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 a range of similar planning problems. This generalization capacity allows FSCs to represent solutions 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 cy-
n the set of fluent 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 \textit{child} (using the action copy\(L(n, \text{child})\)) and making the recursive call \( \text{call}(\text{child}) \), the FSC \( C[n] \) recursively executed on all left sub-trees. The controller state \( Q_1 \) is a terminal state, and the action visit\( (\text{child}) \) on the transition to \( Q_1 \) is in fact not needed and could be removed. However, 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 generation 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 corresponds 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 literals \( L \) 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 complement 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 action \( a \in A \) has a set of literals \( \text{pre}(a) \) called the \textit{precondition} and a set of conditional effects \( \text{cond}(a) \). Each conditional effect \( C \rightarrow E \in \text{cond}(a) \) 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 \textit{triggered effects} is

\[
\text{eff}(s, a) = \bigcup_{C \rightarrow E \in \text{cond}(a), C \subseteq s} E,
\]

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 \( (s_0, s_1, \ldots, s_n) \) 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 \( C = (Q, T, q_0, q_\perp) \), where \( Q \) is a set of controller states, \( T : Q \times 2^F \rightarrow Q \times A \) is a (partial) transition function that assumes full observability, and \( q_0 \in Q \) and \( q_\perp \in Q \) are the initial and terminal controller 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 machines), transitions do not depend on explicit input sequences 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 \( q_\perp \), 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 \( (q', a) = T(q, s) \) is the result of applying the transition function 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 \( (q_\perp, s_\perp) \) that contains the terminal controller state \( q_\perp \). An FSC \( C \) solves \( P \) if \( G \subseteq s_\perp \) upon termination, i.e. if the goal condition holds in \( s_\perp \). 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 \( P_t \in P \) is thus defined as \( P_t = (F, A, I_t, G_t) \), where only the initial state \( I_t \) and goal condition \( G_t \) differ from other planning problems in \( P \). An FSC \( C \) solves a generalized planning problem \( P \) if and only if it solves every problem \( P_t \in P \).

3 Generating Finite State Controllers

This section presents a compilation that takes as input a classical 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 verifying 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 $q_0 \equiv q_a$. 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 $T : Q \times 2^Q \rightarrow Q \times A$ using the following three functions, $\Gamma$, $\Lambda$, and $\Phi$:

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

The transition from a world state $(q, s)$ depends on the truth value of $\Gamma(q)$ in $s$, 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 the following functions: $P_n = \{F_n, A_n, I_n, G_n\}$. The idea behind the compilation is to define two types of actions: program actions that program the three functions $\Gamma$, $\Lambda$, and $\Phi$ for each controller state of $C$, and execute actions that simulate the execution of $C$ on $P$ by evaluating the functions in the current plan state.

The set of actions is $F_n = F \cup F_T \cup F_{\text{aux}}$, where $F_T$ contains the fluenets needed to encode the transition function:

- For each $q \in Q$, an action $pcond_{q,a}$ that holds if $q$ is the current controller state.

Moreover, $F_{\text{aux}}$ contains the following actions:

- For each $q \in Q$, a fluent $cs_q$ that holds if $q$ is the current controller state.

- For each $q \in Q$, a fluent $evl$ 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.

The initial state and goal condition are defined as $I_n = I \cup \{cs_q\} \cup \{nocond_q, noact_q\}$, $\Phi(q, \Gamma(q) \in s)$. Finally, the set of actions $A_n$ replaces the actions in $A$ with the following actions:

- For each $q \in Q$, a fluent $pcond_{q,a}$ that programs $pcond_{q,a} : f = f'$.

- For each $q \in Q$, a fluent $econd_{q,a}$ that evaluates the condition of the current controller state:

- For each $q \in Q$, a fluent $econd_{q,a}$ that evaluates the condition of the current controller state:

- For each $q \in Q$, a fluent $esucc_{q,a}$ that program the three functions $\Gamma$, $\Lambda$, and $\Phi$, respectively, while $econd_{q,a}$, $eact_{q,a}$ and $esucc_{q,a}$ execute the corresponding function. Fluenets $evl$ 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 $\text{esucc}_{q,a}$, which first requires programming and executing the functions $\Gamma$, $\Lambda$, and $\Phi$ in that order. Once programmed, the plan $\pi$ can no longer change the current controller state since there are no actions that add fluents $\text{nocond}_q$, $\text{noact}_q$, $\text{evl}$, and $\text{app}$. Once programmed for all states and Boolean values $b \in \{0, 1\}$, the three functions $\Gamma$, $\Lambda$, and $\Phi$ together define an FSC $C$.

We show that $\pi$ simulates an execution of $C$ on $P$. The initial state $I \cup \{cs_q\}$ corresponds to the world state $(q_0, t)$. In any world state $(q, s)$, the plan to apply the partial action sequence $(\text{econd}_{q,a}, \text{eact}_{q,a}, \text{esucc}_{q,a})$. Action $\text{econd}_{q,a}$ adds $b$ where $b \in \{0, 1\}$ is the truth value of $f$. Action $\text{eact}_{q,a}$ applies the action $a$ in $s$ to obtain a new state $s' = (q, t, a)$. Finally, action $\text{esucc}_{q,a}$ transitions to controller state $q''$. The 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 $P' = \{P_1, \ldots, P_T\}$. In this case, the solution to $P_n$ builds an FSC $C$ and simulates the execution of $C$ on all the individual planning problems $P_t \in P$. The extension introduces actions $\text{exec}_{t}$, $1 \leq t < T$, with precondition $G_t \cup \{cs_{q_n}\}$ and conditional effects that reset the
world state to \((q_0, I_{r+1})\) after solving \(P_t\). In addition, the initial state and goal condition are redefined as \(I_n = I_1 \cup \{cs_{q_0}\} \cup \{\text{noact}_{v}, \text{nosoact}_{v}: q \in Q, b \in \{0,1\}\}\) and \(G_n = G_T \cup \{cs_{q_0}\}\).

### 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 variables \(\Omega\) and a set of value objects \(\Omega_x\), and that for each \(v \in \Omega\), and \(x \in \Omega_x\, F\) contains a fluent assign\(_{v,x}\) that models an assignment of type \(v = x\). Let \(F_a \subseteq F\) be the set of such assignment fluents and let \(F_r = F \setminus F_a\) be the remaining fluents.

Given a planning problem \(P\) with fluents \(F_a \subseteq F\) induced from sets \(\Omega\) and \(\Omega_x\), a hierarchical FSC is a tuple \(\mathcal{H} = (\mathcal{C}, C_0)\), 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 \(C_i \in \mathcal{C}\) has an associated parameter list \(\ell_i \in \Omega_i^p\) consisting of \(k_i\) variable objects in \(\Omega_v\). The set of possible FSC calls is then given by \(Z = \{C_i[p]: C_i \in \mathcal{C}, p \in \Omega_i^p\}\), i.e., all ways to select an FSC \(C_i\) 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 \(C_i\). 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 \(T_i(q, s) = (q', C_j[p])\) returns a call to controller \(C_j[p]\) \(\in Z\), 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 \(C_j\). 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 \((q_\perp, s_\perp)\) is returned to the parent controller \(C_i\). Specifically, the state of \(C_i\) becomes \((q', s')\), where \(s'\) is obtained from \(s_\perp\) 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 \((q_\perp, s_\perp)\) on stack level 0, and \(\mathcal{H}\) solves \(P\) iff \(G \subseteq s_\perp\).

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

We now describe a compilation from \(P\) to a classical planning problem \(P_{n,m} = (F_{n,m}, A_{n,m}, I_{n,m}, G_{n,m})\), 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 \(F_{n,m}^\ell = F_r \cup F_a \cup F_{\text{call}} \cup F_{\text{aux}}\), and

\[
F_a = \{f^i : f \in F_a, 0 \leq l \leq \ell\}, \quad \text{i.e. each fluent of type assign}_{v,x} \text{ has a copy for each stack level } l.
\]

\[
F_{\text{call}} = \{f^i : f \in F_{\text{call}}, 1 \leq i \leq m\}, \quad \text{i.e. each fluent in } F_T \text{ has a copy for each FSC } C_i \in \mathcal{C} \text{ defining its corresponding transition function } T_i.
\]

\[
F_{\text{aux}} = \{f^i : f \in F_{\text{aux}}, 0 \leq l \leq \ell\}, \quad \text{i.e. each fluent in } F_{\text{aux}} \text{ has a copy for each stack level } l.
\]

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

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

The initial state and goal condition are now defined as \(I_{n,m}^0 = (I \cap F_{\text{call}}) \cup \{f^0 : f \in I \cap F_{\text{call}}\} \cup \{cs_{q_0}\} \cup \{\text{noact}_{v}, \text{nosoact}_{v}: q \in Q, b \in \{0,1\}\}\) and \(G_{n,m}^\ell = G \cup \{cs_{q_0}\}\). In other words, fluents of type assign\(_{v,x}\) in \(F_{\text{call}}\) are initially marked with stack level 0, the controller state on level 0 is \(q_0\), the current stack level is 0, and 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 \(A_{n,m}^\ell\), 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 \(lv^l\) and \(\text{fsc}_{i,l}\), and modifying the remaining preconditions and effects accordingly. As an illustration we provide the definition of the resulting action \(\text{pcond}_{q,b}^{i,l}\):

\[
\text{pcond}_{q,b}^{i,l} = (lv^l, \text{fsc}_{i,l}, cs_{q_0}, \text{noact}_{q,b}).
\]

\[
\text{eff}(\text{pcond}_{q,b}^{i,l}) = \{0 \triangleright \{\text{noact}_{q,b}, \text{cond}_{q,b}^{i,l}\}\}.
\]

Compared to the old version of \(\text{pcond}_{q,b}^{i}\), the current controller state \(cs_{q_0} \in F_{\text{aux}}^\ell\) refers to the stack level \(l\), and fluents \(\text{noact}_{q,b}\) and \(\text{cond}_{q,b}^{i,l}\) in \(F_{\text{call}}^m\) 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 \(A_{n,m}^\ell\) also contains the following new actions:

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

\[
\text{pre}(\text{pcall}_{q,b}^{i,j}(p)) = \{lv^l, \text{fsc}_{i,l}, cs_{q_0}, cv^l, o^{b,i}, \text{noact}_{q,b}\}.
\]

\[
\text{eff}(\text{pcall}_{q,b}^{i,j}(p)) = \{0 \triangleright \{\text{noact}_{q,b}, \text{call}_{q,b}^{i,j}(p)\}\}.
\]
For each \( q \in Q, b \in \{0, 1\}, C_i, C_j \in \mathcal{C}, p \in \Omega_{p}^k \) and \( l, 0 \leq l < \ell \), an action \( \text{ecall}^{b,i,l}_{q,j}(p) \) that executes a call:
\[
\begin{align*}
\text{pre}(\text{ecall}^{b,i,l}_{q,j}(p)) &= \{l!^l, \text{fsc}^i_l, \text{cs}^i_{q,j}, \text{evl}^l, o^{b,i}_l, \text{call}^{b,i}_{q,j}(p), \neg \text{app}'\}, \\
\text{eff}(\text{ecall}^{b,i,l}_{q,j}(p)) &= \{\emptyset \supset \{l!^{l+1}, \text{cs}^i_{q,j+1}, \text{app}'\}\}
\end{align*}
\]
\[
\cup \{\{\text{assign}_{l,x,v}^i\} : 1 \leq k < j, x \in \Omega_x\}.
\]

For each \( C_i \in \mathcal{C} \) and \( l, 0 < j \leq l \), an action \( \text{term}^{i,l}_j \):
\[
\begin{align*}
\text{pre}(\text{term}^{i,l}_j) &= \{l!^l, \text{fsc}^i_l, \text{cs}^i_{q,j}\}, \\
\text{eff}(\text{term}^{i,l}_j) &= \{\emptyset \supset \{\neg l!^{l-1}, \text{fsc}^i_l, \neg \text{cs}^i_{q,j}, l!^{l-1}\}\}
\end{align*}
\]
\[
\cup \{\emptyset \supset \{\neg \text{assign}_{l,v}^i : v \in \Omega_v, x \in \Omega_x\}\}.
\]

An action \( \text{pcall}^{b,i,l}_{q,j}(p) \) programs an FSC \( C_j[p] \), where \( \Phi_i(q, b) = C_j[p] \). Action \( \text{ecall}^{b,i,l}_{q,j}(p) \) executes this FSC call by incrementing the current stack level to \( l + 1 \) and setting the controller state on stack level \( l + 1 \) to \( q_0 \). The conditional effect \( \{\text{assign}_{l,x,v}^i\} \supset \{\text{assign}_{l+1,x,v}^i\} \) effectively copies the value of the argument \( p^k \) on level \( l \) to the corresponding parameter \( l^k \) of \( C_j \) on level \( l + 1 \). When in the terminal state \( q_n \), the termination action \( \text{term}^{i,l}_j \) decrements the stack level to \( l - 1 \) and deletes all temporary information about stack level \( l \).

**Theorem 2.** Any plan \( \pi \) that solves \( P_{n,m} \) induces a hierarchical FSC \( \mathcal{H} \) that solves \( P \).

**Proof sketch.** Similar to the proof in theorem 1, the plan \( \pi \) has to program the functions \( \Gamma_i, \Lambda_i \) and \( \Phi_i \) of each FSC \( C_i \in \mathcal{C} \). Because of the new actions \( \text{pcall}^{b,i,l}_{q,j}(p) \), this includes the possibility of making FSC calls. Hence \( \pi \) implicitly defines a hierarchical FSC \( \mathcal{H} \).

Moreover, \( \pi \) simulates an execution of \( \mathcal{H} \) on \( P \) starting from \((q_0, I)\) on stack level 0. In all but one state \((q, s)\) on stack level \( l \) while executing the FSC \( C_i \), the plan contains a partial action sequence \( \langle \text{cond}^{i,l}_{q,j}, \text{ecall}^{b,i,l}_{q,j}(p), \text{esucc}^{i}_{q,j} \rangle \) that involves an FSC call, the effect of \( \text{ecall}^{b,i,l}_{q,j}(p) \) is to increment the stack level, causing execution to proceed on stack level \( l + 1 \) for the FSC \( C_j \). The only action that decrements the stack level is \( \text{term}^{j,l+1}_j \), which is only applicable once we reach the terminal state \( q_n \) on stack level \( l + 1 \). Once \( \text{term}^{j,l+1}_j \) has been applied, we can now apply action \( \text{esucc}^{b,i,l}_{q,j} \) to transition to the new controller state \( q' \).

\[
\text{If } \pi \text{ solves } P_{n,m}, \text{ execution terminates in a state } (q_n, s_n) \text{ on level 0 and the goal condition holds in } s_n, \text{ satisfying the condition that } \mathcal{H} \text{ solves } P. \]

We remark that the action \( \text{pcall}^{b,i,l}_{q,j}(p) \) can be used to implement recursion by setting \( l = n \), making the FSC \( C_i \) call itself. We can also partially specify the functions \( \Gamma_i, \Lambda_i \) and \( \Phi_i \) of an FSC \( C_i \) by adding fluents of type \( \text{cond}^{i,l}_{q,j}, \text{act}^{b,i}_{q,j}, \text{esucc}^{b,i}_{q,j} \) and \( \text{call}^{b,i}_{q,j}(p) \) to the initial state \( I_{n,m} \). This way we can incorporate prior knowledge regarding the configuration of some previously existing FSCs in \( \mathcal{C} \).

The compilation can be extended to a generalized planning problem \( P = \{P_1, \ldots, P_R\} \) in a way analogous to \( P_n \). Specifically, each action \( \text{end}_i, 1 \leq i < T \), should have pre-condition \( G_i \cup \{\text{cs}^i_{q_0}\} \) and reset the state to \( I_{k+1} \cup \{\text{cs}^i_{q_0}\} \), i.e. the system should reach the terminal state \( q_n \) on stack level 0 and satisfy the goal condition \( G_i \) before execution proceeds on the next problem \( P_{i+1} \in P \). To solve \( P_{n,m} \), a plan hence has to simulate the execution of \( \mathcal{H} \) on all planning problems in \( P \).

## 5 Evaluation

We evaluate our approach in a set of generalized planning benchmarks and programming tasks taken from Bonet et al. [2010] and Segovia-Aguas et al. [2016]. In all experiments, we run the classical planner Fast Downward [Helmert, 2006] with the LAMA-2011 setting [Richter and Westphal, 2010] on a processor Intel Core i5 3.10GHz x 4 with a 4GB memory bound and time limit of 3600s.

We briefly describe each domain used in experiments. In Blocks, the goal is to unstack blocks from a single tower until a green block is found. In Gripper, the goal is to transport a set of balls from one room to another. In List, the goal is to transport a green block is found. In Gripper, the goal is to transport a green block is found. In Gripper, the goal is to transport a green block is found. In Gripper, the goal is to transport a green block is found.

Table 1 summarizes the obtained experimental results. In all but two domains our compilation makes it possible to find a single FSC (OC=One Controller) that solves all planning instances in the input. Moreover, we manually verified that the resulting FSC solves all other instances from the same domain. These results reflect those of earlier approaches, but in the domains from Segovia-Aguas et al. [2016], the FSC is able to store generalized plans more compactly, and generation of the FSC is faster.

In Tree/DFS, as mentioned in the introduction, generating a single FSC that solves the problem iteratively without recursive calls is difficult. In contrast, since our compilation simulates a call stack, we are able to automatically generate the FSC in Figure 2. There are some discrepancies with respect to the compilation that we address below:

- As described, a solution to the compiled planning problem \( P_{n,m} \) has to program a condition for each controller state, while the FSC in Figure 2 includes deterministic transitions. However, since all fluents in \( f \) are potential conditions, programming a condition on a fluent that is static is effectively equivalent to programming a deterministic transition, since the outcome of the evaluation will always be the same for this fluent.

- In the solution generated by the planner, the condition \( \text{leaf}(n) \) is actually evaluated by a condition \( \text{equals}(n, n) \), where equals is a derived predicate that tests whether two variables have the same value. The reason this works is that when applied to a leaf node \( n \), the action \( \text{copyR}(n, n) \) deletes the current value of \( n \) without
We then use the compilation to generate a planning problem ond solves the subproblem of returning to the first column. A single row, visiting all cells along the way, and the sec-
FSCs, where the first solves the subproblem of iterating over hierarchical FSC incrementally. We first generate two single given time bound. Instead, our approach is to generate a hi-

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-Agus et al., 2016]. Programs are a special case of FSCs, and in general, FSCs can represent a plan more compactly. 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 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 domains of individual variables of the instance. 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.

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.

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.

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.

Acknowledgment

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 partially supported by the Juan de la Cierva program funded by the Spanish government.
References


