Hierarchical Finite State Controllers for Generalized Planning

Javier Segovia-Aguas and Sergio Jiménez and Anders Jonsson
Dept. Information and Communication Technologies, Universitat Pompeu Fabra
Roc Boronat 138, 08018 Barcelona, Spain
{javier.segovia,sergio.jimenez,anders.jonsson}@upf.edu

Abstract

Finite State Controllers (FSCs) are an effective way to represent sequential plans compactly. By imposing appropriate conditions on transitions, FSCs can also represent generalized 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. We show that hierarchical FSCs can represent generalized plans more compactly than individual FSCs. Moreover, our call mechanism makes it possible to generate hierarchical FSCs in a modular fashion, or even to apply recursion. We also introduce a compilation that enables a classical planner to generate hierarchical FSCs that solve challenging generalized planning problems. The compilation takes as input a set of planning problems from a given domain and outputs a single classical planning problem, whose solution corresponds to a hierarchical FSC.

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: 1) solution compactness [Bäckström et al., 2014]; and 2) the ability to represent generalized plans that solve problems to arbitrarily large problems, as well as problems with partial observability and non-deterministic actions [Bonet et al., 2010; Hu and Levesque, 2011; Srivastava et al., 2011; Hu and 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 consists of 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 Depth-First Search (DFS) only requires a few lines of code. However, a standard FSC cannot implement recursion, and the iterative definition of DFS is considerably more complicated, involving an external data structure.

In this paper we introduce a novel formalism for representing and computing compact and generalized planning solutions that we call hierarchical FSCs. 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 an example hierarchical FSC $C[n]$ that implements DFS traversal of binary trees using recursion. Here, $n$ is the lone parameter of the controller and represents the current node of the binary tree. Condition leaf($n$) tests whether $n$ is a leaf node, while a hyphen ‘-’ indicates that the transition fires no matter what. Action visit($n$) visits node $n$, while copyL($n,m$) and copyR($n,m$) assign the left and right child of node $n$ to $m$, respectively. Action call($m$) is a recursive call to the FSC itself, assigning argument $m$ to the only parameter of the controller and restarting execution from its initial node $Q_0$.

Intuitively, by repeatedly assigning the right child of $n$ to $n$ itself (using the action copyR($n,n$)) and following the cycle of controller states $Q_0, Q_1, Q_2, Q_3, Q_0, \ldots$, the FSC $C[n]$ has the effect of visiting all nodes on the rightmost branch of
the tree until a leaf node is reached. Moreover, by assigning
the left child of $n$ to $\text{child}$ (using the action $\text{copy}_L(n, \text{child})$)
and making the recursive call $\text{call}((\text{child})$, the FSC $C[n]$ is re-
versely executed on all left sub-trees. The controller state
$Q_4$ is a terminal state, and the action $\text{visit}(\text{child})$ on the tran-
sition to $Q_4$ is in fact not needed and could be removed. How-
ever, the FSC is automatically generated by our approach, so
we present conditions and actions exactly as they appear.

Compared to previous work on the automatic generation of
FSCs for planning the contributions of this paper are:
1. A reformulation of the transition function of FSCs that
allows binary branching only in order to reduce the
space of possible controllers.
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 compilation that enables the automatic gener-
ation of hierarchical FSCs for challenging generalized
planning tasks. The compilation takes as input a set of
planning problems from a given domain and outputs a
single classical planning problem whose solution corre-
sponds to a hierarchical FSC. This output is expressed
in PDDL, thus an off-the-shelf classical planner can be
used to generate hierarchical FSCs. The compilation
also makes it possible to incorporate prior knowledge
in the form of existing FSCs to automatically complete
the definition of the remaining FSCs.

2 Background
This section defines our model for classical planning and
presents the formalism we use to define FSCs for planning.

2.1 Classical Planning with Conditional Effects

We describe states and actions in terms of literals. Formally,
given a set of fluents $F$, a literal $l$ is a valuation of a fluent
in $F$, i.e. $l = f$ or $l = \neg f$ for some $f \in F$. A set of lit-
erals $L$ thus represents a partial assignment of values to fluents
(WLOG we assume that $L$ does not assign conflicting values
to any fluent). Given $L$, let $\neg L = \{\neg l : l \in L\}$ be the com-
plement of $L$. A state $s$ is a set of literals such that $|s| = |F|$,
i.e. a total assignment of values to fluents.

A classical planning problem is a tuple $P = (F, A, I, G)$,
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 ac-
ction $a \in A$ has a set of literals $\text{pre}(a)$ called the precondi-
tion and a set of conditional effects $\text{cond}(a)$. Each conditional
effect $C \triangleright E \in \text{cond}(a)$ is composed of sets of literals $C$ (the
condition) and $E$ (the effect). We often describe the initial
state $I \subseteq F$ compactly as the subset of fluents that are true.

Action $a$ is applicable in state $s$ if and only if $\text{pre}(a) \subseteq s$,
and the resulting set of triggered effects is

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

i.e. effects whose conditions hold in $s$. The result of applying
$a$ in $s$ is a new state $\theta(s, a) = (s \setminus \text{eff}(s, a)) \cup \text{eff}(s, a)$.

A plan for $P$ is an action sequence $\pi = (a_1, \ldots, a_n)$ 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
$\pi$ solves $P$ if and only if $G \subseteq s_n$, i.e. if the goal condition is
satisfied following the application of $\pi$ in $I$.

2.2 Finite State Controllers

Given a planning problem $P = (F, A, I, G)$, an FSC is
defined as a tuple $C = (Q, T, q_0, q_f)$, where $Q$ is a set
of controller states, $T : Q \times 2^F \rightarrow Q \times A$ is a (par-
tial) transition function that assumes full observability, and
$q_0 \in Q$ and $q_f \in Q$ are the initial and terminal con-
troller states, respectively. This definition relates to previous
work on FSCs for generalized planning [Bonet et al., 2010;
Hu and De Giacomo, 2013] as follows:

- Just like in previous approaches (and unlike Mealy ma-
achines), transitions do not depend on explicit input se-
quents but on the current planning state.

- Previous approaches assume partial observability of the
current planning state, defining the transition function $T$
on $Q \times O$, where $O$ is the observation set. We instead
define $T$ on $Q \times 2^F$, i.e. on the full set of fluents.

- We define an explicit terminal state $q_f$, while previous
approaches terminate upon reaching the goal condition $G$.
The reason is that we will later extend our definition to
hierarchies of FSCs where goals $G$ are not necessarily
satisfied when the execution of an FSC terminates.

We briefly describe the execution semantics of an FSC $C$
on planning problem $P$. The current world state is a pair
$(q, s) \in Q \times 2^F$ of a controller state and a planning state.
From a pair $(q, s)$, the system transitions to $(q', s')$, where
$(q', s') = T(q, s)$ is the result of applying the transition func-
tion in $(q, s)$ and $s' = \theta(s, a)$ is the result of applying action
$a$ in $s$. Execution starts at $(q_0, I)$ and repeatedly transitions
until reaching a pair $(q_f, s_f)$ that contains the terminal con-
troller state $q_f$. An FSC $C$ solves $P$ iff $G \subseteq s_f$ upon termi-
nation, i.e. if the goal condition holds in $s_f$. The execution of
$C$ fails if it reaches a pair $(q, s)$ that was already visited.

A generalized planning problem $P = \{P_1, \ldots, P_T\}$ is a
set of multiple individual planning problems that share fluents
and actions. Each individual planning problem $P_i \in P$ is thus
defined as $P_i = (F, A, I_i, G_i)$, where only the initial state $I_i$
and goal condition $G_i$ differ from other planning problems in
$P$. An FSC $C$ solves a generalized planning problem $P$ if
and only if it solves every problem $P_i \in P$.

3 Generating Finite State Controllers

This section presents a compilation that takes as input a clas-
sical planning problem $P = (F, A, I, G)$ and a bound $n$ on
the maximum number of controller states, and produces as
output a classical planning problem $P_n$. Actions in $P_n$
are defined such that any plan that solves $P_n$ has to both generate
an FSC $C$ and simulate the execution of $C$ on $P$, thus ver-
ifying that $C$ solves $P$. We later extend this compilation to
generalized planning problems and hierarchies of FSCs.
To generate an FSC \( C = \langle Q, T, q_0, q_f \rangle \) using this compilation we first define \( Q = \{q_0, \ldots, q_n\} \) and set \( q_\bot \equiv q_a \). The only thing that remains is to construct the transition function \( T \). Our approach is to reduce the space of possible controllers by compactly representing \( T : Q \times 2^E \to Q \times A \) using the following three functions \( \Gamma, \Lambda, \Phi \):

- \( \Gamma : Q \to F \) associates a fluent \( f = \Gamma(q) \) to each \( q \in Q \).
- \( \Lambda : Q \times \{0, 1\} \to Q \) returns a successor state in \( Q \).
- \( \Phi : Q \times \{0, 1\} \to A \) returns an action in \( A \).

The transition from a world state \((q, s)\) depends on the truth value of \( \Gamma(q) \) in \( s \), hence allowing binary branching. Let \( \Gamma(q) \in s \) be a test whose outcome is interpreted as a Boolean value in \( \{0, 1\} \). The transition function is then defined as \( T(q, s) = (\Lambda(q, \Gamma(q) \in s), \Phi(q, \Gamma(q) \in s)) \).

We proceed to define \( P_n = \{F_n, A_n, I_n, G_n\} \). The idea behind the compilation is to define two types of actions: 

- *Programming actions* that program the three functions \( \Gamma, \Lambda, \Phi \) for each controller state of \( C \), and
- *execute actions* that simulate the execution of \( C \) on \( P \) by evaluating the actions in the current planning state.

The set of fluents is \( F_n = F \cup F_T \cup F_{aux} \), where \( F_T \) contains the fluents needed to encode the transition function:

- For each \( q \in Q \) and \( f \in F \), a fluent \( \text{cond}_q^f \) that holds iff \( f \) is the condition of \( q \), i.e. if \( \Gamma(q) = f \).
- For each \( q, q' \in Q \) and \( b \in \{0, 1\} \), a fluent \( \text{succ}_{q,q'}^b \) that holds iff \( \Lambda(q, b) = q' \).
- For each \( q \in Q \), \( b \in \{0, 1\} \), and \( a \in A \), a fluent \( \text{act}_{q,a}^b \) that holds iff \( \Phi(q, b) = a \).
- For each \( q \in Q \) and \( b \in \{0, 1\} \), fluent \( \text{nond}_{q,a}^b \) and \( \text{noact}_q^b \) that hold iff we have yet to program the functions \( \Gamma, \Lambda, \Phi \), respectively.

Moreover, \( F_{aux} \) contains the following fluents:

- For each \( q \in Q \), a fluent \( \text{cs}_q \) 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 \( o^b \) and \( o^c \) representing the outcome of the evaluation.

The initial state and goal condition are defined as \( I_n = I \cup \{\text{cs}_{q_0}\} \cup \{\text{nond}_{q_0}, \text{noact}_{q_0}^b, \text{noscucc}_{q_0}^b : q \in \{0, 1\}\} \) and \( G_n = G \cup \{\text{cs}_{q_f}\} \). 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 \( \text{pcond}_q^f \) for programming \( \Gamma(q) = f \):
  \[
  \text{pre}(\text{pcond}_q^f) = \{\text{cs}_q, \text{nond}_q\},
  \text{eff}(\text{pcond}_q^f) = \{\emptyset \triangleright \{\neg\text{nond}_q, \text{cond}_q^f\}\}.
  \]

- For each \( q \in Q \) and \( f \in F \), an action \( \text{cond}_q^f \) that evaluates the condition of the current controller state:
  \[
  \text{pre}(\text{cond}_q^f) = \{\text{cs}_q, \text{cond}_q^f, \neg\text{evl}\},
  \text{eff}(\text{cond}_q^f) = \{\emptyset \triangleright \{\text{evl}\}, \{\neg f\} \triangleright \{o^0\}, \{f\} \triangleright \{o^1\}\}.
  \]

- For each \( q \in Q, b \in \{0, 1\} \) and \( a \in A \), an action \( \text{pact}_{q,a}^b \) for programming \( \Phi(q, b) = a \):
  \[
  \text{pre}(\text{pact}_{q,a}^b) = \text{pre}(a) \cup \{\text{cs}_q, \text{evl}, o^b, \text{noact}_q^b\},
  \text{eff}(\text{pact}_{q,a}^b) = \{\emptyset \triangleright \{\neg\text{noact}_q^b, \text{act}_q^a\}\}.
  \]

- For each \( q \in Q, b \in \{0, 1\} \) and \( a \in A \), an action \( \text{act}_{q,a}^b \) that applies the action of the current controller state:
  \[
  \text{pre}(\text{act}_{q,a}^b) = \text{pre}(a) \cup \{\text{cs}_q, \text{evl}, o^b, \text{act}_{q,a}^b, \neg\text{app}\},
  \text{eff}(\text{act}_{q,a}^b) = \text{eff}(a) \cup \{\emptyset \triangleright \{\text{app}\}\}.
  \]

- For each \( q, q' \in Q \) and \( b \in \{0, 1\} \), an action \( \text{psucc}_{q,q'}^b \) that transitions to the next controller state:
  \[
  \text{pre}(\text{psucc}_{q,q'}^b) = \{\text{cs}_q, \text{evl}, o^b, \text{app}, \text{succ}_{q,q'}^b\},
  \text{eff}(\text{psucc}_{q,q'}^b) = \{\emptyset \triangleright \{\neg\text{succ}_{q,q'}^b, \text{succ}_{q,q'}^b\}\}.
  \]

- For each \( q, q' \in Q \) and \( b \in \{0, 1\} \), an action \( \text{esucc}_{q,q'}^b \) that transitions to the next controller state:
  \[
  \text{pre}(\text{esucc}_{q,q'}^b) = \{\text{cs}_q, \text{evl}, o^b, \text{app}, \text{succ}_{q,q'}^b\},
  \text{eff}(\text{esucc}_{q,q'}^b) = \{\emptyset \triangleright \{\neg\text{cs}_q, \neg\text{evl}, \neg o^b, \neg\text{app}, \text{cs}_{q'}\}\}.
  \]

Actions \( \text{pcond}_q^f, \text{pact}_{q,a}^b, \text{psucc}_{q,q'}^b \) and \( \text{esucc}_{q,q'}^b \) program the three functions \( \Gamma, \Phi, \Lambda \), respectively, while \( \text{econd}_q^f, \text{act}_{q,a}^b \) and \( \text{esucc}_{q,q'}^b \) execute the corresponding function. Fluents evl and app control the order of the execution such that \( \Gamma \) is always executed first, then \( \Phi \), and finally \( \Lambda \).

**Theorem 1.** Any plan \( \pi \) that solves \( P_n \) induces an FSC \( C \) that solves \( P \).

**Proof sketch.** The only way to change the current controller state is to apply an action of type \( \text{esucc}_{q,q'}^b \), which first requires programming and executing the functions \( \Gamma, \Phi, \Lambda \) in that order. Once programmed, the plan \( \pi \) can no longer change these functions since there are no actions that add fluents among \( \text{nond}_q, \text{noact}_q^b \) and \( \text{noscucc}_q^b \). Once programmed for all states and Boolean values \( b \in \{0, 1\} \), the three functions \( \Gamma, \Phi, \Lambda \) together define an FSC \( C \).

We show that \( \pi \) simulates an execution of \( C \) on \( P \). The initial state \( I \cup \{\text{cs}_{q_0}\} \) corresponds to the world state \((q_0, I)\). In any world state \((q, s)\), the plan has to apply the partial action sequence \( \langle \text{econd}_q^f, \text{act}_{q,a}^b, \text{esucc}_{q,q'}^b \rangle \). Action \( \text{econd}_q^f \) adds \( o^b \) where \( b \in \{0, 1\} \) is the truth value of \( f \) in \( s \). Action \( \text{act}_{q,a}^b \) applies the action \( a \) on \( s \) to obtain a new state \( s' = \theta(s, a) \). Finally, action \( \text{esucc}_{q,q'}^b \) transitions to controller state \( q' \). This deterministic execution continues until we reach a terminal state \((q_n, s_n)\) or revisit a world state. If \( \pi \) solves \( P_n \), execution finishes in \((q_n, s_n)\) and the goal condition \( G \) holds in \( s_n \), which is the definition of \( C \) solving \( P \).

We extend the compilation to address generalized planning problems \( \mathcal{P} = \{P_1, \ldots, P_T\} \). In this case a solution to \( P_n \) builds an FSC \( C \) and simulates the execution of \( C \) on all the individual planning problems \( P_t \in \mathcal{P} \). The extension introduces actions \( \text{end}_t \), \( 1 \leq t < T \), with precondition \( G_t \cup \{\text{cs}_{q_n}\} \) and conditional effects that reset the
world state to \((q_0, I_{i+1})\) after solving \(P_t\). In addition, the initial state and goal condition are redefined as \(I_n = I_1 \cup \{cs_{qs}\} \cup \{\text{nocond}_q, \text{noact}_q, \text{nosucc}_q : q \in Q, b \in \{0, 1\}\} \) and \(G_n = G_T \cup \{cs_{qs}\} \).

### 4 Hierarchical Finite State Controllers

This section extends our formalism for FSCs to hierarchical FSCs. We do so by allowing FSCs to call other FSCs. An FSC \(C\) can now have parameters, and calls to \(C\) specify the arguments passed to the parameters of \(C\). Again, we first describe hierarchical FSCs for solving a single planning problem \(P = (F, A, I, G)\), and then extend the idea to generalized planning.

As in PDDL, we assume that fluents in \(F\) are instantiated from predicates. Moreover, we assume that there exist a set of variable objects \(\Omega_v\), a set of value objects \(\Omega_x\), and that for each \(v \in \Omega_v\) and \(x \in \Omega_x\), \(F\) contains a fluent assign \(v, x\) that models an assignment of type \(v = x\). Let \(F_n \subseteq F\) be the set of such assignment fluents and let \(F_r = F \setminus F_n\) be the remaining fluents.

Given a planning problem \(P\) with fluents \(F_n \subseteq F\) induced from sets \(\Omega_v\) and \(\Omega_x\), a hierarchical FSC is a tuple \(\mathcal{H} = (\mathcal{C}, C_1)\), where \(\mathcal{C} = \{C_1, \ldots, C_m\}\) is the set of FSCs in the hierarchy and \(C_1 \in \mathcal{C}\) is the root FSC. We assume that all FSCs in \(\mathcal{C}\) share the same set of controller states \(Q\) and that each \(C_1 \in \mathcal{C}\) has an associated parameter list \(L_i \in \Omega^{v_i}\) consisting of \(k_i\) variable objects in \(\Omega_v\). The set of possible FSC calls is then given by \(Z = \{C_1[p] : C_1 \in \mathcal{C}, p \in \Omega^{v_i}\}\), i.e., all ways to select an FSC \(C_1\) from \(\mathcal{C}\) and assign arguments to its parameters. The transition function \(T_i\) of each FSC \(C_i\) is redefined as \(T_i: Q \times 2^F \rightarrow Q \times (A \cup Z)\) to include possible calls to the FSCs in \(\mathcal{C}\). As before, we represent \(T_i\) compactly using functions \(\Gamma_i\), \(\Lambda_i\) and \(\Phi_i\).

To define the execution semantics of a hierarchical FSC \(\mathcal{H}\) we introduce a call stack. Execution starts in the root FSC, at state \((q_0, I)\) and on level 0 of the stack. In general, for an FSC \(C_i\) and a world state \((q, s)\) and given that \(T_i(q, s) = (q', a)\) returns an action \(a \in A\), the execution semantics is as explained in Section 2 for single FSCs. However, when \(T_i(q, s) = (q', C_1[p])\) returns a call to controller \(C_1[p] \in Z\), we set the state on the next level of the stack to \((q_0, s[p])\), where \(s[p]\) is obtained from \(s\) by copying the value of each variable object in \(p\) to the corresponding parameter of \(C_1\). Execution then proceeds on the next level of the stack following transition function \(T_j\), which can include other FSC calls that invoke higher stack levels. If \(T_j\) reaches a terminal state \((q_{f}, s_{f})\), control is returned to the parent controller \(C_i\). Specifically, the state of \(C_i\) becomes \((q', s')\), where \(s'\) is obtained from \(s_f\) by substituting the original assignments of values to variables on the previous stack level. The execution of a hierarchical FSC \(\mathcal{H}\) terminates when it reaches a terminal state \((q_{f}, s_{f})\) on stack level 0, and \(\mathcal{H}\) solves \(P\) iff \(G \subseteq s_{f}\).

#### 4.1 An Extended Compilation for Hierarchical Finite State Controllers

We now describe a compilation from \(P\) to a classical planning problem \(P'_{n,m} = (F'_{n,m}, A'_{n,m}, F^l_{n,m}, F^T_{n,m}, G'_{n,m})\), such that solving \(P'_{n,m}\) amounts to programming a hierarchical FSC \(\mathcal{H} = (\mathcal{C}, C_1)\) and simulating its execution on \(P\). As before, \(n\) bounds the number of controller states, while \(m\) is the maximum number of FSCs in \(\mathcal{C}\) and \(\ell\) bounds the size of the call stack. The set of fluents is \(F^l_{n,m} = F_r \cup F_a \cup F^T_{n,m} \cup F^\text{aux}_{n,m} \cup F_H\) where

- \(F_a = \{f : f \in F_r, 0 \leq l \leq \ell\}\), i.e., each fluent of type assign has a copy for each stack level \(l\).
- \(F^T_{n,m} = \{f : f \in F_T, 1 \leq i \leq m\}\), i.e., each fluent in \(F_T\) has a copy for each FSC \(C_i \in \mathcal{C}\) defining its corresponding transition function \(T_i\).
- \(F^\text{aux}_{n,m} = \{f : f \in F_{aux}, 0 \leq l \leq \ell\}\), i.e., each fluent in \(F_{aux}\) has a copy for each stack level \(l\).

Moreover, \(F_H\) contains the following additional fluents:

- For each \(l, 0 \leq l \leq \ell\), a fluent \(\text{lv}^{\ell}\) that holds iff \(l\) is the current stack level.
- For each \(C_i \in \mathcal{C}\) and \(l, 0 \leq l \leq \ell\), a fluent \(\text{fsc}^{l,i}\) that holds iff \(C_i\) is the FSC being executed on stack level \(l\).
- For each \(q \in Q\) and \(b \in \{0, 1\}\), \(C_i, C_j \in \mathcal{C}\) and \(p \in \Omega^{v_i}\), a fluent \(\text{call}^{l,i}(p)\) that holds iff \(\Phi_i(q, b) = C_j[p]\).

The initial state and goal condition are now defined as \(I^l_{n,m} = (I \cap F_r) \cup \{f^0 : f \in I \cap F_T\} \cup \{\text{cs}_{qs}\} \cup \{\text{nocond}_q, \text{noact}_q, \text{nosucc}_q : q \in Q, b \in \{0, 1\}, C_i \in \mathcal{C}\}\) and \(G^l_{n,m} = G \cup \{\text{cs}_{qs}\}\). Other words, fluents of type 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 functions \(\Gamma_i\), \(\Lambda_i\) and \(\Phi_i\) 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 \(A^l_{n,m}\), we first adapt all actions in \(A_n\) by parameterizing on the FSC \(C_i \in \mathcal{C}\) and stack level \(l, 0 \leq l \leq 0\), adding preconditions \(\text{lv}^{l}\) and \(\text{fsc}^{l,i}\), and modifying the remaining preconditions and effects accordingly. As an illustration we provide the definition of the resulting action \(\text{pcond}^{l,i}_q\).

\[
\text{pre}(\text{pcond}^{l,i}_q) = \{l^{v_i}, f^{s_i}, c^{s_i}, \text{nocond}_i,\}
\]

\[
\text{eff}(\text{pcond}^{l,i}_q) = \{\theta \triangleright \{\text{nocond}^{l,i}_q, \text{cond}^{l,i}_o\}\}
\]

Compared to the old version of \(\text{pcond}^i_q\), the current controller state \(c^{s_i}_q \in F^\text{aux}_{n,m}\) refers to the stack level \(l\), and fluents \(\text{nocond}^{l}_i\) and \(\text{cond}^{l,i}_o\) in \(F^T_{n,m}\) refer to the FSC \(C_i\). The precondition models the fact that we can only program the function \(\Gamma_i\) 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 \(\Gamma_i\) has not been previously programmed in \(q\).

In addition to the actions adapted from \(A_{n}\), the set \(A^l_{n,m}\) also contains the following new actions:

- For each \(q \in Q, b \in \{0, 1\}, C_i, C_j \in \mathcal{C}, p \in \Omega^{v_i}\) and \(l, 0 \leq l \leq \ell\), an action \(\text{pcall}^{l,i}_q\) to program a call from the current FSC, \(C_i\), to FSC \(C_j\):

\[
\text{pre}(\text{pcall}^{l,i}_q(p)) = \{l^{v_i}, f^{s_i}, c^{s_i}, \text{ev}^{l}, o^{b_i}, \text{noact}^{l}_i,\}
\]

\[
\text{eff}(\text{pcall}^{l,i}_q(p)) = \{\theta \triangleright \{\text{noact}^{l}_i, \text{call}^{l,i}_q(p)\}\}
\]
For each $q \in Q$, $b \in \{0, 1\}$, $C_i, C_j \in C$, and $p \in \Omega^b_i$, and $l$, $0 \leq l < \ell$, an action $\text{ecall}^{b,i,l}_{q,j}(p)$ that executes a call:

$$\text{pre}(\text{ecall}^{b,i,l}_{q,j}(p)) =$$

$$\{lv^l, \text{fsc}^l, cs^l_{q,j}, \text{evl}^l, o^l, \text{call}^{b,i}_{q,j}(p), -\text{app}'\},$$

$$\text{eff}(\text{ecall}^{b,i,l}_{q,j}(p)) = \{0 \triangleright \{-lv^l, lv^{l+1}, cs_{q,j}^l, \text{app}'\}\} \cup \{\text{assign}^{l+1}_{v,x}, 1 \leq k \leq k_j, x \in \Omega_{x}\}.$$

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

$$\text{pre}(\text{term}^{i,l}) = \{lv^l, \text{fsc}^l, cs_{q,i}^l\},$$

$$\text{eff}(\text{term}^{i,l}) = \{0 \triangleright \{-lv^l, -\text{fsc}^l, -cs_{q,i}^l, lv^{l-1}\}\} \cup \{\text{assign}^{l}_{v,x}, v \in \Omega_{v}, x \in \Omega_{x}\}.$$
adding another value, since \( n \) does not have a right child. Hence evaluating \( \text{equals}(n, n) \) returns false, since there is no current value of \( n \) to unify over.

- As previously mentioned, the transition to the terminal state \( Q_4 \) includes an action \( \text{visit}(\text{child}) \) which is not needed; the reason this action is generated by the planner is that there is no option for leaving the action “blank”. Effectively, when executing the FSC the action in question has no effect.

Finally, in Visitall, attempting to generate a single controller for solving all input instances fails. Moreover, even if we set \( m > 1 \) and attempt to generate a hierarchical controller from scratch, the planner does not find a solution within the given time bound. Instead, our approach is to generate a hierarchical FSC incrementally. We first generate two single FSCs, where the first solves the subproblem of iterating over a single row, visiting all cells along the way, and the second solves the subproblem of returning to the first column. We then use the compilation to generate a planning problem \( P_{n,m}^{\text{H}} \) in which two of the FSCs are already programmed, so the classical plan only has to automatically generate the root controller.

## 6 Related Work

The main difference with previous work on the automatic generation of FSCs [Bonet et al., 2010; Hu and De Giacomo, 2013] is that they generate single FSCs relying on a partially observable planning model. In contrast, our compilation generate hierarchical FSCs that can branch on any fluent since we consider all fluents as observable. Our approach also makes it possible to generate recursive solutions and to incorporate prior knowledge as existing FSCs, and automatically complete the definition of the remaining hierarchical FSC.

Hierarchical FSCs are similar to planning programs [Jiménez and Jonsson, 2015; Segovia-Aguas et al., 2016]. Programs are a special case of FSCs, and in general, FSCs can represent a plan more compactly. Another related formalism is automation plans [Bäckström et al., 2014], which also store sequential plans compactly using hierarchies of finite state automata. However, automation plans are Mealy machines whose transitions depend on the symbols of an explicit input string. Hence automaton plans cannot store generalized plans, and their focus is instead on the compression of sequential plans.

FSCs can also represent other objects in planning. Hickmott et al. [2007] and LaValle [2006] used FSCs to represent the entire planning instance. In contrast, Toropila and Barták [2010] used FSCs to represent the domains of individual variables of the instance. Baier and McIlraith [2006] showed how to convert an LTL representation of temporally extended goals, i.e. conditions that must hold over the intermediate states of a plan, into a non-deterministic FSC.

## 7 Conclusion

In this paper we have presented a novel formalism for hierarchical FSCs in planning in which controllers can recursively call themselves or other controllers to represent generalized plans more compactly. We have also introduced a compilation into classical planning which makes it possible to use an off-the-shelf planner to generate hierarchical FSCs. Finally we have showed that hierarchical FSCs can be generated 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 et al., 2001], we believe that defining subproblems is a step towards automation.

Last but not least, 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.

### Acknowledgment

This work is partially supported by grant TIN2015-67959 and the María de Maeztu Units of Excellence Programme MDM-2015-0502, MEC, Spain. Sergio Jiménez is partially supported by the Juan de la Cierva program funded by the Spanish government.
References


