A Generic Framework and Solver for Synthesizing Finite-State Controllers

Yuxiao Hu
Department of Computer Science
University of Toronto
Toronto, ON M5S3G4, Canada
yuxiao@cs.toronto.edu

Giuseppe De Giacomo
Dipartimento di Informatica e Sistemistica
SAPIENZA Università di Roma, Italy
Via Ariosto 25, 00185 Roma, Italy
degiacomo@dis.uniroma1.it

Abstract

Finite-state controllers are a compact and effective plan representation for agent behavior control widely used in AI. In this paper, we propose a generic framework and related solver for synthesizing bounded finite-state controllers, and show its instantiations to three different applications, including generalized planning, planning programs and service composition under partial observability and controllability. We show that our generic solver is sound and complete, and is amenable to heuristics that take into account the structure of the specific target instantiation. Experimental results show, quite surprisingly, that instantiations of our solver to the problems above outperform most of the tailored approaches in the literature. This suggests that our proposal is a promising base point for future research on finite-state controller synthesis.

Introduction

Finite-state controllers are a compact and effective plan representation for agent behavior control widely used in AI. Consider, for example, the following recently-proposed domains:

Generalized planning: Bonet *et al* (2009) presented a simple but interesting example of a type of contingent planning: In a 1×5 grid world shown in Figure 1(a), the robot is initially in one of the leftmost two cells. The goal is to visit cell B, and then go to A. The robot can perform left and right moves within the grid, and can observe whether its current location is A, B or neither of them. Bonet *et al.* claim that the finite-state controller in Figure 1(c) represents a notable alternative to traditional "conditional plans," as the controller is not only a correct plan for this particular instance, but also for all $1 \times N$ grids with $N \geq 2$. In a sense, Figure 1(c) is a generalized plan for all planning problems of the form shown in Figure 1(b). A number of similar generalized planning problems with controller-based solutions have recently been proposed, *e.g.*, treechopping (Levesque 2005), delivery (Srivastava, Immerman, and Zilberstein 2008), *etc.*

Service composition: Imagine a service composition task (De Giacomo, De Masellis, and Patrizi 2009), where the goal is to provide a target service (*e.g.*, M_T shown in Figure 2(a)) from a set of available services (*e.g.* M_1 and M_2

Copyright © 2011, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

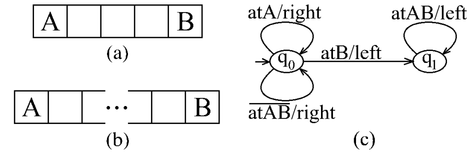

Figure 1: Generalized planning example.

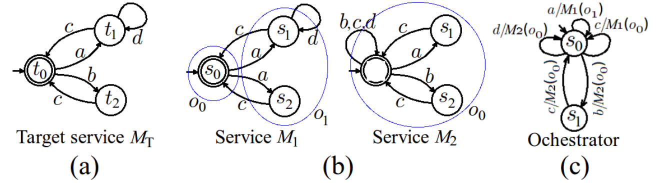

Figure 2: Service composition example.

shown in Figure 2(b)). Initially, both the target and the available services are in their initial states. At any time, the target service may request an action from its current state, and an orchestrator should select one of the available services to perform this requested action. Upon action completion, the target and the chosen services will update their control states according to the transitions, but the orchestrator only has partial observability as which states the services are in (*e.g.* o_0 and o_1). It is the orchestrator's responsibility to guarantee that all legal requests of the target can be satisfied at any time, and when the target is in its final states, so must be all the available services. Figure 2 shows one possible orchestrator for our example problem.

Planning programs: As a middle ground of AI planning (Ghallab, Nau, and Traverso 2004) and high-level action languages (Levesque et al. 1997), De Giacomo *et al.* (2010) proposed a new approach to agent programming via "planning programs." Given a dynamic domain (*e.g.*, the researcher's world involving walking, driving and bus-riding between her home, department, the parking lot and the pub, shown in Figure 3(a)), and a goal network involving maintenance and achievement goals (Figure 3(b)), the planning problem is to find a strategy such that all goal requests can be accommodated for the long-lived agent. For example, if the current goal node is t_1 in Figure 3(b), the next goal may be either a transition to t_0, requesting to "be home

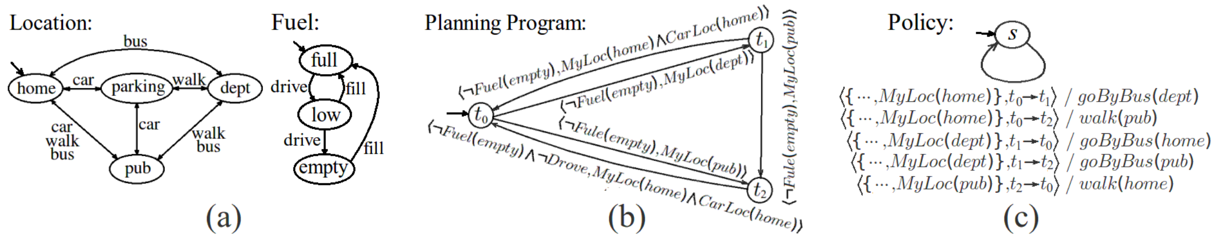

Figure 3: Planning program example.

with the car parked at home while maintaining a non-empty fuel tank," or a transition to t_2, requesting to "be in the pub while maintaining a non-empty tank." In either case, the researcher must behave in a way that not only achieves the current goals, but also ensures that all possible future requests can still be satisfied after her actions. Figure 3(c) shows a policy for this problem.

The three applications are very different in structure, but share one thing in common: the desired plans are all finite-state controllers, *e.g.*, the generalized plan in Figure 1(c), the orchestrator in Figure 2(c), and the policy in figure 3(c).

Such controllers are usually hand-written by domain experts, so the design process may involve tremendous amount of time and expertize, especially when the systems become complex. It is thus desirable to generate the controllers automatically from a declarative specification of the agent's bahavior. This idea is drawing increasing attention in the research community, and leading to nice theoretical and algorithmic results, *e.g.*, in the aforementioned literature (Bonet, Palacios, and Geffner. 2009; De Giacomo, Patrizi, and Sardina 2010; De Giacomo, De Masellis, and Patrizi 2009).

In this paper, we propose a generic framework and related solver for the synthesis of *bounded* finite-state controller. In particular, the solver is based on direct search in AND-OR trees that incrementally capture the possible executions of the (partial) controller in its environment. We show that our generic solver is indeed sound and complete, and is amenable to heuristics that take into account the structure of the specific problem it is applied to. We also show that simple adaptations of our solver to the different problems above, are sound and complete (notice that all such problems allows for bounded controllers), and, quite surprisingly, outperform most of the tailored approaches in the literature. This suggests that our proposal is a promising base point for future research on finite-state controller synthesis.

In the following, we first give a formal definition of the controller synthesis framework, and then elaborate our generic solver with correctness guarantee. After that, we explain the instantiation of our framework to the three types of problems above, and show the very encouraging experimental results. We conclude with a note on immediate future work.

Controller Synthesis Framework

We are interested in control problems of the following form: given a dynamic environment and a behavior specification for an agent acting in this environment, find a strategy in the form of a finite-state controller so that the behavior is realized.

Formally, we define the *dynamic environment* as a tuple $\mathcal{E} = \langle \mathcal{A}, \mathcal{O}, \mathcal{S}, \mathcal{I}, \Delta, \Omega \rangle$, where

- $\mathcal{A}$ is a finite set of actions,
- $\mathcal{O}$ is a finite set of observations,
- $\mathcal{S}$ is a finite set of world states (the state space),
- $\mathcal{I} \subseteq \mathcal{S}$ is a set of possible initial states,
- $\Delta \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the transition relation, and
- $\Omega : \mathcal{S} \to \mathcal{O}$ is the observation function.

We use the notation $s \xrightarrow{a} s'$ to denote $\langle s, a, s' \rangle \in \Delta$.

An *(N-bounded) finite-state controller* for an environment $\mathcal{E} = \langle \mathcal{A}, \mathcal{O}, \mathcal{S}, \mathcal{I}, \Delta, \Omega \rangle$ is a tuple $\mathcal{C} = \langle Q, q_0, T \rangle$, where

- $Q = \{1, \ldots, N\}$ is the finite set of control states,
- $q_0 = 1$ is the initial control state,
- $T : \langle Q \times \mathcal{O} \rangle \to \langle Q \times \mathcal{A} \rangle$ is the transition function.

We use $q \xrightarrow{o/a} q'$ to denote $T(q, o) = \langle q', a \rangle$. It is easy to see that finite-state controllers with this definition are essentially Mealy machines, and have a natural graphical representation.

An *(execution) history* of controller $\mathcal{C}$ in environment $\mathcal{E}$ is a finite sequence $h = \langle q_0, s_0 \rangle, \langle q_1, s_1 \rangle, \cdots, \langle q_n, s_n \rangle$, such that there is a corresponding sequence of actions $r = a_1 a_2 \cdots a_n$, called the *run* of $\mathcal{C}$ in $\mathcal{E}$, satisfying

- $s_0 \in \mathcal{I}$ (recall that $q_0 = 1$ by definition),
- $q_i \xrightarrow{\Omega(s_i)/a_{i+1}} q_{i+1}$ and $s_i \xrightarrow{a_{i+1}} s_{i+1}$.

We use $h \preceq h'$ to mean that the history h is a prefix of the history h'. We call extension of an history h any history h' such that $h \preceq h'$.

For the purpose of this paper, we focus on finitely verifiable controller specifications bounded by state repetition. Intuitively, if an execution history contains repeating states, then there is no need to consider any extension of it.

Formally, a *controller specification* for $\mathcal{E}$ and control states Q is a function $\beta : (Q \times \mathcal{S})^* \to \{\mathsf{true}, \mathsf{false}, \mathsf{unknown}\}$ satisfying the following condition: for all $h' \in (Q \times \mathcal{S})^*$, if $\langle q_i, s_i \rangle, \langle q_j, s_j \rangle \in h'$ for some $i \neq j$, $q_i = q_j$ and $s_i = s_j$, then there exists a prefix $h \preceq h'$ such that $\beta(h) \in \{\mathsf{true}, \mathsf{false}\}$. Intuitively, a true value for $\beta(h)$ means that h is valid and conclusive, *i.e.*, there is no need to further extend it; false means it is invalid, *i.e.*, h should never be generated by the controller; $\mathsf{unknown}$ means validity cannot

be concluded yet, so all one-step extensions of h need to be examined.

We say that a controller $\mathcal{C}$ for an environment $\mathcal{E}$ *satisfies* the controller specification β iff for all its histories h in $\mathcal{E}$ we have that $\beta(h) \neq \mathsf{false}$ and there exists an extension h' such that $\beta(h') = \mathsf{true}$. (In fact after a certain number of steps all its extension become true, due to the required condition above.) Given a controller $\mathcal{C}$ in $\mathcal{E}$ we can check whether it satisfies β. More interestingly, if we do not have the controller yet, we can use β to actually search for it. This is what we study next.

Generic Solver

Associated to the above framework, we propose a (surprisingly simple) generic solver that systematically searches the space of bounded finite-state controllers by traversing the AND-OR execution tree of incremental partial controllers. Here, the OR nodes are the choice points for the controller's actions and transitions, while the AND nodes handle all possible environment feedback. Each node of the search tree keeps a current partial controller, along with its current control state, the world states and the execution history so far.

Given a dynamic environment $\mathcal{E}$, a bound N on the number of control states, and a behavior specification β, the algorithm in Figure 4 generates a controller $\mathcal{C}$ by a call to **synthesize**$_{\mathcal{E},N}(\mathcal{I})$, where $\mathcal{I}$ is the set of initial states in $\mathcal{E}$ (line 1). This creates the root of the search tree, which is an AND node containing an empty controller $\emptyset$,[1] the initial control state 1, initial world states $\mathcal{I}$, and empty history $\emptyset$.

The function AND_step (lines 4–7) represents AND nodes in the search tree that handle all contingencies in the world states. For this, the function OR_step is called for each possible state $s \in S$, with the history h augmented with the current control and world states $\langle q, s\rangle$. Note that C is updated after each call to OR_step (line 6), so that the resulting controller is incrementally implemented to handle all states in S.

The function OR_step (lines 9–20) simulates a one-step execution of the current partial controller C for a given control state q and world state s with execution history h. Four different cases may arise during this simulation:

1. If the behavior specification function $\beta(h)$ returns true, it means the current controller C has generated a valid and conclusive history, so no further extension is necessary. In this case, C is returned as a good partial controller (line 10).

2. If $\beta(h)$ returns false, it indicates that h is illegal, so no extension of C can be a valid controller. In this case, the current search branch fails, and the algorithm backtracks to the most recent non-deterministic choice point (see below), from where alternative choices are explored (line 11).

3. Otherwise, $\beta(h)$ must have returned unknown, indicating a legal but non-conclusive history, so further simulation of the controller is needed. If an action and transition is already specified in C for the current control state q and observation $\Omega(s)$, then we simply follow it by recursively calling AND_step with the current controller C and history h, but successor control state q' and the set S' of all possible successor world states (lines 12–14).

[1] $\langle\{1, \ldots N\}, 1, \emptyset\rangle$ to be precise, but we only denote the transition relation when there is no ambiguity, since Q and q_0 are fixed.

```
 1: C = synthesize_{E,N}(I)
 2:     return AND_step_{E,N}(∅, 1, I, ∅);
 3:
 4: AND_step_{E,N}(C, q, S, h)
 5:     for each s ∈ S
 6:         C := OR_step_{E,N}(C, q, s, h · ⟨q, s⟩);
 7:     return C;
 8:
 9: OR_step_{E,N}(C, q, s, h)
10:     if β(h) = true return C;
11:     else if β(h) = false fail;
12:     else if (q --Ω(s)/a--> q') ∈ C
13:         S' := {s' | s --a--> s'};
14:         return AND_step_{E,N}(C, q', S', h);
15:     else
16:         NON-DETERMINISTICALLY CHOOSE
17:             a ∈ A and 1 ≤ q' ≤ N;
18:         S' := {s' | s --a--> s'};
19:         C' := C ∪ {q --Ω(s)/a--> q'};
20:         return AND_step_{E,N}(C', q', S', h);
```

Figure 4: A generic algorithm for controller synthesis.

4. If, on the other hand, no action or transition is specified in C for q and $\Omega(s)$, then the algorithm makes a non-deterministic choice for a and q' (lines 16–17). We recursively call AND_step to handle all successor states in the same way as in the previous case, except that the new transition is appended to the controller before calling (lines 19–20).

The algorithm above is formulated using nondeterministic choices to reveal the compactness of our solution. It is not hard to see that the resulting search actually *strategically* enumerates all valid controllers with up to N states, *e.g.* never revisiting isomorphic (identical by state renaming) controllers and avoiding controllers with unreachable states, so we have:

Theorem 1 (Soundness and Completeness)**.** *Given environment $\mathcal{E}$ with initial states $\mathcal{I}$, and a behavior specification β, $\mathcal{C} =$ **synthesize**$_{\mathcal{E},N}(\mathcal{I})$ iff $\mathcal{C}$ is an N-bounded finite-state controller in $\mathcal{E}$ that satisfies β, up to isomorphism.*

Throughout this paper, we assume that the bound N is given. If not, an iterative deepening search over N could be used, which is guaranteed to terminate, since the environment, the controller and required behavior tests are all finite.

In practice, for the generic solver to work well, the nondeterministic choice at line 16 must be resolved wisely, making use of the structure of the target problem. For example, in most application domains, compact controllers are preferable, so one should try to reuse control states as much as possible; for planning tasks like generalized planning and planning programs, one could make use of domain-independent heuristics developed in the state-of-the-art planners (Hoffmann and Nebel 2001; Richter and Westphal 2010) for effective action selection. In the following, we will show how our generic solver can be instantiated to efficiently synthesize controllers in the different applications illustrated in the introduction.

Generalized Planning

Problem Formalization. Bonet *et al.* (2009) formalize the generalized planning problem (what they call "control problem") as $P = \langle F, I, A, G, R, O, D \rangle$, where

- F is a set of (primitive) fluents,
- I is a set of F-clauses representing the initial situation,
- A is a set of actions with conditional effects,
- G is a set of literals representing the goal situation,
- R is a set of non-primitive fluents,
- O is the set of observable fluents, $O \subseteq R$, and
- D is the set of axioms defining the fluents in R.

Without going into the details, *e.g.*, the evaluation of non-primitive fluents in R using the axioms in D, we note that this problem can be modeled in our framework by defining the dynamic environment as $\mathcal{E} = \langle \mathcal{A}, \mathcal{O}, \mathcal{S}, \mathcal{I}, \Delta, \Omega \rangle$, where

- $\mathcal{A} = A$, $\mathcal{O} = \mathbf{d}(O)$, $\mathcal{S} = \mathbf{d}(F)$, where $\mathbf{d}(V)$ is the cross product of the domains for all variables $v \in V$,
- $\mathcal{I} = \{s \in S \mid s \models I\}$,
- $\langle s, a, s' \rangle \in \Delta$ iff action a changes state s to s' in P,
- $\Omega(s) = o$ iff o is observed in state s in P.

Notice that with this definition, we can allow actions to have preconditions like in (Pralet et al. 2010), by eliminating illegal transitions, as well as non-deterministic effects.

The behavior specification accepts all legal execution histories leading to a state satisfying the goal, and rejects those that contain repeated configurations (indicating infinite loop) and that cannot be extended (indicating dead end). Formally,

$$\beta(\langle q_0, s_0 \rangle, \cdots, \langle q_k, s_k \rangle) = \begin{cases} \textsf{true} & \text{if } s_k \models G; \\ \textsf{false} & \text{if } \langle s_k, a, s' \rangle \notin \Delta \text{ for all } a \in \mathcal{A}, s' \in \mathcal{S}, \text{ or} \\ & \langle q_k, s_k \rangle = \langle q_i, s_i \rangle \text{ for some } 0 \leq i < k; \\ \textsf{unknown} & \text{otherwise.} \end{cases}$$

Solver adaptation. The structure of the generalized planning problems is very close to that of our generic problem, so the adaptation of the solver is straightforward. We implemented a planner in SWI-Prolog, whose body is shown in Figure 5.

After the list of all initial states is obtained by initStates/1, andStep/5 is activated to process each of the possible states in it (line 1). The parameters to andStep are the input plan, the output plan, the current control state, the list of current possible world states, and the execution history, respectively. (orStep below has similar parameters except that the fourth parameter is a single world state instead of a list of states.) Lines 3–5 here are the AND step (corresponding the lines 4–7 in Figure 4). Notice the use of variable P in line 5 for updating the current plan after each state is handled by orStep/5.

Lines 7–18 are the OR step (corresponding to lines 9–20 in Figure 4), with line 7 handling goal achievement, line 8 enforcing backtracking due to state repetition, lines 12–13 following existing transition, and 14–17 trying new transitions. In the last case, bestAct/2 succeeds with all possible actions for the current state, so heuristics could be

```
 1: plan(P) :- initStates(SL), andStep([], P, 1, SL, []).
 2:
 3: andStep(P, P, _, [], _).
 4: andStep(P0, P1, Q, [S|SL], H) :-
 5:     orStep(P0, P, Q, S, H), andStep(P, P1, Q, SL, H).
 6:
 7: orStep(P, P, _, S, _) :- goal(G), holds(G,S), !.
 8: orStep(_, _, Q, S, H) :- member(⟨Q, S⟩, H), !, fail.
 9: orStep(P0, P1, Q, S, H) :-
10:     observation(S, O), H' = [⟨Q, S⟩|H],
11:     (
12:         member(⟨Q, O, A, Q'⟩, P0), !, legalAct(A, S),
13:             nextStates(S, A, SL'), andStep(P0, P1, Q', SL', H');
14:         bestAct(A, S), nextStates(S, A, SL'), size(P0, M),
15:             (between(1, M, Q');
16:                 Q' is M + 1, bound(N), Q' =< N),
17:             andStep([⟨Q, O, A, Q'⟩|P0], P1, Q', SL', H')
18:     ).
```

Figure 5: Prolog code for generalized planning.

implemented here so that the most promising actions are unified first. For choosing the target control state Q' for the transition, line 15 enumerates all currently used states, before attempting to create a new one (not exceeding the bound) in line 16. More advanced ordering, *e.g.*, the "most-recently used state" heuristics, could be explored here for better search efficiency.

Theorem 2 (Correctness)**.** *If the predicate "plan(P)" succeeds, then P is an N-bounded plan that solves the generalized planning problem, and vice versa (up to isomorphism).*

Experimental Results. We run our planner on a set of benchmark problems, and compared its performance with the compilation approach of Bonet, Palacios and Geffner (BPG) (2009) and the constraint programming based planner (Dyncode) by Pralet *et al.* (2010). Since their planners are not publicly available, we did not rerun their experiments, but instead used the data in the respective paper directly, so this comparison only serves as a feasibility check for our approach, due to the different experiment settings explained in Table 1.

In the table, column N shows the number of control states required for the smallest plan, and for each planner, "Solve" is the solution time (in seconds) with N states, and "Prove" is the time (in seconds) needed for the planner to prove that no plan exists with less than N states. Surprisingly, the simple adaptation of our generic solver, which essentially performs depth-first search, achieves comparable performance to Dyncode, and works much faster than BPG in some cases. We believe that rewriting our planner in a compiled language like C and including effective heuristics will further improve the performance of our planner. Especially the latter may play a big role when we are faced with more difficult problems.

Service Composition

Problem formalization. We consider service composition problems (Calvanese et al. 2008) where the task is to realize a target service, represented by a finite-state machine M_T,

Problem	N	BPG Solve	Dyncode Solve	Dyncode Proof	Our solver Solve	Our solver Proof
Hall-A 1×4	2	0.0	0.01	0.02	0.01	0.0
Hall-A 4×4	4	5730.5	0.26	2.35	0.21	1.86
Hall-R 1×4	1	0.0	0.01	0	0.01	0
Hall-R 4×4	1	0.0	0.02	0	0.01	0
Prize-A 4×4	1	0.0	0.02	0	0.01	0
Corner-A 4×4	1	0.1	0.02	0	0.01	0
Prize-R 3×3	2	0.1	0.03	0.03	0.04	0.01
Prize-R 5×5	3	2.7	2.37	0.97	2.71	1.3
Corner-R 2×2	1	0.0	0.01	0	0.01	0
Corner-R 5×5	1	1.6	0.02	0	0.01	0
Prize-T 3×3	1	0.1	0.05	0	0.01	0
Prize-T 5×5	1	0.3	0.34	0	0.02	0
Blocks 6	2	0.8	0.02	0.02	0.02	0.0
Blocks 20	2	34.8	0.04	0.02	0.02	0.0
Visual-M $(8,5)$	2	1289.5	3.59	0.27	0.02	0.0
Gripper $(3,5)$	2	4996.1	0.06	0.02	0.01	0.0

Table 1: Comparison of generalized planning approaches. BPG is run on a Xeon 1.86GHz CPU with 2GB RAM, Dyncode on a Xeon 2GHz CPU with 1GB RAM, and our Prolog adaptation on an Intel Core2 3.0GHz CPU with 3GB RAM.

by choosing from a set of available services $M_1, \cdots, M_n$ at each step. The example in the introduction is a generalized case studied by De Giacomo *et al.* (2009), who define each service M_i as a tuple $\langle A, O_i, S_i, s_{i0}, F_i, \delta_i, o_i\rangle$, where

- A is the (shared) set of actions;
- O_i is a set of observations;
- S_i is a set of service states;
- $s_{i0} \in S_i$ is the initial service state;
- $F_i \subseteq S_i$ is a set of final states, where the service is allowed to terminate;
- $\delta_i \subseteq S_i \times A \times S_i$ is the transition relation of the service;
- $o_i : S_i \to O_i$ is the observation function.

To formalize their definition in our framework, we model the joint behavior of the services as a dynamic environment $\mathcal{E} = \langle \mathcal{A}, \mathcal{O}, \mathcal{S}, \mathcal{I}, \Delta, \Omega\rangle$, where

- $\mathcal{A} = \{1, \cdots, n\}$
- $\mathcal{O} = O_1 \times \cdots \times O_n \times A$;
- $\mathcal{S} = S_T \times S_1 \times \cdots \times S_n \times A$;
- $\mathcal{I} = \{\langle s_{T0}, s_{10}, \cdots, s_{n0}, a\rangle \mid \langle s_{T0}, a, \cdot\rangle \in \delta_T\}$,
- $\Delta(\langle s_T, s_1, \cdots, s_n, a)\rangle, k, \langle s'_T, s'_1, \cdots, s'_n, a'\rangle)$ iff
  1. $\delta_T(s_T, a, s'_T)$,
  2. $\delta_k(s_k, a, s'_k)$,
  3. $s_i = s'_i$ for all $i \notin \{T, k\}$, and
  4. $\delta_T(s'_T, a', s)$ for some $s \in S_T$.
- $\Omega(\langle s_T, s_1, \cdots, s_n, a\rangle) = \langle o_1(s_1), \cdots, o_n(s_n), a\rangle$.

Notice that the state of the target service M_T is not observed, but the orchestrator can keep track of it using its internal states, since the target service is deterministic. Intuitively, the behavior specification needs to consider the following factors *(i)* at any time, the requested action (stored as the last element of a state tuple) must be executable by at least one service; *(ii)* (final state constraint:) whenever the target service is in a final state, so must be all composing services; *(iii)* (state repetition:) if we reach a configuration (control state and world state combined) that has been visited already in the execution history, then all future executions from the current configuration would be handled in the same way as from its previous occurrence, so there is no need to consider further extensions.

Formally, we define the behavior specification as

$$\beta(\langle q_0, \sigma_0\rangle, \cdots, \langle q_k, \sigma_k\rangle) = \begin{cases} \mathsf{true} & \text{if } q_i = q_k \text{ and } \sigma_i = \sigma_k \text{ for some } 0 \le i < k; \\ \mathsf{false} & \text{if } \langle \sigma_k, a, \sigma\rangle \notin \Delta \text{ for all } a \in \mathcal{A}, \sigma \in \mathcal{S}, \text{ or} \\ & \sigma_k = \langle s_T, s_1, \cdots, s_n\rangle \text{ where } s_T \in F_T \\ & \text{but } s_i \notin F_i \text{ for some } i \in \{1, \cdots, n\}; \\ \mathsf{unknown} & \text{otherwise.} \end{cases}$$

Solver adaptation. Although the non-determinism of the environment is specified as a single transition relation in our formalization above, it is not hard to see that the non-determinism comes from two different sources, namely, the uncertainty as which action the target may request, and the nondeterministic effect of each composing service. During our instantiation of the generic solver, we take this special structure of the problem into account, by creating two AND steps, one for each source of non-determinism above. This dramatically reduces the observation cases in the search nodes, since only the state of the chosen service changes and thus needs to be observed, and observations on all other services can be safely ignored. Exploiting this structure makes the branching factor of the AND nodes exponentially smaller, and thus contributes to a much more efficient solver.

Following these intuitions, we implemented a general solver for service composition problems in Prolog, the body of which is shown in Figure 6, where procReqs/7 (lines 10–15) is the AND step for processing all possible requests from the target and procTrans/9 (lines 22-26) handle all nondeterministic effects of the executed service. Following a similar idea, we also separate the OR step into chuzServ/7 (lines 17–20) which tries all possible services and chuzTran/9 which tries the control state to transition to next. Finally, procStates/6 (lines 4–8) checks the behavior specification according to the history and the current states.

Theorem 3 (Correctness). *If the predicate "compose(C)" succeeds, then C is an N-bounded orchestrator that realizes the composition, and vice versa (up to isomorphism).*

The problem is known to be EXPTIME-complete, even with complete observability. Moreover, from an analysis of the composition technique in (De Giacomo, De Masellis, and Patrizi 2009), one can conclude that the size of the orchestrator is bounded by the size of target services in the fully observable case, and by the size of the Cartesian product of the powerset of the states of the target and the available services in the partially observable case. As a result, if we perform an iterative deepening search starting with N equal to 1 and stopping when we reach the above bound, we get a sound and complete technique to compute compositions, which has the notable property of computing the smallest orchestrator possible.

Experimental Results. We experimented our solver on 18 benchmark problems on an Intel Core2 3.0GHz CPU

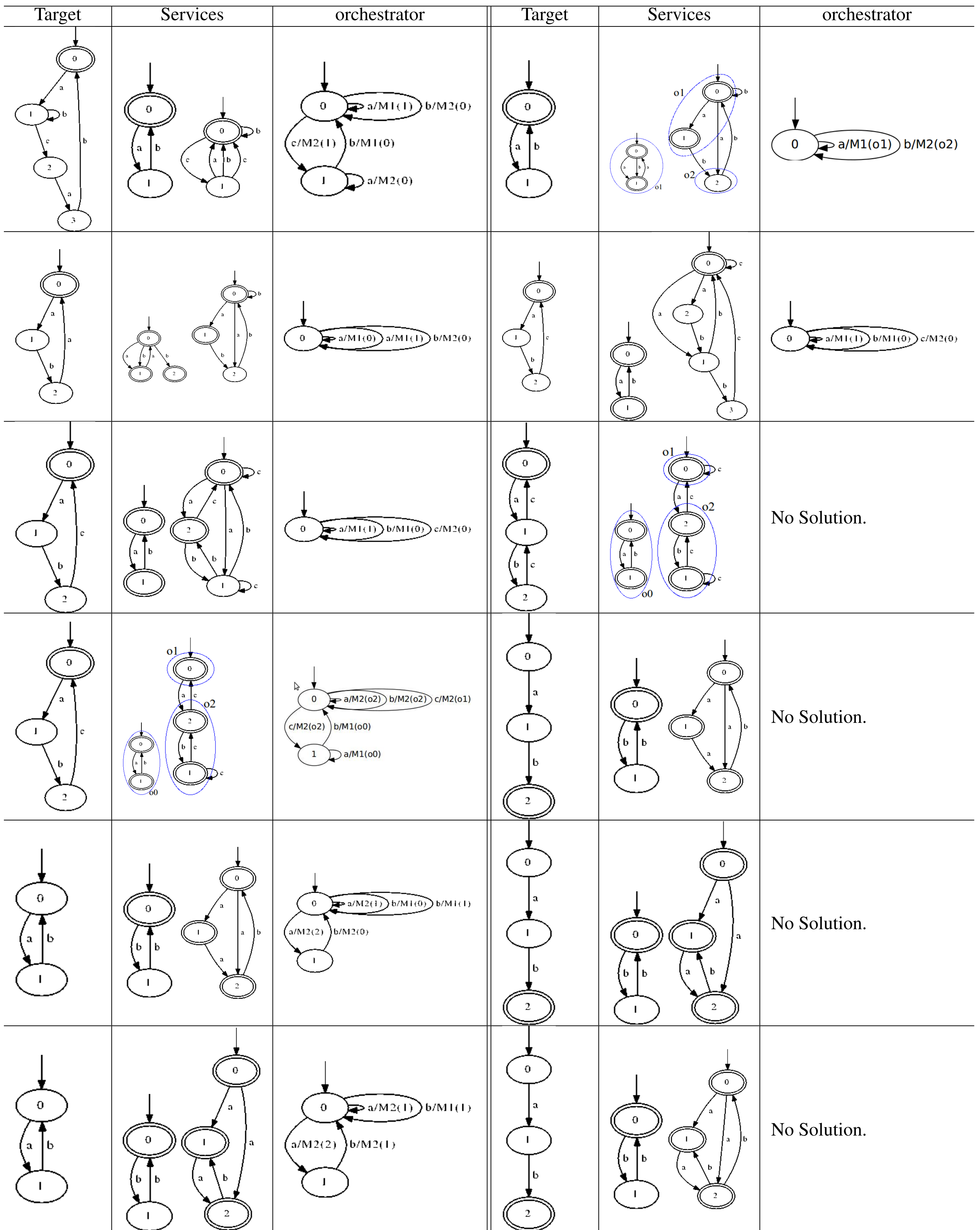

Figure 7: Example composition problems used in our experiments. Labels of sensing results are omitted when the service is fully observable.

```
1: compose(C) :-
2:     procState(s_T0, ⟨s_10, ··· , s_n0⟩, 1, [], [], C).
3:
4: procState(S_T, SL, Q, H, C, C) :-
5:     member(⟨S_T, SL, Q⟩, H), !.
6: procState(S_T, SL, Q, H, C0, C1) :-
7:     (isFinal(S_T)->allFinal(SL);true),
8:     procReqs(S_T, δ_T, SL, Q, [⟨S_T, SL, Q⟩|H], C0, C1).
9:
10: procReqs(_, [], _, _, _, C, C).
11: procReqs(S_T, [⟨S_T, A, S'_T⟩|T], SL, Q, H, C0, C1) :- !,
12:     chuzServ(S'_T, SL, Q, A, H, C0, C),
13:     procReqs(S_T, T, SL, Q, H, C, C1).
14: procReqs(S_T, [_|T], SL, Q, H, C0, C1) :-
15:     procReqs(S_T, T, SL, Q, H, C0, C1).
16:
17: chuzServ(S_T, SL, Q, A, H, C0, C1) :-
18:     (member(⟨Q, A, K, _, _⟩, C0), !; between(1,n,K)),
19:     findall(X,member(⟨S_K, A, X⟩, δ_K),SL'_K),
20:     procTrans(S_T, SL, Q, A, K, SL_K, H, C0, C1).
21:
22: procTrans(_, _, _, _, _, [], _, C, C).
23: procTrans(S_T, SL, Q, A, K, [S'_K|SL'_K], H, C0, C1) :-
24:     replace(SL, S_K, S'_K, SL'), observe(S'_K, O),
25:     chuzTran(S_T, SL', Q, A, K, O, H, C0, C),
26:     procTrans(S_T, SL, Q, A, K, SL'_K, H, C, C1).
27:
28: chuzTran(S_T, SL, Q, A, K, O, H, C0, C1) :-
29:     member(⟨Q, A, K, O, Q'⟩,C0), !,
30:     procState(S_T, SL, Q', H, C0, C1).
31: chuzTran(S_T, SL, Q, A, K, O, H, C0, C1) :-size(C0, M),
32:     (between(1, M, Q'); Q' is M + 1, bound(N), Q' ≤ N),
33:     procState(S_T, SL, Q', H, [⟨Q, A, K, O, Q'⟩|C0], C1).
```

Figure 6: Prolog code for service composition.

with 3.0GB memory. The problems requires the composition of target services of 2–4 states from 2–5 available services ranging from 2 to 10 states. Twelve of the smaller problem instances used in the experiments are shown in Figure 7, and the full list of problems is available online at `http://www.cs.toronto.edu/~yuxiao/compose.tgz`. All problems are either solved or proved unsolvable within less than 0.01 second. This matches the results obtained by model checking for problems with full observability, and is much faster, finding smaller orchestrators, for case with partial observability (De Giacomo, De Masellis, and Patrizi 2009). The obtained results are not definitive since the existing benchmarks are quite small, but certainly they show the effectiveness of the proposed approach in practice.

Planning Programs

Problem Formalization. Next, we move on to planning programs, like the researcher's example in the introduction, recently proposed by De Giacomo *et al.* (2010). According to them, a *domain* is a tuple $D = \langle P, A, S_0, \rho\rangle$, where P is a set of propositions, A is a set of actions, $S_0 \in 2^P$ is the initial state, and $\rho \subseteq 2^P \times A \times 2^P$ is the set of transitions. A *planning program* for D is a tuple $\mathcal{T} = \langle T, G, t_0, \delta\rangle$, where

- $T = \{t_0, \ldots, t_q\}$ is the finite set of *program states*;
- G is a finite set of goals of the form *"achieve ϕ while maintaining ψ,"* denoted by pairs $g = \langle \psi, \phi\rangle$, where ψ and ϕ are propositional formulae over P;
- $t_0 \in T$ is the *program initial state*;
- $\delta \subseteq T \times G \times T$ is the *program transition relation*.

This synthesis problem for planning programs can be modeled in our frame work as $\mathcal{E} = \langle \mathcal{A}, \mathcal{O}, \mathcal{S}, \mathcal{I}, \Delta, \Omega\rangle$, where

- $\mathcal{A} = A$ is their set of available actions;
- $\mathcal{O} = \mathcal{S} = 2^P \times T \times G \times T$;
- $\mathcal{I} = \{\langle S_0, t_0, g, t\rangle \mid \langle t_0, g, t\rangle \in \delta\}$;
- $\Delta(\langle s, t, \langle\psi, \phi\rangle, t'\rangle, a, \langle s', t, \langle\psi, \phi\rangle, t'\rangle)$ iff
  (i) $\langle s, a, s'\rangle \in \rho$,
  (ii) $s \models \psi$ and $s' \models \psi$, and
  (iii) $s' \not\models \phi$;
- $\Delta(\langle s, t, \langle\psi, \phi\rangle, t'\rangle, a, \langle s', t', g, t''\rangle)$ iff
  (i) $\langle s, a, s'\rangle \in \rho$,
  (ii) $s \models \psi$ and $s' \models \psi$,
  (iii) $s' \models \phi$,
  (iv) $\langle t', g, t''\rangle \in \delta$;
- $\Omega(\sigma) = \sigma$.

Let $\sigma_i = \langle s_i, t_i, \langle\psi_i, \phi_i\rangle, t'_i\rangle$, the behavior specification is

$$\beta(\langle q_0, \sigma_0\rangle, \cdots, \langle q_k, \sigma_k\rangle) = \begin{cases} \mathsf{false} & \text{if } s_k \not\models \psi_k, \text{ or } \langle\sigma_k, a, \sigma\rangle \notin \Delta \text{ for all } a, \sigma, \text{ or} \\ & \sigma_i = \sigma_k \text{ for some } 0 \le i < k \text{ and } t_k = t_{k-1}; \\ \mathsf{true} & \text{if } \sigma_i = \sigma_k \text{ if for some } 0 \le i < k \\ & \text{and } t_k \neq t_{k-1}; \\ \mathsf{unknown} & \text{otherwise.} \end{cases}$$

Solver adaptation. Like in service composition, a controller for a planning program is faced with two sources of nondeterminism in each cycle, namely, the uncertainty about goal request, and the nondeterministic effects of the actions. Figure 8 shows the body of the Prolog code for planning programs, where procGoals/6 (lines 8–11) is the AND step for the former, and procEffects/9 (lines 24–27) for the latter source of nondeterminism. planGoal/9 (lines 13–22) implements the OR step that makes the action choices, where bestAct/3 (line 20) returns the most promising action by using the additive heuristics (Bonet and Geffner 2001).

Theorem 4 (Correctness). *If the program "plan(P)" succeeds, then P is a controller that realizes the planning program $\mathcal{T}$ in the domain D, and vice versa (up to isomorphism).*

The problem is known to be EXPTIME-complete. Moreover, according to the analysis by De Giacomo *et al.* (2010), the size of the controller can be bounded by 1. Interestingly, if we assume that the planning programs to be deterministic, we can avoid observing the target as in the case of service composition, and then the controller may need multiple states to remember the state of the target, in which case a conservative bound for N can be identified similarly.

Experimental Results. We use our planner to solve the researcher's world example on an Intel Core2 3.0Hz CPU with 3GB RAM, and it generated the correct solution within

```
 1: plan(P) :- initState(S), procState(0, S, [ ], [ ], P).
 2:
 3: procState(T, S, U, C, C) :- member(⟨T, S⟩, U), !.
 4: procState(T, S, U, C0, C1) :-
 5:     findall(⟨M, G, T'⟩, goal(T, M, G, T'), GL), !,
 6:     procGoals(T, GL, S, [⟨T, S⟩|U], C0, C1).
 7:
 8: procGoals(_, [ ], _, _, C, C).
 9: procGoals(T, [⟨M, G, T'⟩|GL], S, U, C0, C1) :-
10:     planGoal(T, M, G, T', S, [ ], U, C0, C),
11:     procGoals(T, GL, S, U, C, C1).
12:
13: planGoal(_, M, _, _, S, _, _, _, _) :- \+ holds(M, S), !, fail.
14: planGoal(_, _, _, _, S, H, _, _, _) :- member(S, H), !, fail.
15: planGoal(T, M, G, T', S, _, _, C, C) :-
16:     member(⟨T, M, G, T', S, _⟩, C), !.
17: planGoal(_, _, G, T, S, _, U, C0, C1) :-
18:     holds(G, S), !, procState(T, S, U, C0, C1).
19: planGoal(T, M, G, T', S, H, U, C0, C1) :-
20:     bestAct(G, A, S), nextStates(S, A, SL'),
21:     H' = [S|H], C = [⟨T, M, G, T', S, A⟩|C0],
22:     procEffects(T, M, G, T', SL', H', U, C, C1).
23:
24: procEffects(_, _, _, _, [ ], _, _, C, C).
25: procEffects(T, M, G, T', [S|SL], H, U, C0, C1) :-
26:     planGoal(T, M, G, T', S, H, U, C0, C),
27:     procEffects(T, M, G, T', SL, H, U, C, C1).
```

Figure 8: Prolog code for planning programs

0.04 second, while the model checking approach in (De Giacomo, Patrizi, and Sardina 2010) requires several minutes.

Recently, De Giacomo *et al.* (2011) applied this algorithm on a much more complicated smart home application, which involves routinely controlling the services (*e.g.* controlling air temperature, filling and emptying bathtub, *etc.*) in an intelligent home environment shown in Figure 9. Their experimental results show that this algorithm solves the challenging task, with far better efficiency than the existing model-checking based approaches.

Conclusion and Future Work

We have presented a generic framework and a related solver for the synthesis of bounded controllers, which can be instantiated for several diverse problems. The solver appears to be quite effective, surpassing specific techniques, typically based on compilation, constraint programming, or model checking technology. Moreover, it is very suitable for heuristic pruning of the search space, a feature that we only partially exploited in this work. Given the promising results, we are currently eager to experiment (instantiations of) our solver in more real cases, *e.g.*, to explore industrial/web service composition and to extend the smart home applications. Regarding planning programs, recently Gerevini *et al.* (2011) proposed an approach based on repetitively solving planning problems to handle planning programs running over deterministic domains. We are indeed quite interested in comparing it with our generic solver. Also, our approach is based on finite history, though all the problem considered here can be seen as special cases of LTL synthesis (Vardi 1996). It remains open whether our approach can be extended to handle full LTL synthesis, despite LTL requiring infinite runs (*vs.* finite histories), using *e.g.* Schuppan and Biere's result (2004).

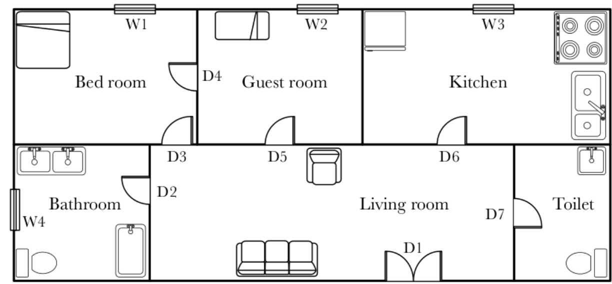

Figure 9: The smart home environment in (De Giacomo et al. 2011).

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129:5–33.

Bonet, B.; Palacios, H.; and Geffner., H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *ICAPS*.

Calvanese, D.; De Giacomo, G.; Lenzerini, M.; Mecella, M.; and Patrizi, F. 2008. Automatic service composition and synthesis: the roman model. *IEEE Data Eng. Bull.* 31(3):18–22.

De Giacomo, G.; Ciccio, C. D.; Felli, P.; Hu, Y.; and Mecella, M. 2011. Goal-based process composition. In preparation, available from the authors.

De Giacomo, G.; De Masellis, R.; and Patrizi, F. 2009. Composition of partially observable services exporting their behaviour. In *ICAPS*.

De Giacomo, G.; Patrizi, F.; and Sardina, S. 2010. Agent programming via planning programs. In *AAMAS*.

Gerevini, A. E.; Patrizi, F.; and Saetti, A. 2011. An effective approach to realizing planning programs. In *ICAPS*. (to appear).

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practise*. Morgan Kaufmann.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Levesque, H.; R., R.; Lesperance, Y.; F., L.; and R., S. 1997. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming* 31:59–84.

Levesque, H. 2005. Planning with loops. In *ICAPS*.

Pralet, C.; Verfaillie, G.; Lemaître, M.; and Infantes, G. 2010. Constraint-based controller synthesis in non-deterministic and partially observable domains. In *ECAI*.

Richter, S., and Westphal, M. 2010. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127177.

Schuppan, V., and Biere, A. 2004. Efficient reduction of finite state model checking to reachability analysis. *STTT* 5(2-3):185–204.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning generalized plans using abstract counting. In *AAAI*.

Vardi, M. Y. 1996. An Automata-Theoretic Approach to Linear Temporal Logic. In *Logics for Concurrency: Structure versus Automata*. Springer.