On Computing Conformant Plans Using Classical Planners: A Generate-And-Complete Approach*

Khoi Nguyen and **Vien Tran** and **Tran Cao Son** and **Enrico Pontelli**
Computer Science Department
New Mexico State University
Las Cruces, New Mexico, USA
knguyen—vtran—tson—epontell@cs.nmsu.edu

Abstract

The paper illustrates a novel approach to *conformant planning* using classical planners. The approach relies on two core ideas developed to deal with incomplete information in the initial situation: the use of a classical planner to solve non-classical planning problems, and the reduction of the size of the initial belief state. Differently from previous uses of classical planners to solve non-classical planning problems, the approach proposed in this paper creates a valid plan from a *possible* plan—by inserting actions into the possible plan and maintaining only one level of non-deterministic choice (i.e., the initial plan being modified). The algorithm can be instantiated with different classical planners—the paper presents the GC[LAMA] implementation, whose classical planner is LAMA. We investigate properties of the approach, including conditions for completeness. GC[LAMA] is empirically evaluated against state-of-the-art conformant planners, using benchmarks from the literature. The experimental results show that GC[LAMA] is superior to other planners, in both performance and scalability. GC[LAMA] is the only planner that can solve the largest instances from several domains. The paper investigates the reasons behind the good performance and the challenges encountered in GC[LAMA].

Introduction

Conformant planning is the problem of computing a sequence of actions that achieves a goal in presence of incomplete information about the initial state (Smith and Weld 1998). Recent research shows that conformant planning could be very useful in the construction of finite-state controllers (Bonet, Palacios, and Geffner 2009) and in contingent planning (Albore, Palacios, and Geffner 2009). One of the most difficult issues, that directly affects the performance and scalability of conformant planners, is the size of the initial belief state—which is often exponential in the number of object constants of the problem. We observed that in many problems drawn from the recent *International Planning Competitions (IPC)* and from the literature, the initial belief states of many large instances contain more than 2^{10} states, creating challenges to existing conformant planners.

Various techniques have been developed to deal with the potentially huge size of the belief state. Some planners employ a compact representation of belief states—e.g., CFF (Brafman and Hoffmann 2004), POND (Bryce, Kambhampati, and Smith 2006), CNF (To, Son, and Pontelli 2010). Other planners develop simplification techniques that can reduce the size of the initial belief state, sometimes by several orders of magnitude—as in CPA (Tran et al. 2009) and DNF (To, Pontelli, and Son 2009). Most of these planners search for solutions in the belief state space. An alternative approach has been proposed in (Castellini, Giunchiglia, and Tacchella 2001) and (Kurien, Nayak, and Smith 2002), where the conformant planning problem is viewed as a set of *sub-problems*, which are classical planning problems, and solutions are computed using a two-step approach.

C-PLAN, developed in (Castellini, Giunchiglia, and Tacchella 2001), starts by computing a solution for a sub-problem, called a *possible plan*, using a SAT-planner. It then checks whether the possible plan is a solution of the original problem. If it is not, the possible plan is discarded, a new possible plan is generated, and the process continues. We refer to this approach as *generate-and-test*. To improve the performance, C-PLAN implements different strategies to limit the number of possible plans that need to be generated.

FRAG-PLAN, proposed in (Kurien, Nayak, and Smith 2002), follows a slightly different approach in computing plans. It begins with the computation of a possible plan and then attempts to extend it to a valid plan. During the extension phase, FRAG-PLAN assumes that a conformant plan for k initial states has been found, selects fragments of this plan, and uses them as the seed to find a conformant plan for $k+1$ sub-problems where $k = 1, 2, \ldots, n-1$ and n is the size of the initial belief state. The main concern in FRAG-PLAN is the potentially huge number of backtracking steps that need to be performed. We refer to the approach used in FRAG-PLAN as *generate-and-extend*.

Both C-PLAN and FRAG-PLAN do not use any technique to reduce the number of initial states. The experimental evaluation in (Kurien, Nayak, and Smith 2002) shows that both planners work well in some domains, but their coverage is limited, and both planners do not scale up well.

In this paper, we propose an alternative approach to the generate-and-extend approach of (Kurien, Nayak, and Smith 2002). We refer to the new approach as *generate-and-complete*. The approach is similar to generate-and-extend, as it first generates a possible plan for a sub-problem and

*Partially supported by NSF-IIS 0812267 grant.

then uses it to construct a valid plan by considering other initial states. In the second phase, our approach *repairs* the possible plan, whenever necessary, to create a possible plan for other sub-problems, one-by-one. The possible plan for the last sub-problem will be checked for being a valid conformant plan. If it is not a solution, a new possible plan for the first sub-problem is generated and the completion process restarted. The key difference between our approach and the approach of FRAG-PLAN lies in that our approach maintains a non-deterministic choice only on the plan for one initial state, and it does not attempt to incrementally compute a conformant plan for increasing subsets the the initial belief state at each iteration. Furthermore, our approach employs the `one-of`-combination technique described in (Tran et al. 2009) to reduce the number of possible initial states that need to be considered.

We develop a generate-and-complete algorithm and implement it in a conformant planner, called GC[LAMA], based on the code of the LAMA classical planner.[1] GC[LAMA] is evaluated using benchmarks from the literature and recent IPCs, and compared to state-of-the-art conformant planners. The results show that GC[LAMA] performs exceptionally well in almost all domains and scales up better than other planners. We discuss the reasons behind the success of GC[LAMA] and identify possible weaknesses.

The Conformant Planning Problem

A conformant planning problem P is specified by a tuple $\langle F, O, I, G\rangle$, where F is a set of propositions, O a set of action descriptions, I a set of formulae describing the initial state of the world, and G a formula describing the goal.

A *literal* is a proposition $p \in F$ or its negation $\neg p$. $\bar{\ell}$ denotes the complement of the literal ℓ—i.e., $\bar{\ell} = \neg\ell$, where $\neg\neg p = p$ for $p \in F$. For a set of literals L, $\overline{L} = \{\bar{\ell} \mid \ell \in L\}$, and L is often used to represent $\wedge_{\ell\in L}\ell$.

A set of literals X is *consistent* if there is no $p \in F$ s.t. $\{p, \neg p\} \subseteq X$. A *state* s is a consistent and *complete* set of literals, i.e., s is consistent, and for each $p \in F$, $p \in s$ or $\neg p \in s$. A *belief state* is a set of states. A set of literals X satisfies a literal ℓ (a set of literals Y) iff $\ell \in X$ ($Y \subseteq X$).

Each action a in O has a precondition, denoted by $pre(a)$, and a set of conditional effects of the form $\psi \rightarrow \ell$ (denoted by $a : \psi{\rightarrow}\ell$), where $pre(a)$ and ψ are sets of literals and ℓ is a literal. We often write $a : \psi \rightarrow \ell_1, \ldots, \ell_k$ as a shorthand for the set $\{a : \psi \rightarrow \ell_1, \ldots, a : \psi \rightarrow \ell_k\}$.

The initial state I is a set of literals, `one-of` clauses (each of the form $\texttt{one-of}(\psi_1, \ldots, \psi_n)$), and `or` clauses (each of the form $\texttt{or}(\psi_1, \ldots, \psi_m)$), where each ψ_i is a set of literals.

A set of literals X satisfies the `one-of` clause $\texttt{one-of}(\psi_1, \ldots, \psi_n)$ if there exists some i, $1 \leq i \leq n$, such that $\psi_i \subseteq X$ and for every $j \neq i$, $1 \leq j \leq n$, $\overline{\psi_j} \cap X \neq \emptyset$. X satisfies the `or` clause $\texttt{or}(\psi_1, \ldots, \psi_m)$ if there exists some $1 \leq i \leq m$ such that $\psi_i \subseteq X$.

By $ext(I)$ we denote the set of all states satisfying every literal in I, every `one-of` clause in I, and every `or` clause in I. E.g., if $F{=}\{g, f, h\}$ and $I{=}\{\texttt{or}(g, h), \texttt{one-of}(f, h)\}$[2] then $ext(I) = \{\{g, h, \neg f\}, \{g, \neg h, f\}, \{\neg g, h, \neg f\}\}$.

The goal G is a collection of literals and `or` clauses.

Given a state s and an action a, a is executable in s if $pre(a) \subseteq s$. A conditional effect $a : \psi \rightarrow \ell$ is applicable in s if $\psi \subseteq s$. The set of effects of a in s, denoted by $e_a(s)$, is defined as: $e_a(s) = \{\ell \mid a : \psi \rightarrow \ell \in O \text{ is applicable in } s\}$. The execution of a in a state s results in a successor state $succ(a, s)$, where $succ(a, s) = (s \cup e_a(s)) \setminus \overline{e_a(s)}$ if a is executable in s, and $succ(a, s) = \mathbf{failed}$, otherwise. Using this function, we define $\widehat{succ}$ for computing the state resulting from the execution of a sequence of actions $\alpha = [a_1, \ldots, a_n]$: $\widehat{succ}(\alpha, s) = s$ if $n = 0$; $\widehat{succ}(\alpha, s) = succ(a_n, \widehat{succ}([a_1, \ldots, a_{n-1}], s))$ if $n > 0$; and $\widehat{succ}(\gamma, \mathbf{failed}) = \mathbf{failed}$ for any sequence of actions γ. For a belief state S and action sequence α, let $\widehat{succ}^*(\alpha, S) = \{\widehat{succ}(\alpha, s) \mid s \in S\}$ if $\widehat{succ}(\alpha, s) \neq \mathbf{failed}$ for every $s \in S$; and $\widehat{succ}^*(\alpha, S) = \mathbf{failed}$, otherwise. α is a *solution* of P iff $\widehat{succ}^*(\alpha, ext(I)) \neq \mathbf{failed}$ and G is satisfied in every state belonging to $\widehat{succ}^*(\alpha, ext(I))$.

Intuition

In this section, we present the basic idea of GC[LAMA]. First, we introduce the notion of a sub-problem.

Definition 1. *Let $P = \langle F, O, I, G\rangle$ be a conformant planning problem. For each $s \in ext(I)$, the planning problem $P(s) = \langle F, O, s, G\rangle$ is a* sub-problem *of P. A solution of a sub-problem $P(s)$ of P is called a* possible plan *of P.*

It is easy to see that, for every $s \in ext(I)$, the problem $P(s)$ is a classical planning problem. The following observation is an obvious consequence of the definition of a solution of a conformant planning problem.

Observation 1. *Let $P = \langle F, O, I, G\rangle$ be a conformant planning problem and $P(s)$ be a sub-problem of P. Then, every solution of P is also a solution of $P(s)$.*

This property has been used in the development of C-PLAN (Castellini, Giunchiglia, and Tacchella 2001) and FRAG-PLAN (Kurien, Nayak, and Smith 2002). In essence, C-PLAN uses Algorithm 1.

Algorithm 1 C-PLAN(P)

```
1: Input: A planning problem P = ⟨F, O, I, G⟩
2: Output: A solution for P
3: repeat
4:    Compute a possible plan α of P
5:    if α is a solution of P then
6:       return α
7:    end if
8: until every possible plan of P has been considered
9: return failed
```

For efficiency, C-PLAN implements a variety of techniques to reduce the number of possible plans that need to be generated (Line (4)) without compromising completeness. A

[1] Note that we do not use the landmark feature of LAMA, only its FF heuristic and multi-valued variable representation.

[2] For simplicity we omit the singleton set notation.

similar idea is present in (Kuter et al. 2008), applied to the context of non-deterministic planning.

The authors of FRAG-PLAN observed that C-PLAN immediately eliminates a possible plan α of P from consideration when it is not a solution of P, indicating that the action sequence is "useless"—nevertheless, there might be parts of α that are useful. This led to the development of FRAG-PLAN, which tries to extend α to construct a solution before dismissing it as useless. While the idea seems reasonable, the approach has some issues of its own. First, the extension phase needs to decide which combinations of plans from the individual initial states should be used. Second, backtracking is required if the current combination is not promising. This is problematic, since the number of possible combinations of fragments of an action sequence is exponential in its size (or in the length of the plan).

In this paper, we propose a compromise between the approaches of FRAG-PLAN and C-PLAN—we employ the two phases of FRAG-PLAN but simplify its extension phase by:

- Considering the possible plan as a single fragment (instead of a collection of fragments, as in FRAG-PLAN) when the extension phase starts; and
- Extending the possible plan to generate a *new possible plan* for other initial states.

The first item removes the burden of having to decide which fragments should be used and what is the order among them (reducing backtracking). This also avoids the rigidity of FRAG-PLAN, that imposes a fixed ordering among fragments in the newly generated plan, and thus requires backtracking when the placement of the fragments is not suitable even though the fragments are useful. The second item relaxes the requirement of immediately generating a conformant plan—by focusing instead on possible plans for the various initial states.

Example 1. Consider the problem

$$P = \langle \{f, p, q, r, h\}, O, I, \{h\} \rangle$$

where

$$O = \left\{ \begin{array}{ll} a : \top \rightarrow p, r & b : \top \rightarrow q, \neg f \\ c : \top \rightarrow \neg f, \neg q, r & k : \top \rightarrow h \end{array} \right\}$$

with $pre(a) = \{q\}$, $pre(b) = \{f\}$, $pre(c) = \{f, q\}$, and $pre(k) = \{r\}$; and

$$I = \{\texttt{one-of}(q, \neg q), \neg p, \neg r, f, \neg h\}.$$

Here, $ext(I) = \{s_0, s_1\}$ with $s_0 = \{q, \neg p, \neg r, f, \neg h\}$ and $s_1 = \{\neg q, \neg p, \neg r, f, \neg h\}$ ($\top$ stands for true).

Let us assume that s_0 is selected to start the search for a solution. Let us consider two scenarios:

- The possible plan $\alpha_1 = [a; k]$ is generated. α_1 is not a solution of P because it is not a solution of $P(s_1)$. Thus, we will attempt to find a solution for $P(s_1)$ which has α_1 as a subsequence. This is done by executing α_1 from s_1. Since $pre(a)$ is not satisfied in s_1, we would like to find a plan that achieves $pre(a)$ from s_1. This process yields $[b]$. If we insert b before a, we obtain the sequence $\beta = [b; a; k]$ which is executable in s_1. Incidentally, β is also a solution of $P(s_1)$—i.e., β is a possible plan of $P(s_1)$.

  A validity test reveals that β is indeed a solution of P, and no other possible plans need to be explored.
- Let us assume that $\alpha_2 = [c; k]$ is generated instead. We can easily check that α_2 is not a solution of P, since it is not a solution of $P(s_1)$. Again, we will try to create a solution for $P(s_1)$ which has α_2 as a subsequence. Similarly to the previous scenario, we would like to find a plan that achieves $pre(c)$ from s_1. This will be unsuccessful, since the only action that can generate q, a precondition of c, is action b. However, b will make f false, and there is no action that can generate f. We can quickly dismiss α_2 and request another possible plan of $P(s_0)$. □

Note that we might need to invoke the classical planner more than once in the completion phase to generate possible plans.

Example 2. Let us consider a modification of Example 1:

$$P = \langle \{f, p, q, r, h\}, O, I, \{h\} \rangle$$

where

$$O = \left\{ \begin{array}{ll} a : f \rightarrow p, r & a : \neg f \rightarrow f, \neg r \\ b : \top \rightarrow q, \neg f & \\ c : \top \rightarrow \neg f, \neg q, r & k : \top \rightarrow h \end{array} \right\}$$

with $pre(a) = \{q\}$, $pre(b) = \{f\}$, $pre(c) = \{f, q\}$, and $pre(k) = \{r\}$; and

$$I = \{\texttt{one-of}(q, \neg q), \neg p, \neg r, f, \neg h\}.$$

Here, $ext(I) = \{s_0, s_1\}$ with $s_0 = \{q, \neg p, \neg r, f, \neg h\}$ and $s_1 = \{\neg q, \neg p, \neg r, f, \neg h\}$. The only difference is that the execution of a yields different results in different contexts.

Assume that we select s_0, generate $\alpha_1 = [a; k]$ as a possible plan, and try to find a solution of $P(s_1)$ which has α_1 as a subsequence. As before, since a is not executable in s_1, we need to insert b before a to achieve the precondition of a. Unlike the previous example, the execution of $[b; a]$ in s_1 results in a state in which action k is no longer executable. This requires the insertion of c before k. The sequence $[b; a; c; k]$ is a solution of $P(s_1)$, and a solution of P as well. □

These examples illustrate the differences between our approach and FRAG-PLAN. For the first scenario in Example 1, to search for a plan of length 3, FRAG-PLAN may have to consider nine possible combinations of all fragments of $[a; k]$; one of them is $[*; a; k]$, where $*$ indicates a missing action, which will lead to the solution. For the second scenario, for plan of length 3, FRAG-PLAN could find a solution of the problem from the combination of fragments $[*; *; k]$. Instead, we dismiss the possible plan $[c; k]$ and compute another possible plan. Example 2 shows that the classical planner is called more than once in the second phase.

Formalizing the Algorithm

The high-level idea of our approach, as discussed above, relies on searching for a conformant plan by inserting actions into a possible plan. The two critical issues are: (*i*) *where* to insert an action or an action sequence; and (*ii*) *how* to determine them. This section will address these issues.

By $reduct(P)$ we denote the set of initial states obtained by applying the `one-of`-combination technique of (Tran et

al. 2009). This technique helps in reducing the size of the initial belief state in several domains, by identifying pairs of `one-of` clauses whose choices are independent—and can thus be combined in a single `one-of` clause. For a thorough evaluation of the impact of `one-of`, the interested reader is referred to (Tran et al. 2012).

Our algorithm has two parameters—the classical planner used to compute possible plans, denoted by Ω, and the conformant planning problem P. Observe that all state-of-the-art sound and complete classical planners have the following properties: they return (*a*) One or some solutions of the problem if the problem is solvable; (*b*) **Failed** if the problem is unsolvable. As such, we can assume that Ω is a sound and complete classical planner. For the sake of simplicity, we denote with $\Omega(X)$ the set of solutions of the planning problem X returned by Ω; $\Omega(X) = \{\mathbf{failed}\}$ if X is unsolvable.

Algorithm 2 GC$[\Omega](P)$

1: **Input**: A planning problem $P = \langle F, O, I, G\rangle$
2: **Output**: A solution for P
3: Let $\Sigma = [s_0, \ldots, s_n] = reduct(P)$
4: {Compute the set of initial states }
5: {that need to be considered}
6: Compute $Sol = \Omega(P(s_0))$
7: **if** $Sol = \{\mathbf{failed}\}$ **then**
8: **return failed**
9: **end if**
10: **while** $Sol \neq \emptyset$ **do**
11: Select $\alpha_{s_0} \in Sol$ {Obtain a solution of $P(s_0)$}
12: **if** α_{s_0} is a solution of P **then**
13: **return** α_{s_0}
14: **else**
15: $\beta = completion(\alpha_{s_0}, P, \Sigma, 1)$
16: **if** β is a solution of P **then**
17: **return** β
18: **end if**
19: **end if**
20: $Sol = Sol \setminus \{\alpha_{s_0}\}$
21: **end while**
22: **return unknown**

At first sight, Algorithm 2 is fairly similar to Algorithm 1. However, they differ in three key aspects. *First,* Algorithm 2 considers only a subset of all possible initial states whenever possible (Line 3), i.e., when the `one-of`-combination is applicable for problem P. *Second,* it attempts to construct a possible plan for other initial states in the reduced set of initial states (Line 15). *Third,* it repeatedly requests for a possible plan from the same initial states. Algorithm 2 differs from the algorithm in FRAG-PLAN in its key step, Line 15, where a new possible plan is constructed.

Intuitively, the algorithm explores the solution space of sub-problem $P(s_0)$ of P; each plan α_{s_0} of $P(s_0)$ is considered (Line 11) and an attempt is made to "repair" it, so that it becomes a plan for other subproblems $P(s_i)$, $s_i \in reduct(P)$. It returns **failed** if Ω indicates that $P(s_0)$ is not solvable. The procedure $completion(\alpha, P, \Sigma, Index)$, executed in Line 15, is the actual algorithm that encodes the process of completing the action sequence α into a potential solution of P, as described in Examples 1-2. If the completion fails, another solution for $P(s_0)$ is considered and the process repeated. The algorithm returns **unknown** if it cannot generate a solution. For this reason, the algorithm is in general incomplete since Ω might not be able to generate all solutions of a sub-problem.

Algorithm 3 $completion(\alpha, P, \Sigma, Index)$

1: **Input**: α–a solution of $P(s_0)$
$P = \langle F, O, I, G\rangle$–conformant planning problem
$\Sigma = [s_0, \ldots, s_n]$–list of initial states of P
$Index$–the index for starting the completion
2: **Output**: A possible solution for P
3: Let $\alpha_{s_{Index-1}} = \alpha$
4: Initialize $\alpha_{s_i} = []$ for $i = Index, \ldots, n$
5: **for** $i = Index$ to n **do**
6: {completion of α for the states $s_{Index}, \ldots, s_n$}
7: $s = s_i$ {current state}
8: Assume that $\alpha_{s_{i-1}} = [a_0, \ldots, a_{last}]$
9: **for** $j = 0$ to $j = last$ **do**
10: $tGoal = pre(a_j)$ {create a temporary goal}
11: **if** $tGoal = \emptyset$ **then**
12: $E = \{a_j : \psi \rightarrow l \mid a_j : \psi \rightarrow l \in O$ is applicable in $\widehat{succ}([a_0, \ldots, a_{j-1}], s_{i-1})\}$
13: $tGoal = tGoal \cup (\bigcup_{[a_j:\psi\rightarrow l]\in E} \psi)$
14: **end if**
15: Select $\gamma \in \Omega(\langle F, O, s, tGoal\rangle)$
16: **if** $\gamma = \mathbf{failed}$ **then**
17: **return failed**
18: **end if**
19: $\alpha_{s_i} = \alpha_{s_i} \circ \gamma \circ [a_j]$ {update current plan}
20: $s = succ(a_j, \widehat{succ}(\gamma, s))$ {update current state}
21: **end for**
22: Select $\delta \in \Omega(\langle F, O, s, G\rangle)$ {select a solution}
23: **if** $\delta = \mathbf{failed}$ **then**
24: **return failed**
25: **end if**
26: $\alpha_{s_i} = \alpha_{s_i} \circ \delta$ {update current plan}
27: **end for**
28: **return** α_{s_n}

The procedure $completion(\alpha, P, \Sigma, Index)$ is described in Algorithm 3. Its parameters are the conformant planning problem P, whose initial belief state, represented as a list, is $\Sigma = [s_0, \ldots, s_n]$, a solution α of $P(s_0)$, and an index used to guide the start of the completion process. The procedure attempts to create solutions α_{s_i} for $P(s_i)$, $i = 1, \ldots, n$.

For each iteration of the **for-loop** in Lines 5–27, the algorithm constructs a solution of the sub-problem $P(s_i)$ from the solution $\alpha_{s_{i-1}}$ of $P(s_{i-1})$, by inserting actions into $\alpha_{s_{i-1}}$. To achieve this, the algorithm starts with the state s_i (Line 7) and an empty plan and considers each action a in $\alpha_{s_{i-1}}$ (Lines 9–21):

- **Task 1**: inserts a sequence of actions before a (loop 9–21), so that *(1)* a is executable and *(2)* the execution of a maintains the effects of a if $pre(a) = \emptyset$, i.e., a is always executable. To do this, the algorithm creates a goal $tGoal$

and modifies it accordingly (line 10) and (lines 11–14). There are several reasons behind this task. The step ensures that the preconditions are achieved to guarantee that a plan remains such. Furthermore, most classical planners are fairly sophisticated and do not generate redundant actions. As such, the existence of a in $\alpha_{s_{i-1}}$ is (almost always) necessary for achieving the goal in s_{i-1}; thus, its effects need to be maintained for the final plan to be a conformant plan. Finally, planning from the current state s to achieve $tGoal$ is expected to be significantly simpler compared to planning to achieve the final goal from s.
Observe that the algorithm needs to choose which effects of a should be maintained (line 9–21). Two obvious choices are: **(1)** the *greedy choice* requires $tGoal$ to be the union of $pre(a)$ and the precondition of all applicable effects of a; **(2)** the *ignorant choice* considers $tGoal = pre(a)$. In GC[Ω], we use a compromise among these two options. We discuss this in the next section.

- **Task 2**: makes sure that the final sequence of actions α_{s_i} achieves the goal of $P(s_i)$ (Line 22)—and this may require adding extra actions at the end of α_{s_i}.

Observe that each α_{s_i} is a solution of $P(s_i)$ but it is possible that none of the α_{s_i} is a solution of P. This due to the fact that the insertion of actions into $\alpha_{s_{i-1}}$ does not guarantee that α_{s_i} remains a solution of $P(s_{i-1})$. This is also the reason why Algorithm 2 includes the test in line 16. This aspect is discussed in the summary section.

We will next discuss some properties of Algorithm 2. The test in Lines 12 and 16 prove the soundness of GC[Ω].

Proposition 1. *If Ω is a sound classical planner, then* GC[Ω] *is sound.*

Under the assumption that Ω is complete (i.e., it can return some plans and return **failed** if the problem is unsolvable), we have the following proposition that indicates a weak form of completeness of Algorithm 2.

Proposition 2. *If Ω is complete and* GC[Ω](P) *returns* **failed** *then P does not have a solution.*

Proof. GC[Ω](P) returns **failed** iff $\Omega(P(s_0)) = \{\mathbf{failed}\}$ (Lines 6-8); this implies that P does not have a solution, because of Observation 1 and the completeness of Ω. □

In general, Algorithm 2 can be incomplete. There are two sources of incompleteness for the algorithm: *(a)* $\Omega(P(s_0))$ may generate only solutions for $P(s_0)$ that can never be completed to a solution of P by the completion algorithm; and *(b)* Algorithm 3 considers only one possible way of completion of a solution of $P(s_0)$. The next two examples illustrate case *(a)* and case *(b)*, respectively.

Example 3. Consider the problem $P = \langle\{p, q, r\}, O, I, \{r\}\rangle$ where O contains $a : \top \rightarrow p, r$; $b : q \rightarrow \neg q$; $c : \top \rightarrow p$; and $d : \neg q, p \rightarrow r$ and $pre(a) = q$; $pre(b) = pre(c) = pre(d) = \top$, $I = \{\texttt{one-of}(q, \neg q), \neg p, \neg r\}$. We have that $ext(I) = \{s_0, s_1\}$ with $s_0 = \{q, \neg p, \neg r\}$ and $s_1 = \{\neg q, \neg p, \neg r\}$.

Suppose that $\Omega(P(s_0))$ generates only one solution, $\alpha = [a]$, for $P(s_0)$. Since $pre(a) = q$ and there is no action that achieves q, the completion will fail (Lines 16–18); therefore, α cannot be completed into a solution of P; GC[Ω](P) will not be able to find the solution $[b, c, d]$ of P. □

Example 4. Consider the problem $P = \langle\{0, 1, 2\}, O, I, \{1\}\rangle$ where O contains $l : i \rightarrow i + 1$ $(i < 2)$ and $r : i \rightarrow i - 1$ $(i > 0)$, $pre(l) = pre(r) = \top$, and $I = \{\texttt{one-of}(0, 1, 2)\}$.

We have that $ext(I) = \{s_0, s_1, s_2\}$ with $s_0 = \{0, \neg 1, \neg 2\}$, $s_1 = \{\neg 0, 1, \neg 2\}$, and $s_2 = \{\neg 0, \neg 1, 2\}$.

Consider the solution $\alpha = [l]$ for $P(s_0)$. Suppose that the completion algorithm uses the ignorant choice; it will return $\beta = [l, r]$ as its answer. β is not a solution of P. In particular, for any plan α of $P(s_0)$, $completion(\alpha, P, [s_0, s_1, s_2], 1)$ will return an action sequence that is not a solution of P if ignorant choice is used. □

The above examples show that, in general, GC[Ω] is incomplete. Let us identify properties of Ω that can guarantee the completeness of GC[Ω]. We say that Ω is *strongly complete* if, for every integer k and problem P, Ω is capable of generating all solutions of P whose length is bounded by k. Observe that strongly complete planners exist, with performance comparable to state-of-the-art planners in many problems and in different settings. Answer set programming based planners (see, e.g., (Eiter et al. 2003; Lifschitz 2002; Tu et al. 2011)) are an example of strongly complete planners. These planners are similar to SAT-planners in that a planning problem is translated into a logical theory (logic program) whose models (answer sets) correspond to solutions of a given length. Answer set programming based planners use answer set solvers to compute solutions and these solvers do generate all answer sets of a logic program.

We assume that calls made to Ω include an extra parameter indicating the bounded length of solutions. Similarly, we assume that GC[Ω] includes an additional argument aimed at providing a bound to the length of the possible solutions. Algorithm 2 can be modified with a loop around Lines 6–21 to explore solutions of $P(s_0)$ of increasing length, up to the given bound.[3] Then, we can easily have the following result.

Proposition 3. *If P has a solution of length k and Ω is strongly complete, then* GC[Ω]*(P,k) finds a solution of P.*

Proof. Consider a solvable problem P. Let α be a solution of P of length k. Thanks to Observation 1, α will also be a solution of $P(s_0)$. As such, $\alpha \in \Omega(P(s_0), k)$. Since GC[$\Omega$]$(P, k)$ will consider every possible plan in $\Omega(P(s_0), k)$, it is clear that it will return a solution of P. □

It is easy to see that, with a minor change, GC[Ω] could be made to return a set of solutions and becomes strongly complete whenever Ω is strongly complete (though its performance could be impacted).

Implementation and Evaluation

We implemented Algorithm 2, setting Ω equal to the classical planner *LAMA*.[4] We refer to this system as GC[LAMA].

[3] For brevity, we omit the details of the modified algorithm.

[4] www.informatik.uni-freiburg.de/~srichter/software/lama.tar.gz.

We note that the choice of LAMA is justified by several reasons. LAMA 2008's performance is exceptional—it was the winner of the Sequential Satisficing track of IPC 2008, and its successor, LAMA 2011, was the winner of the same track in IPC 2011. Furthermore, LAMA's object-oriented implementation makes it easy to instantiate new planning modules with different initial states and goals. For efficiency reason, we do not compute all solutions of a problem (Algorithm 2, Line 6). Instead, we request possible plans one by one.

The source code of GC[LAMA] is written in Python and C++. In order to test GC[LAMA] on a wide range of conformant planning problems and achieve good performance, we have made the following modifications to LAMA:

- The parser has been modified to consider various types of actions, unsupported by LAMA (e.g., the action *right* in the `cube` domain)—i.e., actions with several conditional effects related to one variable. This problem could be dealt with by modifying the encodings of the domains. For the sake of fairness, we opted to maintain the original encoding of the domains and modify the planner instead.
- The parser has been modified to enable the computation of the initial belief state of conformant planning problems and to identify the set of possible initial states that need to be considered according to the `one-of`-combination of (Tran et al. 2009).
- Algorithms 2 and 3 have been integrated in LAMA. To generate multiple solutions of a problem, we disable the A^* feature of LAMA by keeping the open list (queue of unexplored nodes) after the first solution is found, and continuing the search for the next solution if needed.

We observe that Lines 12–13 in Algorithm 3 select a set of applicable effects of an action and attempt to maintain these effects. This represents an hybrid selection between the two possible choices discussed in the previous section (i.e., greedy vs. ignorant). This design is the results of our empirical analysis over a large pool of benchmarks. Using the *ignorant choice*, we can solve problems in all but two domains (`cube` and `sqr-center`). On the other hand, using the *greedy choice*, we can solve problems from all domains but efficiency and scalability of the planner are not as good as with the ignorant choice. This is due to the fact that the call in Line 15 of Algorithm 3 often fails because not all of the effects of a_j can be maintained. To this end, we choose to maintain only those applicable effects whose preconditions are *simple*, where an effect $a : \psi \rightarrow l$ is simple if $|\psi| \leq 1$.

Furthermore, in order to mitigate the fact that the insertion of actions into $\alpha_{s_{i-1}}$ does not guarantee that α_{s_i} remains a solution of $P(s_{i-1})$, we introduce an additional call to the *completion* procedure in Algorithm 2—i.e., we add the following lines after line 18:

18' : **if** $(\beta \neq \textbf{failed})$ **then**
18" : $\quad \beta = completion(\beta, P, \Sigma, 0)$
18"': **endif**

This second cycle attempts to refine β (a solution for $P(s_n)$) to become a solution of the other sub-problems. Apart from these modifications, the implementation follows the previously described design without any additional changes (e.g., without any modifications concerning completeness).

We compare GC[LAMA] to other state-of-the-art planners—i.e., CPA (Tran et al. 2009), DNF (To, Pontelli, and Son 2009), and `t0` (Palacios and Geffner 2009)—on problems from the literature and from planning competitions. Previously developed conformant planners, such as CFF (Brafman and Hoffmann 2004), POND (Bryce, Kambhampati, and Smith 2006), KACMBP (Cimatti, Roveri, and Bertoli 2004), and C-PLAN (Castellini, Giunchiglia, and Tacchella 2001) have been extensively investigated and do not outperform the above planners; thus, we do not consider them in our study. The authors of DNF have two new planners, but their performance is similar to DNF—thus, we include only DNF in our experiment. Similarly, the system `t1` (Albore, Ramirez, and Geffner 2011) has been shown to have a performance comparable to the best of `t0` or DNF. Finally, we do not include in the experiment the planner CPLS in (Nguyen et al. 2011) since CPLS uses a preliminary version of GC[LAMA], called CPCL, in its implementation. In this sense, a comparison between GC[LAMA] and CPLS is already presented in (Nguyen et al. 2011) and we can say that GC[LAMA] is comparable with CPLS in most benchmarks. There exists no extensive comparison between FRAG-PLAN and other planners. We were unable to obtain a running version of FRAG-PLAN and, thus, could not conduct experiments with it. Judging from the original paper on FRAG-PLAN, we believe that its performance is comparable to C-PLAN and will not be able to outperform current state-of-the-art planners. The experiments are conducted on an Intel Core2 Quad CPU Q9400 2.66GHz machine, with 4Gb memory, and a run-time time-out of 30 minutes.

The benchmark set contains 777 instances of 20 domains from recent IPCs (2006 and 2008), from the distributions of CFF and `t0`, and from the literature. In particular, 5 domains are from IPC 2006 (Bonet and Givan 2006): `coins` (30 instances), `comm` (25), `blw` (4), `uts` (30), `sortnet` (15). IPC 2008 (Bryce and Buffet 2008) introduces three new domains: `uts-cycle` (15), `raos-keys` (30), and `forest` (9). The distribution of CFF (Brafman and Hoffmann 2004) contains classical domains such as `bomb` (16 instances), `ring` (100), `safe` (10), `sqr-center` (31), `cube` (35). The distribution of `t0` (Palacios and Geffner 2009), contains a number of interesting conformant planning domains: `dispose` (90), `push` (90), `look-n-grab` (66), `1-dispose-dis` (90), and `sortnum` (30). The authors of (Bonet, Palacios, and Geffner 2009) introduced two domains `halls-A` (26), `halls-R` (16), and `markers` (4), which appear to be challenging for most conformant planners. Due to lack of space, we omit the precise description of each domain.

Summary of Experimental Results: Table 1 summarizes the results in terms of number of solved instances. In each row, the number in boldface indicates the best among the various planners. Observe that GC[LAMA] dominates all of the other planners in term of coverage, i.e., it solves more instances than any other planner in any domain. GC[LAMA] solves 93% of the 777 instances from 20 domains. CPA, `t0`, and DNF solve 46%, 50%, and 58% of the 777 instances, respectively. Observe that the performance of CPA or DNF in some domains (e.g., `comm`) relies on the *goal splitting* technique, described in (Tran et al. 2009). This technique can

Domain(#Instances)	CPA(H)	t0	DNF	GC[LAMA]
blw(4)	3	3	3	**4**
coins(30)	20	20	20	**30**
comm(25)	**25**	**25**	**25**	**25**
sortnet(15)	**15**	9	**15**	**15**
sortnum(30)	4	7	6	**25**
uts(30)	**30**	**30**	**30**	**30**
uts-cycle(30)	11	7	12	**15**
raos-keys(30)	2	2	2	**3**
forest(9)	2	8	2	**9**
bomb(16)	15	**16**	15	**16**
cube(35)	14	27	20	**35**
sqr-center(31)	19	25	25	**31**
ring(100)	5	48	5	**100**
safe(10)	**10**	**10**	**10**	**10**
dispose(90)	66	62	89	**90**
push(90)	44	33	75	**90**
look-n-grab(66)	39	15	39	**66**
1-dispose-dis(90)	9	7	24	**90**
hall-A(26)	19	20	20	**23**
hall-R(16)	10	**16**	10	**16**
marker-enc1(4)	0	0	1	**4**
Total (777)	362	390	448	**727**

Table 1: Number of solved problems

Instance	CPA(H)	t0	DNF	GC[LAMA]
blw-01	0.20/4	0.056/5	0.20/7	**0.049**/8
blw-03	20.4/205	48.51/80	307/325	**1.3**/266
blw-04	AB	AB	AB	**29.5**/1384*
coins-10	**0.03**/48	0.04/26	0.20/27	0.037/36
coins-30	AB	AB	AB	**1.0**/1107
comm-15	2.29//95	**0.092**/110	3.43/125	0.1/97
comm-25	1222/389	1.55/453	1797/501	**0.8**/294
sortnet-5	**0.02**/13	0.18/15	0.03/15	0.05/15
sortnet-15	240/74	AB	**35**/118	63.9/120
sortnum-5	AB	1.9/10	1.67/10	**0.81**/10
sortnum-20	AB	AB	AB	**12.3**/190
uts-10	14.3/89	0.88/59	2.66/66	**0.26**/58
uts-30	4.9/74	0.79/67	1.39/73	**0.17**/64
uts-cycle-03	**0.01**/3	0.14/3	0.01/3	0.04/3
uts-cycle-05	0.12/10	1.84/10	0.10/10	**0.11**/12
uts-cycle-15	AB	AB	AB	**1314**/272
raos-keys-02	0.26/32	**0.02**/21	0.09/39	0.05/38
raos-keys-03	4.21/152	0.22/66	0.80/153	**0.207**/172
raos-keys-04	AB	AB	AB	**16.78**/163
forest-03	AB	0.62/45	TO	**0.46**/167
forest-09	AB	AB	AB	**183.8**/963

Table 2: Results for IPC 2006/08 Domains (Time in *seconds*)

also be applied in GC[LAMA] and will be one of our considerations for improving GC[LAMA] in the future.

GC[LAMA] is the only planner that can solve the coins-21,..., coins-30 instances, a task that has been accomplished so far only by CNF (To, Son, and Pontelli 2010) with a special technique called *or-relaxation*. GC[LAMA] is also the only planner that can solve many instances that cannot be solved by any other planners (e.g., raos-keys-04, sortnum-09,...,sortnum-14, forest-09, marker-enc, and so on). In addition, it is also the only planner that can solve *all instances* of the challenging problems included in the distribution of t0.

Domains from IPCs: Table 2 contains the results of our experiments with domains from the IPCs 2006/2008—in terms of the time and length of the first solution reported by each planner. Boldface indicates the fastest planner. *AB* denotes an execution aborted by the planner due to"out of memory," and *TO* denotes time-out. For the large instances of these domains, the main difficulty lies in the huge numbers of objects, which lead to very large initial belief states (e.g., the coins-30 instance has, theoretically, 10^{25} initial states). This is the main reason for the AB results in the table.

GC[LAMA] performs exceptionally well, both in term of efficiency and scalability. The two hardest instances for GC[LAMA] are uts-cycle-15 and forest-9, for which GC[LAMA] takes 22 and 3 minutes, respectively, to find a solution. For all other instances, it uses about a minute. GC[LAMA] consistently outperforms other planners on large instances. In many domains, GC[LAMA] is the only planner that can solve all available instances (e.g., blw, coins, forest). It is also interesting to observe that the length of solutions generated by GC[LAMA] in these domains is somewhat mixed. In some cases, GC[LAMA] generates the shortest plan among all four planners. In other cases, it generates the longest plan.

We have used the verifier software from the IPC2008 to re-verify the output of GC[LAMA]. In some cases, GC[LAMA] uses about half a minute to find a solution but the verifier takes about half an hour to check its correctness. In an extreme case (marked with *), the verifier stops due to insufficient memory (blw-4).

Instance	CPA(H)	t0	DNF	GC[LAMA]
bomb-10-1	0.15/19	**0.01**/20	0.05/19	0.02/19
bomb-100-100	AB	6.26/200	TO	**4.27**/100
cube-39-19	TO	3.6/171	198/1023	**0.23**/171
cube-119-59	AB	AB	AB	**1.9**/531
sqr-center-32-16	14.8/928	0.95/93	7.3/340	**0.12**/94
sqr-center-120-60	AB	AB	AB	**1.03**/358
ring-20	AB	1.78/95	AB	**0.12**/72
ring-100	AB	AB	AB	**8.71**/1640
safe-10	0.04/10	0.02/10	0.029/10	**0.027**/10
safe-100	339/100	1.26/100	4.06/100	**0.93**/100

Table 3: Results for Domains inn the CFF Distribution

Domains in CFF's Distribution: Table 3 contains the results from the domains in CFF's distribution. As with the domains in the IPCs, scalability is the main issue in these domains. Again, we can see that GC[LAMA] is outstanding in all domains. It is slower than t0 only in the smaller instances of the bomb domain. GC[LAMA] is the only planner that scales well and the only planner that can solve every instance included in the distribution. To test the scalability of GC[LAMA], we also generated some new, larger instances for these domains; GC[LAMA] is able to solve them while all the other planners cannot deal with them (we omit these additional results due to lack of space). We note

that GC[LAMA] also generates the shortest plan in all but one domain in this set of problems.

Challenging Domains: Table 4 shows that GC[LAMA] is particularly effective in the domains from the grid family. In this group of domains, GC[LAMA] is exceptional, both in performance and in scalability. It is the only planner that can solve *all* instances from these domains, each in less than 2 minutes. However, the solutions generated by GC[LAMA] are often longer than those generated by other planners.

GC[LAMA] is an order of magnitude faster than other planners when there are more than 2 objects. Its performance does not seem to change when the number of objects increases, while this causes an order of magnitude performance change in CPA and `t0`.

Domains for Finite-State Controllers: The domains in this group are described in the paper (Bonet, Palacios, and Geffner 2009). They encode the problem of automatically generating finite-state controllers and represent a challenge in scalability for `t0`. Table 5 presents the results of GC[LAMA] and compares them to those obtained from `t0`. The results show that GC[LAMA] is again superior to other planners, both in performance and scalability.

Instance	CPA(H)	`t0`	DNF	GC[LAMA]
ds-8-1	15.0/741	1.67/426	47.12/150	**1.15**/706
ds-8-3	224.7/2227	133.92/761	34.42/629	**1.57**/962
ds-12-5	AB	AB	AB	**10.03**/3460
ds-12-9	AB	AB	AB	**20.34**/4612
push-8-1	17.76/465	61.85/464	51.21/163	**1.73**/655
push-8-3	AB	AB	103.8/1477	**3.44**/1105
push-12-5	AB	AB	AB	**23.29**/2429
push-12-9	AB	AB	AB	**35.10**/2761
lng-8-1-1	182.4/554	109.2/145	51.28/99	**1.72**/692
lng-12-1-4	AB	AB	AB	**95.18**/2010
lng-12-5-2	AB	AB	AB	**114.7**/2010
lng-12-5-4	AB	AB	AB	**264.7**/2010
1-ds-8-5-dis	AB	AB	AB	**2.61**/1026
1-ds-8-9-dis	AB	AB	AB	**3.81**/1026
1-ds-12-9-dis	AB	AB	AB	**83.16**/3462

Table 4: Results for Challenging Domains

Instance	CPA(H)	`t0`	DNF	GC[LAMA]
hall-A-ui-q2-1x10	39 / 77	1.09/56	0.29/58	**0.01**/46
hall-A-ui-q2-1x20	AB	7.2/116	0.7/118	**0.04**/86
hall-A-ui-q4-3x2	AB	TO	AB	**0.05**/19
hall-A-ui-q4-4x2	AB	TO	AB	**0.04**/42
hall-R-ui-q1-1x10	828/93	0.42/109	680/91	**0.01**/138
hall-R-ui-q1-2x2	AB	0.03/24	AB	**0.0**/24
hall-R-ui-q1-3x3	AB	0.12/40	AB	**0.01**/63
hall-R-ui-q1-4x4	AB	0.19/50	AB	**0.01**/110
marker-enc1-q2-5x4	AB	TO	124/25	**0.37**/25
marker-enc1-q2-7x5	AB	TO	AB	**11.1**/31
marker-enc1-q2-8x5	AB	TO	AB	**11.5**/39
marker-enc1-q2-9x5	AB	TO	AB	**0.02**/45

Table 5: Finite-State Controllers Domains

Related Work and Discussion

GC[LAMA] is closely related to C-PLAN and FRAG-PLAN. Algorithmic differences and similarities between GC[LAMA] and these planners have been discussed in detail in the previous sections. The experimental results in the last section show that GC[LAMA] is better than C-PLAN and FRAG-PLAN, both in term of coverage and scalability.

Although both GC[LAMA] and `t0` employ a classical planner in the search for conformant plans, the method used by these planners is radically different. `t0` translates the original problem into a classical planning problem, and hence, changes the problem and often increases its size. GC[LAMA] does not change the original problem. We believe that this factor is an important reason behind the limits in scalability of `t0` in several domains.

The experimental evaluation shows that GC[LAMA]'s performance is superior to other state-of-the-art planners. The simplicity of the algorithm implemented in GC[LAMA] raises the question of what are the reasons behind these performance results. We believe that there are two reasons for this. First of all, by dealing with the sub-problems *separately*, GC[LAMA] can take advantage of techniques that have been developed in conformant planning research for reducing the size of the initial belief state. Second, by attempting to achieve only the precondition of an action during the completion phase, which often needs only a few actions, GC[LAMA] can exploit the best heuristic classical planners in computing a solution.

We observe that, even though GC[LAMA] can solve a wide range of benchmarks from various sources, which seem to be difficult for other state-of-the-art conformant planners, there are still domains in which GC[LAMA] does not work well. Among them, the `adder` and `rao-keys` domains seem to be the most difficult ones. The `adder` domain is special in that the size of the initial belief state is very small, but the number of actions which can be executed in a state is very large. Furthermore, the conditional effects of the actions are much more complex than those in other domains. The `rao-keys` domain is similar to `adder`, as it contains actions with many conditional effects (we can solve only 3 out of 30 instances). We hypothesize that these two factors make this domain difficult for conformant planners.

Conclusion and Future Work

We proposed a novel approach to conformant planning using classical planners whose key ingredient is a completion algorithm, which takes a solution of a sub-problem and creates a solution for a sequence of sub-problems. We implemented the algorithm using the source code of the classical planner LAMA, and evaluated the new planner, GC[LAMA], against state-of-the-art conformant planners. GC[LAMA] outperforms other planners in both performance and scalability, indicating that the proposed approach is a strong alternative to current state-of-the-art approaches. The experimental results, especially in the new set of domains (Finite-State Controller), confirm that the proposed technique can be applied to a wide range of problems and that GC[LAMA] could be used in real-world applications.

The exceptional performance of GC[LAMA] also provides us with a set of questions that we would like to further investigate in the near future. First, we would like to investigate alternatives to the completion algorithm (e.g., identifying effects that need to be maintained). Second, we would like to construct domains that are difficult for GC[LAMA]. One observation that we made during our experiment is that, for several domains, GC[LAMA] was able to find a solution by generating only one possible plan for $P(s_0)$ (Alg. 2). This might provide hints for the construction of difficult problems for GC[LAMA] and for other conformant planners.

References

Albore, A.; Palacios, H.; and Geffner, H. 2009. A translation-based approach to contingent planning. In *IJCAI*, 1623–1628.

Albore, A.; Ramirez, M.; and Geffner, H. 2011. Effective Heuristics And Belief Tracking For Planning With Incomplete Information. In *ICAPS*.

Bonet, B., and Givan, B. 2006. Results of the conformant track of the 5th planning competition. http://www.ldc.usb.ve/~bonet/.

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In Gerevini, A.; Howe, A. E.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*. AAAI.

Brafman, R., and Hoffmann, J. 2004. Conformant planning via heuristic forward search: A new approach. In Koenig, S.; Zilberstein, S.; and Koehler, J., eds., *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*, 355–364. Whistler, Canada: Morgan Kaufmann.

Bryce, D., and Buffet, O. 2008. The unvertainty part of the 6th international planning competition.

Bryce, D.; Kambhampati, S.; and Smith, D. 2006. Planning Graph Heuristics for Belief Space Search. *Journal of Artificial Intelligence Research* 26:35–99.

Castellini, C.; Giunchiglia, E.; and Tacchella, A. 2001. Improvements to sat-based conformant planning. In *Proc. of 6th European Conference on Planning (ECP-01)*.

Cimatti, A.; Roveri, M.; and Bertoli, P. 2004. Conformant Planning via Symbolic Model Checking and Heuristic Search. *Artificial Intelligence Journal* 159:127–206.

Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2003. Answer set planning under action costs. 19:25–71.

Kurien, J.; Nayak, P. P.; and Smith, D. E. 2002. Fragment-based conformant planning. In *AIPS*, 153–162.

Kuter, U.; Nau, D. S.; Reisner, E.; and Goldman, R. P. 2008. Using classical planners to solve nondeterministic planning problems. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E. A., eds., *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia, September 14-18, 2008*, 190–197. AAAI.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence* 138(1–2):39–54.

Nguyen, H.-K.; Tran, D.-V.; Son, T. C.; and Pontelli, E. 2011. On improving conformant planners by analyzing domain-structures. In Burgard, W., and Roth, D., eds., *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. AAAI Press.

Palacios, H., and Geffner, H. 2009. Compiling Uncertainty Away in Conformant Planning Problems with Bounded Width. *Journal of Artificial Intelligence Research* 35:623–675.

Smith, D., and Weld, D. 1998. Conformant Graphplan. In *AAAI*, 889–896.

To, S. T.; Pontelli, E.; and Son, T. C. 2009. A Conformant Planner with Explicit Disjunctive Representation of Belief States. In Gerevini, A.; Howe, A. E.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*, 305–312. AAAI.

To, S. T.; Son, T. C.; and Pontelli, E. 2010. A New Approach to Conformant Planning using CNF. In *Proceedings of the 20th International Conference on Planning and Scheduling (ICAPS)*, 169–176.

Tran, D.-V.; Nguyen, H.-K.; Pontelli, E.; and Son, T. C. 2009. Improving performance of conformant planners: Static analysis of declarative planning domain specifications. In Gill, A., and Swift, T., eds., *Practical Aspects of Declarative Languages, 11th International Symposium, PADL 2009, Savannah, GA, USA, January 19-20, 2009. Proceedings*, volume 5418 of *Lecture Notes in Computer Science*, 239–253. Springer.

Tran, V.; Nguyen, K.; Son, T.; and Pontelli, E. 2012. A Conformant Planner Based on Approximation: CPA(H). Technical report, New Mexico State University, Department of Computer Science.

Tu, P. H.; Son, T. C.; Gelfond, M.; and Morales, R. 2011. Approximation of action theories and its application to conformant planning. *Artificial Intelligence Journal* 175(1):79–119.