$. Interestingly, this idea can be exploited to determine whether a given string e belongs to the regular language defined by a given FSC by ignoring the actions for programming the transition function (Segovia-Aguas et al., 2017a). In this case, a solution plan π represents the transitions in the given FSC proving that the string e is accepted.

The $P^\ell_{n,m}$ compilation can also be extended to address a generalized planning problem $\mathcal{P} = \{P_1, \ldots, P_T\}$ in a way analogous to P_n. Specifically, each action end_t, $1 \leq t < T$, should have precondition $G_t \cup \{\mathsf{cs}^0_{q_n}\}$ and reset the state to $I_{t+1} \cup \{\mathsf{cs}^0_{q_0}\}$, i.e. the system should reach the terminal state q_n on stack level 0 and satisfy the goal condition G_t of P_t before execution proceeds on the next problem $P_{t+1} \in \mathcal{P}$. To solve $P^\ell_{n,m}$, a plan hence has to simulate the execution of $\mathcal{H}$ on all planning problems in $\mathcal{P}$.

4.4 Example

We show how the compilation $P^\ell_{n,m}$ computes the hierarchical FSC in Figure 2 for binary tree traversal (with $m=1$). Recall that the initial state is given by $I^\ell_{n,m} = (I \cap F_r) \cup \{f^0 : f \in I \cap F_a\} \cup \{\mathsf{cs}^0_{q_0}, \mathsf{lvl}^0, \mathsf{fsc}^{1,0}\} \cup \{\mathsf{nocond}^i_q, \mathsf{noact}^{b,i}_q, \mathsf{nosucc}^{b,i}_q : q \in Q, b \in \{0,1\}, C_i \in \mathcal{C}\}$, with I given by the expression (2).

At $I^\ell_{n,m}$ the only applicable actions are $\mathsf{pcond}^{f,1,0}_{q_0}$, $f \in F$, for programming the value of $\Phi_1(q_0)$ on stack level 0. In Figure 2 this transition fires no matter what so any static fluent can be programmed here, e.g. $f = \mathsf{left}(x_1, x_3)$. Similar to the example in Section 3.3, the following action sequence programs and simulates the complete transition from q_0 to q_1 on level 0 of the stack:

$$\langle \mathsf{pcond}^{\mathsf{left}(x_1,x_3),1,0}_{q_0}, \mathsf{econd}^{\mathsf{left}(x_1,x_3),1,0}_{q_0}, \mathsf{pact}^{0,1,0}_{q_0,\mathsf{copyL}(n,child)}, \mathsf{eact}^{0,1,0}_{q_0,\mathsf{copyL}(n,child)}, \mathsf{psucc}^{0,1,0}_{q_0,q_1}, \mathsf{esucc}^{0,1,0}_{q_0,q_1} \rangle.$$

The action sequences for programming the transitions from q_1 to q_2 and from q_2 to itself are analogous. The resulting action sequence on A is $\langle \mathsf{copyL}(n, child), \mathsf{visit}(n), \mathsf{copyR}(n,n) \rangle$, and the corresponding effect on F is $\{\mathsf{assign}^0_{child,x_2}, \mathsf{visited}(x_1), \neg\mathsf{assign}^0_{n,x_1}, \mathsf{assign}^0_{n,x_9}\}$.

The programming and simulation of the recursive call of the controller in Figure 2 occurs in the partial state $\{\mathsf{cs}_{q_2}, \mathsf{o}^0\}$, i.e. when $\mathsf{isVisited}(n)$ is false in controller state q_2. To replicate the FSC in Figure 2, the planner should program and simulate the recursive call using actions $\mathsf{pcall}^{0,1,0}_{q_2,1}(child)$ and $\mathsf{ecall}^{0,1,0}_{q_2,1}(child)$. The effect of $\mathsf{ecall}^{0,1,0}_{q_2,1}(child)$ is $\{\neg\mathsf{lvl}^0, \mathsf{lvl}^1, \mathsf{fsc}^{1,1}, \mathsf{cs}^1_{q_0}, \mathsf{app}^0, \mathsf{assign}^1_{n,x_2}\}$, where the assignment $\mathsf{assign}^1_{n,x_2}$ is copied from $\mathsf{assign}^0_{child,x_2}$ due to the argument *child* being passed to the lone parameter n of the FSC C_1 being called. As a result, execution of the controller on P continues on level 1 of the call stack.

On stack level 1, execution is deterministic, resulting in the same transition sequence $q_0 \to q_1 \to q_2 \to q_2 \to q_0$ and causing another recursive call but using now action $\mathsf{ecall}^{0,1,1}_{q_2,1}(child)$, that assigns node x_3 to n on level 2 of the stack. Recursion continues until $\mathsf{isNull}(n)$ becomes true, in which case we can program and simulate the transition from q_1 to the terminal controller state q_3. In turn, this allows us to pop an FSC call from the stack. At this point, since app^l was added by $\mathsf{ecall}^{0,1,l}_{q_2,1}(child)$ at the previous stack level l, we can finally program and simulate the transition from q_2 to q_0 using actions $\mathsf{psucc}^{0,1,l}_{q_2,q_0}$ and $\mathsf{esucc}^{0,1,l}_{q_2,q_0}$. Note that the bound ℓ on the stack size has to be sufficiently large to accommodate all recursive calls otherwise $P^\ell_{n,m}$ will not have a solution. In the particular case of binary tree traversal, ℓ has to be larger than the tree depth.

4.5 Properties

Here we analyze the theoretical properties of our compilation $P^\ell_{n,m}$ for synthesizing hierarchical FSCs.

Theorem 3 (Soundness). *Any plan π that solves $P^\ell_{n,m}$ induces a hierarchical FSC $\mathcal{H} = (\mathcal{C}, C_1)$ that solves P.*

Proof. First we show that at any moment, a single fluent of type lvl^l is true, denoting the top level of the stack, and that all fluents for levels $l+1$ and higher are set to `False`. In the initial state $I^\ell_{n,m}$, the top level is lvl^0 and all fluents for level 1 and higher are false. The only actions for changing the top level are $\mathsf{ecall}^{b,i,l}_{q,j}(p)$ for pushing an FSC call onto level $l+1$ of the stack, and $\mathsf{term}^{i,l}$ for popping an FSC call from level l of the stack. Note that $\mathsf{ecall}^{b,i,l}_{q,j}(p)$ deletes lvl^l and adds lvl^{l+1}, and only adds fluents for level $l+1$. Likewise, $\mathsf{term}^{i,l}$ deletes lvl^l and adds lvl^{l-1}, and deletes all fluents for level l proving that the claim holds.

Next we show that for each stack level l, at or below the top, fluents $\mathsf{fsc}^{i,l}$ and cs^l_q uniquely determine the current FSC $C_i \in \mathcal{C}$ and the current controller state $q \in Q$. In the initial state, only $\mathsf{fsc}^{1,0}$ and $\mathsf{cs}^0_{q_0}$ are true, i.e. the current FSC is C_1 in controller state q_0 on stack level 0. Action $\mathsf{ecall}^{b,i,l}_{q,j}(p)$ adds $\{\mathsf{fsc}^{j,l+1}, \mathsf{cs}^{l+1}_{q_0}\}$, indicating that C_j is the FSC pushed onto level $l+1$ of the stack in controller state q_0. Action $\mathsf{term}^{i,l}$ deletes $\{\mathsf{fsc}^{i,l}, \mathsf{cs}^l_{q_n}\}$, indicating that the FSC call to C_i is popped from level l of the stack. As happens with simple FSCs, the only action for changing the controller state is $\mathsf{esucc}^{b,i,l}_{q,q'}$, which transitions from q to q' when b is the outcome of the observation function in q. The only difference here is that $\mathsf{esucc}^{b,i,l}_{q,q'}$ is parameterized by i and l. Because of precondition $\{\mathsf{lvl}^l, \mathsf{fsc}^{i,l}\}$ of $\mathsf{esucc}^{b,i,l}_{q,q'}$, l has to be the top level of the stack and C_i has to be the FSC that is executing on level l of the stack.

Now we show that actions $\mathsf{pcall}^{b,i,l}_{q,j}(p)$ and $\mathsf{ecall}^{b,i,l}_{q,j}(p)$ for programming and executing an FSC call $C_j[p]$ and action $\mathsf{term}^{i,l}$ for terminating an FSC call correctly simulate the execution model for *hierarhical FSCs*. Since $\mathsf{pcall}^{b,i,l}_{q,j}(p)$ has the same precondition $\{\mathsf{cs}^l_q, \mathsf{evl}^l, \mathsf{o}^{b,l}, \mathsf{noact}^{b,i}_q\}$ as $\mathsf{pact}^{b,i,l}_{q,a}$, and since both delete $\mathsf{noact}^{b,i}_q$, programming an FSC call for q and b in FSC C_i is an alternative to programming an action for q and b, effectively establishing the value of the function $\Gamma_i(q, b) \in A \cup \mathcal{Z}$. The effect of $\mathsf{ecall}^{b,i,l}_{q,j}(p)$ is $\{\mathsf{lvl}^{l+1}, \mathsf{fsc}^{j,l+1}, \mathsf{cs}^{l+1}_{q_0}\}$, pushing the FSC call $C_j[p]$ onto the call stack and making C_j the active FSC on the top level $l+1$. Moreover, the conditional effect $\{\{\mathsf{assign}^l_{p_k,x}\} \rhd \{\mathsf{assign}^{l+1}_{v_k,x}\} : 1 \leq k \leq k_j, x \in \Omega_x\}$ copies the values of the variable objects in p onto the variable objects $v_1, \ldots, v_{k_j}$ that constitute the parameters of C_j. Finally, action $\mathsf{term}^{i,l}$ pops the FSC call involving C_i from the stack when the terminal controller state q_n has been reached.

It only remains to reuse the argument from the proof of Theorem 1 to show that the actions adapted from A_n, program and simulate the execution of an FSC C_i on a single stack level. The actions adapted from A_n, i.e. those for programming or executing a function among Φ_i, T_i and Γ_i, $1 \leq i \leq m$, include the extra precondition $\{\mathsf{lvl}_l, \mathsf{fsc}^{i,l}\}$. In other words, the corresponding FSC C_i has to be active on the top of the stack. Other than that, these actions behave just as for single FSCs.

□

Theorem 4. *If there exists a hierarchical FSC $\mathcal{H} = (\mathcal{C}, C_1)$ that solves P then there exists a plan π that solves $P^\ell_{n,m}$ given that the n, m, and ℓ bounds are large enough.*

Proof. The plan π is built as follows. Whenever the execution of $\mathcal{H}$ on P (starting with the initial planning state I, the first state of the root controller C_1, and an empty call stack) reaches a controller state $q \in Q$ of an FSC C_i for the first time, then π programs the three functions Φ_i, T_i and Γ_i of the FSC C_i as specified by $\mathcal{H}$, and using the corresponding programming actions $\mathsf{pcond}^{f,i,l}_q, \mathsf{pact}^{b,i,l}_{q,a}$ or $\mathsf{psucc}^{b,i,l}_{q,q'}$. As an alternative to $\mathsf{pact}^{b,i,l}_{q,a}$, the FSC calls of $\mathcal{H}$ are programmed with $\mathsf{pcall}^{b,i,l}_{q,j}(p)$.

Once this execution reaches a controller state whose transition functions encoded by Φ_i, T_i and Γ_i are already programmed, an action sequence of type $\langle \mathsf{econd}^{f,i,l}_q, \mathsf{eact}^{b,i,l}_{q,a}, \mathsf{esucc}^{b,i,l}_{q,q'} \rangle$ that simulates the execution of the corresponding transition is added to the plan π. Note that in this case an action $\mathsf{ecall}^{b,i,l}_{q,j}(p)$ is executed as an alternative to $\mathsf{eact}^{b,i,l}_{q,a}$, when the transition to simulate represents a controller call, and a $\mathsf{term}^{i,l}$ action is used to simulate the termination of the execution of a controller $C_i \in \mathcal{C}$.

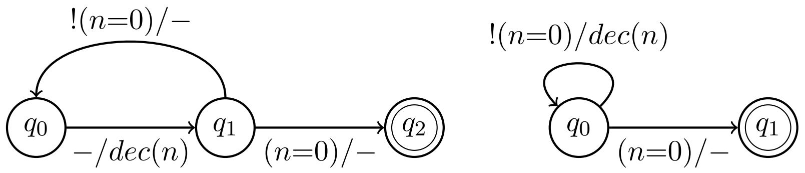
A plan π built this way has the effect of programming $\mathcal{H}$ and simulating the execution of $\mathcal{H}$ on P. The fact that $\mathcal{H} = (\mathcal{C}, C_1)$ solves P implies that the simulation of the execution of $\mathcal{H}$ on P ends achieving G while leaving the root controller C_1 at its terminal state and with the call stack empty, which is also the definition of a plan π solving $P^\ell_{n,m}$. □

5. Comparing FSCs and Planning Programs

Planning programs represent compact and generalized plans assigning planning actions to an enumeration of program lines (Jiménez & Jonsson, 2015; Segovia-Aguas et al., 2016a). Apart from planning actions, the lines of a planning program can also contain *goto instructions* to implement control flow. Figure 4(a) shows a three-line planning program for decreasing the value of variable n until achieving the goal condition $n = 0$. This program assumes that n initially has a positive value, and that action `dec(n)` decrements the current value of n. Instruction `goto(0,!(n=0))`, in line one, indicates a conditional jump to line zero if the value of variable n is not 0.

```
0. dec(n)
1. goto(0,!(n=0))
2. end
```

(a)



(b) (c)

Figure 4: a) Three-line *planning program* for decreasing variable n until achieving $n = 0$; b) an equivalent three-state FSC; c) a more compact FSC representing the same generalized plan.

Planning programs can be understood as syntactic sugar to improve the readability of FSCs that, separates control flow from the primitive planning actions. On the other hand, FSCs can be more compact. Both programs and FSCs can represent hierarchical and recursive solutions as well as reuse existing solutions. Moreover, both programs and FSCs can be computed following a *top-down* approach that searches in a bounded space of solutions.

Theorem 5. *For any planning program Π with n program lines, there exists an equivalent FSC C with n controller states.*

Proof. Given a planning problem $P = \langle F, A, I, G\rangle$ and a planning program $\Pi = \langle w_0, \dots, w_n\rangle$, we prove the theorem by constructing an equivalent FSC $C = (\langle Q, q_0, q_n, \{0,1\}, A, T, \Gamma\rangle, \Phi)$, where $Q = \{q_0, \dots, q_n\}$ has as many controller states as Π has program lines. For each controller state $q_i \in Q$, the functions Φ, T and Γ are defined as follows:

- If the instruction w_i on line i of program Π is a primitive planning action, i.e. $w_i \in A$, then $T(q_i, 0) = T(q_i, 1) = q_{i+1}$ always transitions to the next controller state q_{i+1}, and $\Gamma(q_i, 0) = \Gamma(q_i, 1) = w_i$ always returns action w_i ($\Phi(q_i)$ can be arbitrarily defined since the transition is always to q_{i+1} no matter what fluent we associate with q_i).

- If w_i is a $goto(j, !f)$ instruction, then $\Phi(q_i) = f$ associates fluent f with q_i. Further, $T(q_i, 0) = q_j$ and $T(q_i, 1) = q_{i+1}$ causing a transition to q_j if f is false, else to q_{i+1}, and $\Gamma(q_i, 0) = \Gamma(q_i, 1) = a_\perp$ returning the `no-op` action.

The world state (s, i) of a planning program Π comprises a planning state s and a program line i. We prove by induction that executing Π from (s, i) is equivalent to executing C from (q_i, s), where q_i is the controller state that corresponds to line i.

The base case is given by (s, n), in which case the execution of both Π and C terminates in the same state s (just as q_n is always a terminal controller state, the last instruction w_n of Π is always a termination instruction). The inductive case is given by world state (s, i) such that $i < n$:

- If $w_i \in A$, the resulting world state is $(s', i+1)$ for Π, and (q_{i+1}, s') for C, where $s' = \theta(s, w_i)$ is the result of applying the action w_i in the planning state s.
- If w_i is a goto instruction $goto(j, !f)$, the resulting world state is (s, j) for Π and (q_j, s) for C if f is false, and $(s, i+1)$ for Π and (q_{i+1}, s) for C if f is true.

In each case, the resulting pair of world states (s', i') and $(q_{i'}, s')$ are identical, so by hypothesis of induction executing Π from (s', i') is equivalent to executing C from $(q_{i'}, s')$. In particular, this means that executing Π from the initial world state $(I, 0)$ is equivalent to executing C from (q_0, I), proving that Π and C are equivalent generalized plans. □

Figure 4(b) shows the equivalent FSC that we construct following the proof of Theorem 5 for the planning program in Figure 4(a). Now we prove also the other direction of Theorem 5 and prove hence, that planning programs are as expressive as FSCs (but not necessarily as compact):

Theorem 6. *For any FSC C with n controller states, there exists an equivalent planning program Π with $5 \times n$ program lines.*

Proof. Given a planning problem $P = \langle F, A, I, G \rangle$ and an FSC $C = (\Delta, \Phi)$ with $\Delta = \langle Q, q_0, q_n, \{0, 1\}, A, T, \Gamma \rangle$, we prove the theorem by constructing an equivalent planning program $\Pi = \langle w_0, \ldots, w_{5n} \rangle$ with five times as many lines as C has controller states.

For a given controller state $q_i \in Q$, let $\Phi(q_i) = f$, $T(q_i, 0) = q_j$, $T(q_i, 1) = q_k$, $\Gamma(q_i, 0) = a$, $\Gamma(q_i, 1) = b$ be the transitions from q_i. We can exactly represent these same transitions from q_i using the following partial program:

```
  5i: goto(5i+3,!f)
5i+1: b
5i+2: goto(5k,!false)
5i+3: a
5i+4: goto(5j,!false)
```

Here, `false` is a dummy fluent whose value is always false, causing the corresponding goto condition to trigger every time. Executing the above partial program replicates the transition from q_i, so Π and C are equivalent generalized plans. □

FSCs are at least as compact as planning programs, and often strictly more compact (as shown in Figure 4(c)). In practice, this gives FSCs an advantage over planning programs, since a smaller bound on the number of controller states typically results in faster generation of FSCs.

6. Evaluation

This section evaluates the performance of our approach for the computation of FSCs in a selection of generalized planning benchmarks and programming tasks taken from Bonet et al. (2010), Srivastava et al. (2011) and Segovia-Aguas et al. (2016a).

6.1 Experimental Setup and Benchmarks

All experiments are run on a processor *Intel Core* i7 2.60GHz x 4 with a 4GB memory bound and time limit of 3600s. FSCs are computed solving the corresponding classical planning problem that results from our compilation. The classical planning problems output by our compilation are solved running the following two classical planners:

1. Fast Downward (Helmert, 2006) with the Lama-2011 setting (Richter & Westphal, 2010).

2. Best-First Width Search with the Dual-BFWS setting (Lipovetzky & Geffner, 2017).

We briefly describe here each of the evaluation benchmarks. In the A^nB^n domain the goal is to compute a controller to parse any string consisting of n A's followed by n B's. In *Blocks*, the goal is to compute a controller that unstack blocks from a single tower until a green block is found. In *Fibonacci* the FSC must compute the n^{th} Fibonacci number. In *Gripper*, the goal is to transport a set of balls from one room to another using a two-gripper robot. In *List*, the goal is to visit all the nodes of a linked list (as in the example of Section 3) while in *Reverse*, the goal is to reverse a linked list. In the *Serial Binary Adder* (SBA) domain, we compute a controller that implements the algorithm for the addition of two binary numbers of unbounded size. In *Triangular Sum*, the goal is to compute $\sum_{i=1}^{n} i$ for a given n. In *Tree/DFS*, the goal is to visit all nodes of binary trees whose the nodes may have one child, two children or none. Finally in *Visitall*, the goal is to visit all the cells of a rectangular grid.

6.2 Computing FSCs and Hierarchical FSCs with Classical Planning

Table 1 summarizes the obtained results when using our compilation to compute controllers for the introduced benchmarks.

					FD		**BFWS**	
Domain	$\lvert\mathcal{C}\rvert$	**Sol**	$\lvert Q\rvert$	$\lvert\mathcal{P}\rvert$	**Time(s)**	$\lvert\pi\rvert$	**Time(s)**	$\lvert\pi\rvert$
A^nB^n	1	R	2	1	0.52	46	0.58	39
Blocks	1	OC	3	5	2.73	65	1.08	65
Fibonacci	2	HC	3,2	2,4	2.81,8.28	30,173	3.54,4.81	34,183
Gripper	1	OC	3	2	5.34	140	8.65	135
Hall-A	5	HC	2,2,2, 2,2	2,2,2, 2,2	13.27,5.60,19.09, 32.93,203.03	58,46,46, 46,195	29.64,204.88,284.91, 82.06,20.31	46,46,43, 46,189
List	1	OC	2	6	0.14	159	0.14	159
Reverse	1	OC	3	2	43.56	62	13.85	49
SBA	7	HC	1,1,1,1 1,1,2	2,2,2,2 4,4,8	0.20,0.31,0.44,0.55, 0.79,0.91,271.92	14,14,14,14, 38,38,267	0.07,0.13,0.01,0.01, 0.62,1.79,ME	14,14,14,14, 38,38,ME
T. Sum	1	OC	2	4	7.06	61	14.40	66
Tree/DFS	1	RP	3	4	921.08	422	ME	ME
Visitall	2	HC	2, 2	3, 2	0.40,25.23	84,314	7.01,ME	84,ME

Table 1: Number of controllers used, solution kind (OC=One Controller, HC=Hierarchical Controller, R=Recursivity, RP=Recursivity with Parameters), solution size and instances in $\mathcal{P}$. For each controller: planning time and plan length required for computing the controller.

In many domains our compilation computes a single FSC (OC = One Controller) that solves the input planning instances; there are two domains where the computed solution is a recursive controller (R = Recursivity; RP = Recursivity with Parameters) that are single FSCs that call themselves. For the rest of domains the solutions are hierarchical FSCs (HC = Hierarchical Controller) where controllers call to other controllers, all solutions with $|C| > 1$ fall into this last category.

In addition to the *kind* and *size* of the computed FSCs we also report $|\mathcal{P}|$, the number of classical planning instances given as input to the compilation. Last but not least we report, for each domain, the planning *time* and *plan length* required by the classical planners Fast Downward and Best-First Width Search to compute the controllers. We report *Memory Exceeded* (ME) if the planner overflows the memory limit (there are no timebound errors in Table 1 because either a FSC is found or memory overflows before exceeding the given time limit).

Overall Best-First Width Search does not require as much preprocessing time as Fast Downward. However in certain benchmarks, BFWS requires a larger amount of memory than FD. Next we discuss the obtained solutions for each domain.

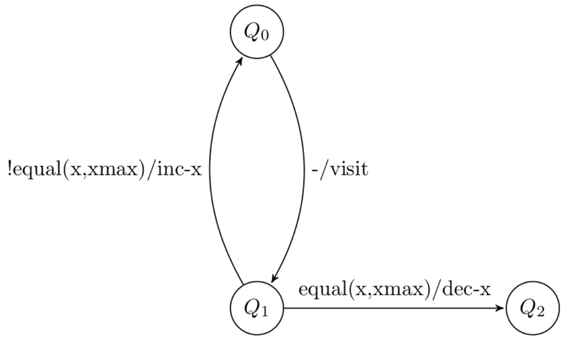
The A^nB^n domain is solved with a recursive FSC without parameters. String parsing problems can be formalized as generalized planning (Segovia-Aguas et al., 2017a). Actions in this domain parse the current letter in the input string and progress the string iterator. If the current letter is still an a, the FSC makes a recursive call to itself else, it parses a b before terminating. Thus, it will process each letter a using n recursive calls, and a letter b before returning from each recursion, processing a total of n b's. The Appendix A includes (1) the PDDL domain and problem definition that is given as input to our compilation (2), the PDDL domain and problem output by our compilation and (3), the solution plan computed by the FD planner to solve the planning instance that is output by our compilation. This solution plan includes the programming of the FSC and its validation on the input instance.

Blocks, *Fibonacci*, *Gripper*, *Hall-A*, *List*, *Reverse*, and *Triangular Sum*, are all domains taken from our previous research work with planning programs (Segovia-Aguas et al., 2016a) to evidence the compactness and effectiveness of *hierarchical FSCs*. Compared to planning programs obtained in that work, the computed FSCs comprise a smaller number of controller states, reducing the time required to compute a generalized plan for the same tasks.

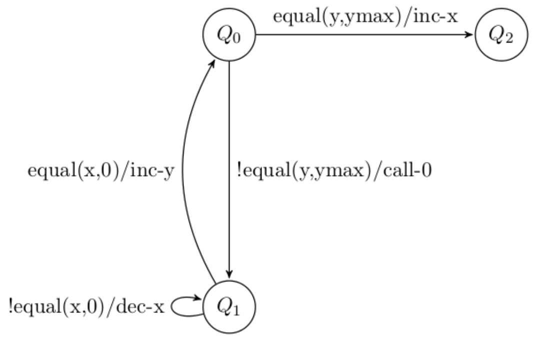
Results on the *Serial-Binary Adder* domain are reported to show how single FSCs, that solve a very specific problem like computing the bit sum and carry from two inputs and a carry, can be combined into a *hierarchical FSC* that iteratively calls previous FSCs to simulate the addition operation of two unbounded binary numbers.

The solution computed for the *Tree/DFS* domain corresponds to the FSC of Figure 2 encoding the condition $\mathsf{isNull}(n)$ as $\mathsf{equals}(n,n)$, where equals is a derived predicate that tests whether two variables have the same value. When applied to a leaf node n, the action $\mathsf{copyR}(n,n)$ deletes the current value of n without adding another value. Hence, evaluating $\mathsf{equals}(n,n)$ returns `false` when n does not have a right child because there is no current value of n to unify with.

In *Visitall* both, Best-First Width Search and Fast Downward, fail to generate a single FSC within the given time and memory bounds. Even if we attempt to generate a hierarchical FSC from scratch, these planners cannot find a solution. Instead, our approach is to generate a hierarchical FSC incrementally. We first generate a single FSC, Figure 5(a), that solves the subproblem of visiting all cells in a single row. We then use our compilation to generate a second FSC, Figure 5(b), that iterates for each row and in every iteration, calls the first controller to visit the current row and then goes back to the first column until there are no more rows to visit.



(a) FSC-0, visits every cell in a row.



(b) FSC-1, for every row, calls FSC-0 and goes back to the first column.

Figure 5: Hierarchical FSC that visits all cells of a grid.

6.3 Assessing the Influence of the Input Instances

Our compilation *programs* a controller and *validates* it on the set $\mathcal{P}$ of classical planning instances that is given as input. The next experiment evidences that the performance of this process is affected by the order in which the planning instances in $\mathcal{P}$ are given as input. This ordering has a positive impact in the performance when the first input instances forces the compilation to program an FSC that generalizes to the remaining instances. On the other hand, the ordering has a negative impact when the programmed FSCs overfits to the first input instances so the planner requires expensive backtracks to validate the remaining input instances.

Table 2 shows the performance of our compilation approach for all the possible orderings of the given input instances. For this analysis we considered only the domains that can be solved with a single FSC. For each domain, the Table 2 reports the number of instances and their possible orderings, factorial in the number of instances. Then for both classical planners, BFWS and FD, the table reports the minimum, maximum and average planning time (in seconds) computed for the given orderings. Last column reports the planning time given the input ordering reported in Table 1 to serve as a reference.

From these results we can conclude that there is a significant difference in the planning times (in some cases by orders of magnitude from *best* to *worst* case, like in the *Blocks* domain) that depends on the particular ordering of the input instances and the used planner. When a human specifies the input order (e.g. the results reported in Table 1) performance usually is below the average running time, but also depends on the used planning system. What can be a good ordering for one planner can result bad for the other planner, like in the *Gripper* domain.

Domain	$\lvert\mathcal{P}\rvert$	**Orderings**	**FD**				**BFWS**			
			Min	**Max**	**Avg**	Ref	**Min**	**Max**	**Avg**	Ref
Gripper	2	2	3.12	4.88	4.00	4.88	8.12	34.86	21.49	8.12
Blocks	5	120	2.18	568.04	87.22	2.36	0.41	520.22	43.25	0.78
List	6	720	0.02	0.04	0.04	0.04	0.02	0.15	0.03	0.02
T. Sum	4	24	2.90	9.78	4.85	7.94	0.26	18.69	6.11	14.46
Reverse	2	2	42.0	51.34	46.67	42.00	16.16	86.59	51.38	16.16

Table 2: For each domain we report the number of instances and possible orderings. For each planner, the minimum, maximum and average times (in secs) for the orderings and the planning time given the ordering from Table 1.

7. Related Work

Previous work on FSCs for planning (Bonet et al., 2010; Hu & De Giacomo, 2013) assumes a partially observable planning model in which the *observation* function is given as input. In our approach the *observation* function is not given as input; our classical planning compilation synthesizes this function (in addition to the *transition* and *output* functions of FSCs). This means that our compilation can generate FSCs that branch on any fluent, since all fluents are considered observable. Further, our approach provides a *call mechanism* that makes it possible to generate recursive and hierarchical solutions as well as to reuse existing FSCs.

Hierarchical FSCs are related to our *planning programs* formalism for the representation and computation of generalized plans (Jiménez & Jonsson, 2015; Segovia-Aguas et al., 2016a; Lotinac, Segovia-Aguas, Jiménez, & Jonsson, 2016). These programs are a special case of FSCs, and in general, FSCs can represent plans more compactly than planning programs. Another related formalism is *automaton plans* (Bäckström et al., 2014), which also store sequential plans compactly using hierarchies of finite state automata. However, automaton plans are *Mealy machines* whose transitions depend on the symbols of an explicit input string. Hence automaton plans are not suitable for representing generalized plans, and their focus is instead on the compression of sequential plans.

Besides solution plans, finite state automata can represent other objects in planning. For instance, the *domain transition graph* is an automaton for representing the possible values of planning state variables (Chen, Huang, & Zhang, 2008). Toropila and Barták (2010) also used finite state machines to represent the domains of the state variables of a given planning instance. Another examples are the LOCM system, that uses finite state machines to represent planning domain models (Cresswell, McCluskey, & West, 2013) or the use of *Petri nets* to represent an entire planning instance (Hickmott, Rintanen, Thiébaux, & White, 2007).

A different application of finite state machines for planning is to compile LTL representations of *temporally extended goals*, i.e. conditions that must hold over the intermediate states of a plan, into a non-deterministic automaton (Baier & McIlraith, 2006). Related to this, the techniques for *bound synthesis* show how to address the computation of finite-state transition systems that satisfy a given LTL formula (Finkbeiner & Schewe, 2013). An interesting research direction is how to adapt our approach for the computation of automata of these kinds.

The aim of *generalized planning* (that is computing solutions that generalize over a set of input instances) is tightly related to *Machine Learning*. In more detail, the *Reinforcement Learning* (RL) literature includes works like Shavlik (1990), Parr and Russell (1998), Dietterich (2000) and Chentanez, Barto, and Singh (2005) that leverages hierarchical decompositions of complex sequential problems and learn when every controller should be applied. In some cases these knowledge can be iteratively included in the bag of controllers, in other cases this set is close but can be reused to solve more complex tasks, options or to learn more complex concepts. Despite, in the best case, learning converges to a solution that minimizes a cost function (or maximizes a reward); in domains with huge state spaces, full observability and a far horizon, it may become unfeasible for RL approaches to reach a goal and even harder to converge, so planning techniques can be applied to explore only promising state space sections.

8. Conclusion

The paper introduced *hierarchical FSCs*, a novel formalism in which controllers can call other controllers (or recursively call themselves) to represent compact generalized plans. The paper also presented a classical planning compilation that makes it possible to use off-the-shelf planners to compute hierarchical FSCs. The compilation is extensible to compute hierarchical FSCs in an incremental fashion to address more challenging generalized planning problems.

Just as in previous work on the automatic generation of FSCs, our compilation takes as input a bound on the number of controller states. Furthermore, for hierarchical FSCs we specify bounds on the number of FSCs and stack levels. An iterative deepening approach could be implemented to automatically derive these bounds. Another issue is the specification of representative subproblems to generate hierarchical FSCs in an incremental fashion. Inspired by *Test Driven Development* (Beck, Beedle, Van Bennekum, Cockburn, Cunningham, Fowler, Grenning, Highsmith, Hunt, Jeffries, et al., 2001), we believe that defining subproblems is a step towards automation.

We follow an inductive approach to generalization, and hence we can only guarantee that the solution generalizes over the instances of the generalized planning problem, much as in previous work on computing FSCs. With this said, all the controllers we report in the paper generalize. In machine learning, the validation of a generalized solution is traditionally done by means of statistics and validation sets. In planning this is an open issue, as well as the generation of relevant examples that lead to solutions that generalize.

Last but not least, our evaluation evidenced the impact of the order of the input tasks in the experimental performance of our compilation. Despite the planner could also be used to determine this order, a better approach is to compute the FSC considering the parallel execution of the controller over all the instances. An interesting research direction is then the use of techniques for progressing belief states, like in *conformant* or *contingent* planning (Palacios & Geffner, 2009; Albore, Palacios, & Geffner, 2009).

Acknowledgments

This work is partially supported by grant TIN2015-67959 and the Maria de Maeztu Units of Excellence Programme MDM-2015-0502, MEC, Spain. Sergio Jiménez is supported by the *RYC15/18009*, program funded by the Spanish government.

Appendix A. Original, Compilation and Generalized Plan of a^nb^n Problem

```
(define (domain AnBn)
 (:requirements :typing)
 (:types index letter)
 (:constants a b empty - letter)
 (:predicates
    (at ?i - index)
    (adjacent ?i1 ?i2 - index)
    (content ?i - index ?l - letter )
    (current-content ?l - letter )
 )

 (:action process-a
  :parameters ()
  :precondition (and (current-content a))
  :effect (and (forall (?i1 ?i2 - index ?l - letter)
               (when  (and (adjacent ?i1 ?i2) (at ?i1) (content ?i2 ?l) )
                      (and (not (at ?i1)) (at ?i2)
                           (not (current-content a)) (current-content ?l)))))
 )

 (:action process-b
  :parameters ()
  :precondition (and (current-content b))
  :effect (and (forall (?i1 ?i2 - index ?l - letter)
                (when  (and (adjacent ?i1 ?i2) (at ?i1) (content ?i2 ?l))
                       (and (not (at ?i1)) (at ?i2)
                            (not (current-content b)) (current-content ?l)))))
  )

)
```

Figure 6: PDDL domain file for parsing strings that belong to the grammar a^nb^n.

```
(define (problem aaaabbbb)
  (:domain AnBn)
  (:objects i0 i1 i2 i3 i4 i5 i6 i7 i8 - index )
  (:init
    (at i0) (current-content a)
    (adjacent i0 i1) (adjacent i1 i2) (adjacent i2 i3) (adjacent i3 i4)
    (adjacent i4 i5) (adjacent i5 i6) (adjacent i6 i7) (adjacent i7 i8)
    (content i0 a) (content i1 a) (content i2 a) (content i3 a)
    (content i4 b) (content i5 b) (content i6 b) (content i7 b)
    (content i8 empty)
  )
  (:goal (and (at i8) ) )
)
```

Figure 7: PDDL problem file for parsing the string *aaaabbbb* that belongs to the grammar a^nb^n.

Next, we show the PDDL domain file that results from compiling the previous classical planning domain for generating an FSC that parses strings that belong to the $a^n b^n$ grammar. The compilation parameters are $Q = 2$ and $l = 5$.

```
( define ( domain compiled-AnBn)
( :requirements :action-costs :conditional-effects :typing )
( :types index letter stackrow stackstate - object )
( :constants
    i0 i1 i2 i3 i4 i5 i6 i7 i8 - index
    a b empty - letter
    row-0 row-1 row-2 row-3 row-4 - stackrow
    state-0 state-1 state-2 - stackstate
)
( :predicates
    ( at ?index0 - index )
    ( adjacent ?index0 ?index1 - index )
    ( content ?index0 - index ?letter1 - letter )
    ( current-content ?letter0 - letter )
    ( empty-cond-stack ?stackstate0 - stackstate )
    ( empty-true-goto-stack ?stackstate0 - stackstate )
    ( empty-false-goto-stack ?stackstate0 - stackstate )
    ( empty-tact-stack ?stackstate0 - stackstate )
    ( empty-fact-stack ?stackstate0 - stackstate )
    ( true-no-act-0 ?stackstate0 - stackstate )
    ( false-no-act-0 ?stackstate0 - stackstate )
    ( state-stack ?stackstate0 - stackstate ?stackrow1 - stackrow )
    ( next-stack-row ?stackrow0 ?stackrow1 - stackrow )
    ( top-stack ?stackrow0 - stackrow )
    ( stack-procedure-0 ?stackrow0 - stackrow )
    ( true-call-0-0 ?stackstate0 - stackstate )
    ( false-call-0-0 ?stackstate0 - stackstate )
    ( accumulator-0 ?stackrow0 - stackrow )
    ( done-evaluating-0 ?stackrow0 - stackrow )
    ( done-applying-0 ?stackrow0 - stackrow )
    ( test-0 ) ( done-programming )
    ( available-state ?stackstate0 - stackstate )
    ( next-state ?stackstate0 ?stackstate1 - stackstate )
    ( ncond-0 ?stackstate0 - stackstate )
    ( true-goto-0 ?stackstate0 ?stackstate1 - stackstate )
    ( false-goto-0 ?stackstate0 ?stackstate1 - stackstate )
    ( cond-at-0 ?index0 - index ?stackstate1 - stackstate )
    ( cond-current-content-0 ?letter0 - letter ?stackstate1 - stackstate )
    ( true-process-a-0 ?stackstate0 - stackstate )
    ( false-process-a-0 ?stackstate0 - stackstate )
    ( true-process-b-0 ?stackstate0 - stackstate )
    ( false-process-b-0 ?stackstate0 - stackstate )
    ( end-cond-0-0 ) ( end-cond-0-1 ) ( end-cond-0-2 )
)
( :functions ( total-cost ) )

( :action program-true-no-act-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( empty-tact-stack ?stackstate0 )
        ( done-evaluating-0 ?stackrow1 )
        ( accumulator-0 ?stackrow1 )
    )
  :effect
    ( and
        ( not ( empty-tact-stack ?stackstate0 ) )
        ( true-no-act-0 ?stackstate0 )
    )
)
( :action program-false-no-act-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( empty-fact-stack ?stackstate0 )
        ( done-evaluating-0 ?stackrow1 )
        ( not ( accumulator-0 ?stackrow1 ) )
    )
  :effect
    ( and
        ( not ( empty-fact-stack ?stackstate0 ) )
        ( false-no-act-0 ?stackstate0 )
    )
)
( :action program-true-process-a-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( current-content a )
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( empty-tact-stack ?stackstate0 )
        ( done-evaluating-0 ?stackrow1 )
        ( accumulator-0 ?stackrow1 )
    )
  :effect
    ( and
        ( not ( empty-tact-stack ?stackstate0 ) )
        ( true-process-a-0 ?stackstate0 )
    )
)
( :action program-false-process-a-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( current-content a )
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( empty-fact-stack ?stackstate0 )
        ( done-evaluating-0 ?stackrow1 )
        ( not ( accumulator-0 ?stackrow1 ) )
    )
  :effect
    ( and
        ( not ( empty-fact-stack ?stackstate0 ) )
        ( false-process-a-0 ?stackstate0 )
    )
)
( :action program-true-process-b-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( current-content b )
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( empty-tact-stack ?stackstate0 )
        ( done-evaluating-0 ?stackrow1 )
        ( accumulator-0 ?stackrow1 )
    )
  :effect
    ( and
        ( not ( empty-tact-stack ?stackstate0 ) )
        ( true-process-b-0 ?stackstate0 )
    )
)
( :action program-false-process-b-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( current-content b )
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( empty-fact-stack ?stackstate0 )
        ( done-evaluating-0 ?stackrow1 )
        ( not ( accumulator-0 ?stackrow1 ) )
    )
  :effect
    ( and
        ( not ( empty-fact-stack ?stackstate0 ) )
        ( false-process-b-0 ?stackstate0 )
    )
)
( :action program-end-2
  :parameters ( ?stackrow0 - stackrow )
  :precondition
    ( and
        ( top-stack ?stackrow0 )
        ( stack-procedure-0 ?stackrow0 )
        ( state-stack state-2 ?stackrow0 )
        ( empty-cond-stack state-2 )
    )
  :effect
    ( and
        ( not ( empty-cond-stack state-2 ) )
        ( end-cond-0-2 )
        ( increase ( total-cost ) 1 )
    )
)
( :action program-nocond-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( empty-cond-stack ?stackstate0 )
    )
  :effect
    ( and
        ( not ( empty-cond-stack ?stackstate0 ) )
        ( ncond-0 ?stackstate0 )
    )
)
( :action program-cond-at-0
  :parameters ( ?index0 - index ?stackstate1 - stackstate ?stackrow2 - stackrow )
  :precondition
    ( and
        ( top-stack ?stackrow2 )
        ( stack-procedure-0 ?stackrow2 )
        ( state-stack ?stackstate1 ?stackrow2 )
        ( empty-cond-stack ?stackstate1 )
    )
  :effect
    ( and
        ( not ( empty-cond-stack ?stackstate1 ) )
        ( cond-at-0 ?index0 ?stackstate1 )
    )
)
( :action program-cond-current-content-0
  :parameters ( ?letter0 - letter ?stackstate1 - stackstate ?stackrow2 - stackrow )
  :precondition
    ( and
        ( top-stack ?stackrow2 )
        ( stack-procedure-0 ?stackrow2 )
        ( state-stack ?stackstate1 ?stackrow2 )
        ( empty-cond-stack ?stackstate1 )
    )
  :effect
    ( and
        ( not ( empty-cond-stack ?stackstate1 ) )
        ( cond-current-content-0 ?letter0 ?stackstate1 )
    )
)
( :action program-true-goto-0
  :parameters ( ?stackstate0 ?stackstate1 - stackstate ?stackrow2 - stackrow )
  :precondition
    ( and
        ( done-evaluating-0 ?stackrow2 )
        ( accumulator-0 ?stackrow2 )
        ( done-applying-0 ?stackrow2 )
        ( top-stack ?stackrow2 )
        ( stack-procedure-0 ?stackrow2 )
        ( state-stack ?stackstate0 ?stackrow2 )
        ( empty-true-goto-stack ?stackstate0 )
        ( available-state ?stackstate1 )
    )
  :effect
    ( and
        ( not ( empty-true-goto-stack ?stackstate0 ) )
        ( true-goto-0 ?stackstate0 ?stackstate1 )
        ( forall
            ( ?stackstate3 - stackstate )
            ( when
                ( next-state ?stackstate1 ?stackstate3 )
                ( available-state ?stackstate3 )
            )
        )
    )
)
( :action program-false-goto-0
  :parameters ( ?stackstate0 ?stackstate1 - stackstate ?stackrow2 - stackrow )
  :precondition
    ( and
        ( done-evaluating-0 ?stackrow2 )
        ( not ( accumulator-0 ?stackrow2 ) )
        ( done-applying-0 ?stackrow2 )
        ( top-stack ?stackrow2 )
        ( stack-procedure-0 ?stackrow2 )
        ( state-stack ?stackstate0 ?stackrow2 )
        ( empty-false-goto-stack ?stackstate0 )
        ( available-state ?stackstate1 )
    )
  :effect
    ( and
        ( not ( empty-false-goto-stack ?stackstate0 ) )
        ( false-goto-0 ?stackstate0 ?stackstate1 )
        ( forall
            ( ?stackstate3 - stackstate )
            ( when
                ( next-state ?stackstate1 ?stackstate3 )
                ( available-state ?stackstate3 )
            )
        )
    )
)
( :action program-true-call-0-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( empty-tact-stack ?stackstate0 )
        ( done-evaluating-0 ?stackrow1 )
        ( accumulator-0 ?stackrow1 )
    )
  :effect
    ( and
        ( not ( empty-tact-stack ?stackstate0 ) )
        ( true-call-0-0 ?stackstate0 )
    )
)
( :action program-false-call-0-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( empty-fact-stack ?stackstate0 )
        ( done-evaluating-0 ?stackrow1 )
        ( not ( accumulator-0 ?stackrow1 ) )
    )
  :effect
    ( and
        ( not ( empty-fact-stack ?stackstate0 ) )
        ( false-call-0-0 ?stackstate0 )
    )
)
( :action repeat-true-no-act-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( done-evaluating-0 ?stackrow1 )
        ( not ( done-applying-0 ?stackrow1 ) )
        ( accumulator-0 ?stackrow1 )
        ( true-no-act-0 ?stackstate0 )
    )
  :effect
    ( and
        ( done-applying-0 ?stackrow1 )
    )
```

```
)
( :action repeat-false-no-act-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( done-evaluating-0 ?stackrow1 )
        ( not ( done-applying-0 ?stackrow1 ) )
        ( not ( accumulator-0 ?stackrow1 ) )
        ( false-no-act-0 ?stackstate0 )
    )
  :effect
    ( and
        ( done-applying-0 ?stackrow1 )
    )
)
( :action repeat-true-process-a-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( current-content a )
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( done-evaluating-0 ?stackrow1 )
        ( not ( done-applying-0 ?stackrow1 ) )
        ( accumulator-0 ?stackrow1 )
        ( true-process-a-0 ?stackstate0 )
    )
  :effect
    ( and
        ( forall
            ( ?index2 ?index3 - index ?letter4 - letter )
            ( when
                ( and
                    ( adjacent ?index2 ?index3 )
                    ( at ?index2 )
                    ( content ?index3 ?letter4 )
                )
                ( and
                    ( not ( at ?index2 ) )
                    ( at ?index3 )
                    ( not ( current-content a ) )
                    ( current-content ?letter4 )
                )
            )
        )
        ( done-applying-0 ?stackrow1 )
    )
)
( :action repeat-false-process-a-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( current-content a )
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( done-evaluating-0 ?stackrow1 )
        ( not ( done-applying-0 ?stackrow1 ) )
        ( not ( accumulator-0 ?stackrow1 ) )
        ( false-process-a-0 ?stackstate0 )
    )
  :effect
    ( and
        ( forall
            ( ?index2 ?index3 - index ?letter4 - letter )
            ( when
                ( and
                    ( adjacent ?index2 ?index3 )
                    ( at ?index2 )
                    ( content ?index3 ?letter4 )
                )
                ( and
                    ( not ( at ?index2 ) )
                    ( at ?index3 )
                    ( not ( current-content a ) )
                    ( current-content ?letter4 )
                )
            )
        )
        ( done-applying-0 ?stackrow1 )
    )
)
( :action repeat-true-process-b-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( current-content b )
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( done-evaluating-0 ?stackrow1 )
        ( not ( done-applying-0 ?stackrow1 ) )
        ( accumulator-0 ?stackrow1 )
        ( true-process-b-0 ?stackstate0 )
    )
  :effect
    ( and
        ( forall
            ( ?index2 ?index3 - index ?letter4 - letter )
            ( when
                ( and
                    ( adjacent ?index2 ?index3 )
                    ( at ?index2 )
                    ( content ?index3 ?letter4 )
                )
                ( and
                    ( not ( at ?index2 ) )
                    ( at ?index3 )
                    ( not ( current-content b ) )
                    ( current-content ?letter4 )
                )
            )
        )
        ( done-applying-0 ?stackrow1 )
    )
)
( :action repeat-false-process-b-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( current-content b )
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( done-evaluating-0 ?stackrow1 )
        ( not ( done-applying-0 ?stackrow1 ) )
        ( not ( accumulator-0 ?stackrow1 ) )
        ( false-process-b-0 ?stackstate0 )
    )
  :effect
    ( and
        ( forall
            ( ?index2 ?index3 - index ?letter4 - letter )
            ( when
                ( and
                    ( adjacent ?index2 ?index3 )
                    ( at ?index2 )
                    ( content ?index3 ?letter4 )
                )
                ( and
                    ( not ( at ?index2 ) )
                    ( at ?index3 )
                    ( not ( current-content b ) )
                    ( current-content ?letter4 )
                )
            )
        )
        ( done-applying-0 ?stackrow1 )
    )
)
( :action repeat-true-goto-0
  :parameters ( ?stackstate0 ?stackstate1 - stackstate ?stackrow2 - stackrow )
  :precondition
    ( and
        ( top-stack ?stackrow2 )
        ( stack-procedure-0 ?stackrow2 )
        ( true-goto-0 ?stackstate0 ?stackstate1 )
        ( state-stack ?stackstate0 ?stackrow2 )
        ( done-evaluating-0 ?stackrow2 )
        ( accumulator-0 ?stackrow2 )
        ( done-applying-0 ?stackrow2 )
    )
  :effect
    ( and
        ( not ( done-evaluating-0 ?stackrow2 ) )
        ( not ( done-applying-0 ?stackrow2 ) )
        ( not ( accumulator-0 ?stackrow2 ) )
        ( not ( state-stack ?stackstate0 ?stackrow2 ) )
        ( state-stack ?stackstate1 ?stackrow2 )
    )
)
( :action repeat-false-goto-0
  :parameters ( ?stackstate0 ?stackstate1 - stackstate ?stackrow2 - stackrow )
  :precondition
    ( and
        ( top-stack ?stackrow2 )
        ( stack-procedure-0 ?stackrow2 )
        ( false-goto-0 ?stackstate0 ?stackstate1 )
        ( state-stack ?stackstate0 ?stackrow2 )
        ( done-evaluating-0 ?stackrow2 )
        ( not ( accumulator-0 ?stackrow2 ) )
        ( done-applying-0 ?stackrow2 )
    )
  :effect
    ( and
        ( not ( done-evaluating-0 ?stackrow2 ) )
        ( not ( done-applying-0 ?stackrow2 ) )
        ( not ( state-stack ?stackstate0 ?stackrow2 ) )
        ( state-stack ?stackstate1 ?stackrow2 )
    )
)
( :action eval-nocond-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 - stackrow )
  :precondition
    ( and
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( ncond-0 ?stackstate0 )
        ( not ( done-evaluating-0 ?stackrow1 ) )
    )
  :effect
    ( and
        ( done-evaluating-0 ?stackrow1 )
        ( accumulator-0 ?stackrow1 )
    )
)
( :action eval-cond-at-0
  :parameters ( ?index0 - index ?stackstate1 - stackstate ?stackrow2 - stackrow )
  :precondition
    ( and
        ( top-stack ?stackrow2 )
        ( stack-procedure-0 ?stackrow2 )
        ( state-stack ?stackstate1 ?stackrow2 )
        ( cond-at-0 ?index0 ?stackstate1 )
        ( not ( done-evaluating-0 ?stackrow2 ) )
    )
  :effect
    ( and
        ( done-evaluating-0 ?stackrow2 )
        ( when
            ( at ?index0 )
            ( and
                ( accumulator-0 ?stackrow2 )
            )
        )
    )
)
( :action eval-cond-current-content-0
  :parameters ( ?letter0 - letter ?stackstate1 - stackstate ?stackrow2 - stackrow )
  :precondition
    ( and
        ( top-stack ?stackrow2 )
        ( stack-procedure-0 ?stackrow2 )
        ( state-stack ?stackstate1 ?stackrow2 )
        ( cond-current-content-0 ?letter0 ?stackstate1 )
        ( not ( done-evaluating-0 ?stackrow2 ) )
    )
  :effect
    ( and
        ( done-evaluating-0 ?stackrow2 )
        ( when
            ( current-content ?letter0 )
            ( and
                ( accumulator-0 ?stackrow2 )
            )
        )
    )
)
( :action repeat-end-main-0-0-1
  :parameters ( )
  :precondition
```

```
    ( and
        ( test-0 )
        ( top-stack row-0 )
        ( state-stack state-1 row-0 )
        ( end-cond-0-1 )
        ( at i8 )
    )
  :effect
    ( and
        ( done-programming )
    )
)
( :action repeat-end-main-0-0-2
  :parameters ( )
  :precondition
    ( and
        ( test-0 )
        ( top-stack row-0 )
        ( state-stack state-2 row-0 )
        ( end-cond-0-2 )
        ( at i8 )
    )
  :effect
    ( and
        ( done-programming )
    )
)
( :action repeat-end-0-0-1
  :parameters ( ?stackrow0 - stackrow ?stackrow1 - stackrow )
  :precondition
    ( and
        ( test-0 )
        ( next-stack-row ?stackrow0 ?stackrow1 )
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack state-1 ?stackrow1 )
        ( end-cond-0-1 )
    )
  :effect
    ( and
        ( not ( top-stack ?stackrow1 ) )
        ( top-stack ?stackrow0 )
        ( not ( state-stack state-1 ?stackrow1 ) )
        ( not ( stack-procedure-0 ?stackrow1 ) )
    )
)
( :action repeat-end-0-0-2
  :parameters ( ?stackrow0 - stackrow ?stackrow1 - stackrow )
  :precondition
    ( and
        ( test-0 )
        ( next-stack-row ?stackrow0 ?stackrow1 )
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack state-2 ?stackrow1 )
        ( end-cond-0-2 )
    )
  :effect
    ( and
        ( not ( top-stack ?stackrow1 ) )
        ( top-stack ?stackrow0 )
        ( not ( state-stack state-2 ?stackrow1 ) )
        ( not ( stack-procedure-0 ?stackrow1 ) )
    )
```

```
)
( :action repeat-true-call-0-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 ?stackrow2 - stackrow )
  :precondition
    ( and
        ( accumulator-0 ?stackrow1 )
        ( true-call-0-0 ?stackstate0 )
        ( next-stack-row ?stackrow1 ?stackrow2 )
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( done-evaluating-0 ?stackrow1 )
        ( not ( done-applying-0 ?stackrow1 ) )
    )
  :effect
    ( and
        ( not ( top-stack ?stackrow1 ) )
        ( top-stack ?stackrow2 )
        ( stack-procedure-0 ?stackrow2 )
        ( state-stack state-0 ?stackrow2 )
        ( done-applying-0 ?stackrow1 )
    )
)
( :action repeat-false-call-0-0
  :parameters ( ?stackstate0 - stackstate ?stackrow1 ?stackrow2 - stackrow )
  :precondition
    ( and
        ( not ( accumulator-0 ?stackrow1 ) )
        ( false-call-0-0 ?stackstate0 )
        ( next-stack-row ?stackrow1 ?stackrow2 )
        ( top-stack ?stackrow1 )
        ( stack-procedure-0 ?stackrow1 )
        ( state-stack ?stackstate0 ?stackrow1 )
        ( done-evaluating-0 ?stackrow1 )
        ( not ( done-applying-0 ?stackrow1 ) )
    )
  :effect
    ( and
        ( not ( top-stack ?stackrow1 ) )
        ( top-stack ?stackrow2 )
        ( stack-procedure-0 ?stackrow2 )
        ( state-stack state-0 ?stackrow2 )
        ( done-applying-0 ?stackrow1 )
    )
)
)
```

```
( define ( problem compiled-aaaabbbb )
( :domain compiled-AnBn )
( :objects )
( :init
    ( at i0 ) ( current-content a )
    ( adjacent i0 i1 ) ( adjacent i1 i2 ) ( adjacent i2 i3 ) ( adjacent i3 i4 )
    ( adjacent i4 i5 ) ( adjacent i5 i6 ) ( adjacent i6 i7 ) ( adjacent i7 i8 )
    ( content i0 a ) ( content i1 a ) ( content i2 a ) ( content i3 a )
    ( content i4 b ) ( content i5 b ) ( content i6 b ) ( content i7 b )
    ( content i8 empty ) ( test-0 )
    ( state-stack state-0 row-0 )
    ( stack-procedure-0 row-0 )
    ( top-stack row-0 )
    ( empty-cond-stack state-0 )
    ( empty-true-goto-stack state-0 ) ( empty-false-goto-stack state-0 )
    ( empty-tact-stack state-0 ) ( empty-fact-stack state-0 )
    ( empty-cond-stack state-1 )
    ( empty-true-goto-stack state-1 ) ( empty-false-goto-stack state-1 )
    ( empty-tact-stack state-1 ) ( empty-fact-stack state-1 )
    ( empty-cond-stack state-2 )
    ( next-stack-row row-0 row-1 ) ( next-stack-row row-1 row-2 )
    ( next-stack-row row-2 row-3 ) ( next-stack-row row-3 row-4 )
    ( available-state state-0 ) ( available-state state-1 )
    ( next-state state-0 state-1 ) ( next-state state-1 state-2 )
)
( :goal ( and ( done-programming ) ) )
( :metric minimize ( total-cost ) )
)
```

Figure 8: PDDL problem file that results from compiling the previous classical planning instance (that encoded the *aaaabbbb* string) for generating an FSC that parses strings that belong to the $a^n b^n$ grammar. The compilation parameters are $Q = 2$ and $l = 5$.

```
(program-nocond-0 state-0 row-0)
(eval-nocond-0 state-0 row-0)
(program-true-process-a-0 state-0 row-0)
(repeat-true-process-a-0 state-0 row-0)
(program-true-goto-0 state-0 state-1 row-0)
(repeat-true-goto-0 state-0 state-1 row-0)
(program-cond-current-content-0 a state-1 row-0)
(eval-cond-current-content-0 a state-1 row-0)
(program-true-call-0-0 state-1 row-0)
(repeat-true-call-0-0 state-1 row-0 row-1)
(eval-nocond-0 state-0 row-1)
(repeat-true-process-a-0 state-0 row-1)
(repeat-true-goto-0 state-0 state-1 row-1)
(eval-cond-current-content-0 a state-1 row-1)
(repeat-true-call-0-0 state-1 row-1 row-2)
(eval-nocond-0 state-0 row-2)
(repeat-true-process-a-0 state-0 row-2)
(repeat-true-goto-0 state-0 state-1 row-2)
(eval-cond-current-content-0 a state-1 row-2)
(repeat-true-call-0-0 state-1 row-2 row-3)
(eval-nocond-0 state-0 row-3)
(repeat-true-process-a-0 state-0 row-3)
(repeat-true-goto-0 state-0 state-1 row-3)
(eval-cond-current-content-0 a state-1 row-3)
(program-false-process-b-0 state-1 row-3)
(repeat-false-process-b-0 state-1 row-3)
(program-false-goto-0 state-1 state-2 row-3)
(repeat-false-goto-0 state-1 state-2 row-3)
(program-end-2 row-3)
(repeat-end-0-0-2 row-2 row-3)
(program-true-goto-0 state-1 state-1 row-2)
(repeat-true-goto-0 state-1 state-1 row-2)
(eval-cond-current-content-0 a state-1 row-2)
(repeat-false-process-b-0 state-1 row-2)
(repeat-false-goto-0 state-1 state-2 row-2)
(repeat-end-0-0-2 row-1 row-2)
(repeat-true-goto-0 state-1 state-1 row-1)
(eval-cond-current-content-0 a state-1 row-1)
(repeat-false-process-b-0 state-1 row-1)
(repeat-false-goto-0 state-1 state-2 row-1)
(repeat-end-0-0-2 row-0 row-1)
(repeat-true-goto-0 state-1 state-1 row-0)
(eval-cond-current-content-0 a state-1 row-0)
(repeat-false-process-b-0 state-1 row-0)
(repeat-false-goto-0 state-1 state-2 row-0)
(repeat-end-main-0-0-2 )
```

Figure 9: Solution plan for the compiled domain and instance. The `program-*` actions define the recursive FSC while `repeat-*` actions execute the programmed FSCs in the given input instance that, in this case, represent the parsing of the *aaaabbbb* string.

References

Albore, A., Palacios, H., & Geffner, H. (2009). A translation-based approach to contingent planning. In *International Joint Conference on Artificial Intelligence*.

Bäckström, C., Jonsson, A., & Jonsson, P. (2014). Automaton plans. *Journal of Artificial Intelligence Research*, *51*, 255–291.

Baier, J., & McIlraith, S. (2006). Planning with Temporally Extended Goals Using Heuristic Search. In *International Conference on Automated Planning and Scheduling*.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al. (2001). The agile manifesto..

Bonet, B., Palacios, H., & Geffner, H. (2010). Automatic derivation of finite-state machines for behavior control. In *AAAI Conference on Artificial Intelligence*.

Brooks, R. (1989). A robot that walks; emergent behaviours from a carefully evolved network. *Neural Computation*, *1*, 253–262.

Buckland, M. (2004). *Programming Game AI by Example.* Wordware Publishing, Inc.

Chen, Y., Huang, R., & Zhang, W. (2008). Fast planning by search in domain transition graph.. In *AAAI Conference on Artificial Intelligence*, pp. 886–891.

Chentanez, N., Barto, A. G., & Singh, S. P. (2005). Intrinsically motivated reinforcement learning. In *Advances in neural information processing systems*, pp. 1281–1288.

Cresswell, S. N., McCluskey, T. L., & West, M. M. (2013). Acquiring planning domain models using locm. *The Knowledge Engineering Review*, *28*(2), 195–213.

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, *13*, 227–303.

Finkbeiner, B., & Schewe, S. (2013). Bounded synthesis. *International Journal on Software Tools for Technology Transfer*, *15*(5-6), 519–539.

Helmert, M. (2006). The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, *26*, 191–246.

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

Hu, Y., & De Giacomo, G. (2013). A generic technique for synthesizing bounded finite-state controllers. In *International Conference on Automated Planning and Scheduling*.

Hu, Y., & Levesque, H. J. (2011). A correctness result for reasoning about one-dimensional planning problems. In *International Joint Conference on Artificial Intelligence*, pp. 2638–2643.

Jiménez, S., & Jonsson, A. (2015). Computing Plans with Control Flow and Procedures Using a Classical Planner. In *International Symposium on Combinatorial Search*, pp. 62–69.

Lipovetzky, N., & Geffner, H. (2017). Best-first width search: Exploration and exploitation in classical planning. In *AAAI Conference on Artificial Intelligence*.

Lotinac, D., Segovia-Aguas, J., Jiménez, S., & Jonsson, A. (2016). Automatic generation of high-level state features for generalized planning. In *International Joint Conference on Artificial Intelligence*.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., & Wilkins, D. (1998). PDDL - The Planning Domain Definition Language. Tech. rep., Yale Center for Computational Vision and Control.

Palacios, H., & Geffner, H. (2009). Compiling uncertainty away in conformant planning problems with bounded width. *Journal of Artificial Intelligence Research*, *35*, 623–675.

Parr, R., & Russell, S. J. (1998). Reinforcement learning with hierarchies of machines. In *Advances in neural information processing systems*, pp. 1043–1049.

Richter, S., & Westphal, M. (2010). The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, *39*, 127–177.

Röger, G., Pommerening, F., & Helmert, M. (2014). Optimal planning in the presence of conditional effects: Extending lm-cut with context-splitting. In *European Conference on Artificial Intelligence*, pp. 765–770.

Segovia-Aguas, J., Jiménez, S., & Jonsson, A. (2016a). Generalized planning with procedural domain control knowledge. In *International Conference on Automated Planning and Scheduling*.

Segovia-Aguas, J., Jiménez, S., & Jonsson, A. (2016b). Hierarchical finite state controllers for generalized planning. In *International Joint Conference on Artificial Intelligence*.

Segovia-Aguas, J., Jiménez, S., & Jonsson, A. (2017a). Generating context-free grammars using classical planning. In *International Joint Conference on Artificial Intelligence*.

Segovia-Aguas, J., Jiménez, S., & Jonsson, A. (2017b). Unsupervised classification of planning instances. In *International Conference on Automated Planning and Scheduling*.

Shavlik, I. W. (1990). Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, *5*(1), 39–70.

Srivastava, S., Immerman, N., Zilberstein, S., & Zhang, T. (2011). Directed search for generalized plans using classical planners. In *International Conference on Automated Planning and Scheduling*.

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

Vallati, M., Chrpa, L., Grzes, M., McCluskey, T. L., Roberts, M., & Sanner, S. (2015). The 2014 international planning competition: Progress and trends. *AI Magazine*, *36*(3), 90–98.