Artificial Intelligence 231 (2016) 64–106

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Agent planning programs

Giuseppe De Giacomo [a], Alfonso Emilio Gerevini [b], Fabio Patrizi [c], Alessandro Saetti [b], Sebastian Sardina [d,*]

[a] Dipartimento di Ingegneria Informatica, Automatica e Gestionale, Sapienza Università di Roma, Italy
[b] Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Brescia, Italy
[c] KRDB Research Centre for Knowledge and Data, Faculty of Computer Science, Free University of Bozen–Bolzano, Italy
[d] School of Computer Science and IT, RMIT University, Melbourne, Australia

ARTICLE INFO

Article history:
Received 19 July 2014
Received in revised form 27 June 2015
Accepted 10 October 2015
Available online 3 November 2015

Keywords:
Agent-oriented programming
Automated planning
Reasoning about action and change
Synthesis of reactive systems

ABSTRACT

This work proposes a novel high-level paradigm, *agent planning programs*, for modeling agents behavior, which suitably mixes automated planning with agent-oriented programming. Agent planning programs are finite-state programs, possibly containing loops, whose atomic instructions consist of a guard, a maintenance goal, and an achievement goal, which act as precondition-invariance-postcondition assertions in program specification. Such programs are to be executed in possibly nondeterministic planning domains and their execution requires generating plans that meet the goals specified in the atomic instructions, while respecting the program control flow. In this paper, we define the problem of automatically synthesizing the required plans to execute an agent planning program, propose a solution technique based on model checking of two-player game structures, and use it to characterize the worst-case computational complexity of the problem as EXPTIME-complete. Then, we consider the case of deterministic domains and propose a different technique to solve agent planning programs, which is based on iteratively solving classical planning problems and on exploiting goal preferences and plan adaptation methods. Finally, we study the effectiveness of this approach for deterministic domains through an experimental analysis on well-known planning domains.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

This work proposes a novel paradigm for programming intelligent agents and controllers in a task-oriented way that mixes automated planning with agent-oriented programming w.r.t. behavior specification.[1] Generally speaking, we envision the designer providing a high-level model of the "space of deliberation" of the agent—called an *agent planning program*—that is meant to be "realized" into an executable program via automatic synthesis. Agent planning programs are finite-state programs, possibly containing loops, whose atomic instructions are classical *precondition-invariance-postcondition* declarative assertions. Such programs are to be executed in possibly nondeterministic domains. A "realization" of such programs in the domain of concern amounts, basically, to a collection of inter-related plans that meet the assertions in the atomic

* Corresponding author.
E-mail addresses: degiacomo@dis.uniroma1.it (G. De Giacomo), alfonso.gerevini@ing.unibs.it (A.E. Gerevini), patrizi@dis.uniroma1.it (F. Patrizi), alessandro.saetti@ing.unibs.it (A. Saetti), sebastian.sardina@rmit.edu.au (S. Sardina).

[1] This work integrates and extends [34,44].

http://dx.doi.org/10.1016/j.artint.2015.10.001
0004-3702/© 2015 Elsevier B.V. All rights reserved.

instructions while respecting the program control flow in its totality (that is, a plan for an assertion should not preclude the realization of potential future instructions).

Technically, the dynamics of the world is described with a *planning domain* and a given initial state, as usually done in domain-independent planning [46] and reasoning about action [88]. On top of such (rooted) domain, an agent planning program is modeled as a finite transition system, typically including loops, in which states represent *choice points* and transitions specify possible objectives that the agent may decide to pursue. Such transitions constitute the high-level actions available to the agent, and are characterized by: *(i)* a *guard*, which poses executability preconditions; *(ii)* a *maintenance goal*, which specifies invariants that are guaranteed to hold for the course of actions to execute; and *(iii)* an *achievement goal*, which specifies the postcondition of the transition. In other words, those triples are a direct counterpart of the classical triple precondition-invariant-postcondition, used in program specification [38,42,55] and nowadays in "design-by-contract" or "code-by contract" development [74].

Intuitively, agent planning programs are meant to work as follows: at any point in time, based on the current state of the domain and that of the agent planning program, the agent decides, autonomously, which enabled program transition to pursue. A (synthesized) plan satisfying the assertions in the chosen transition is then executed, thus moving the domain and the program to their next states, from which a new transition will be "requested" by the agent, a new plan executed again, and so on. The agent planning program is said to be *realized* if an adequate plan can always be associated to the execution of transitions, according to the planning program control flow. Note that, although the planning program is a finite transition system, it may generate, due to loops, an infinite computation tree; in principle, one needs to synthesize plans for each of the infinitely many transitions of such a tree. A key point is that, in synthesizing a plan for a particular transition, one needs to take into account that the resulting state of the domain must not only satisfy the corresponding achievement goal assertion, but also must allow for the existence of plans for each possible next transition, and this must hold again after such plans, and so on.

By combining declarative and procedural approaches to behavior specification, together with automatic synthesis techniques, the agent planning program approach has the potential to provide convenient and powerful specification of behavior in complex scenarios. For example, they can be used to encode knowledge-intensive business processes (processes reflecting "preferred work practices" whose execution is controlled by contingent agent decision making, coupled with contextual data and knowledge production) [27,99],[2] or even non-linear *storylines* behind characters' actions in a video game [18,85]. Planning programs can also be a convenient model of an embedded system for a smart house controller [52] or a Holonic manufacturing system [50], in which the actual concrete manner of doing things may vary from setting to setting. Last, but not least, they can be used to specify the requirements for a web-service [72]. The assumption is that the agent (e.g., a human interacting with a business process, embedded system, or a game narrative generator) issues, step-by-step, *goal requests within the given space of deliberation*, which are to be fulfilled by appropriate plans computed by the solver.

In this paper, we study the above realizability (and associated synthesis) problem and provide the following contributions:

- A formal definition of the problem of realizing an agent planning program and its solution.
- A correct and terminating technique for synthesizing realizations, which resorts to automated synthesis for certain kinds of Linear-time Temporal Logic (LTL) specifications based on model checking game structures [80,82]. Interestingly, such a technique can be readily implemented using available tools for synthesis based on model checking of game structures, such as the well-known TLV [84], JTLV [83], or the more recent NuGaT [17], which we use for our experiments.
- A worst-case complexity characterization of the problem as EXPTIME-complete, where we use the above technique for establishing membership in EXPTIME. The EXPTIME-hardness comes from the EXPTIME-completeness of conditional planning with full observability in nondeterministic domains [90], which is a special case of our problem: a planning program formed by a single transition labelled with an achievement goal.

The output obtained from our general realization technique is akin to a sort of sophisticated form of universal plan [92], which is obviously a costly solution [48]. To deal with this, in the second part of the paper, we look for alternative computational approaches based on exploiting state-of-the-art classical planning systems. In particular, we focus on the case of deterministic underlying domains, widely studied in automated planning, for which classical planning systems have shown excellent performance. The contributions for this case are the following:

- We show that the worst-case complexity of the problem remains EXPTIME-complete even for deterministic domains. In particular, for membership we can still use the general algorithm, while for the hardness we show a reduction from the service composition problem [76].
- We devise a technique for realizing planning programs that is based on classical planning tools, which involves iteratively *constructing* and *synchronizing* a set of plans. Importantly, the technique makes use of *goal preferences* and *plan adaptation* to considerably speed up plan synthesis and synchronization when realizing looping transitions.

[2] In particular, in [27], an early version of agent planning programs was used for expressing behavioral routines for people with special needs in dedicated smart homes.

- We develop and perform a thorough set of experiments to test our planning-based approach using benchmarks from planning competitions. In the experiments, we drop all transition guards (i.e., we set them to true). Note that realizing planning programs without guards does not represent a simplification in experimenting our algorithm, since it forces the algorithm to realize all outgoing transitions, even those that guards would disable. We consider both maintenance and achievement goals stated as conjunctions.
- We demonstrate, via experimental analysis, that our planning-based approach excels in domains in which the backtracking due to planning failures during the transitions realization is limited. In particular, this is the case for planning domains without deadends, where the failures are due to the given limits of the computational resources (CPU time or memory consumption limits), or to the incompleteness of the planner used. As expected, though, as the need for backtracking increases, the performance degrade, and the high worst-case complexity shows up.

Such experimental evaluation indicates that our iterated planning technique constitutes an effective baseline for handling agent planning programs in deterministic domains.

As mentioned, agent planning programs advocate a novel paradigm for "programming" complex task-oriented behaviors, by suitably mixing key ingredients of automated planning [46] and agent-oriented programming [66,67,87,97]. From the former, they inherit the ability to specify behaviors in a *declarative* manner, thus providing an abstract and powerful mechanism that caters for *flexible* behaviors from first principle, that is, without the need to specify detailed procedural information. As already accepted in the literature, declarative goals provide several other advantages, including decoupling plan execution and goal achievement, facilitating goal dynamics and plan failure handling, enabling reasoning about goal and plan interaction, and enhancing goal and plan communication [101]. From the latter, in turn, planning programs draw the ability to specify useful available "*know-how*" information, albeit at a high-level of abstraction. By doing so, it is possible to better focus on the relevant decisions—the "space of deliberation." Concretely, this is achieved by encoding the temporal/procedural relations among the declarative goals of concern: planning programs restrict the options that will be available after the (current) goal is brought about.[3]

We note that research on integrating these two approaches has a certain tradition in AI. For example, [106] advocates the use of a high-level model for describing the behaviors of interest in embedded systems, which then need to be "compiled" on-the-fly by a suitable solver into a low-level executable code for a given plant. In [93] planning for temporal goals consisting of a mixture of declarative and procedural assertions are considered. Mixing planning and programs is one of the original motivations behind the Golog family of high-level programming languages [5,29,68], possibly the most prominent programming language proposals in reasoning about action. There has also been substantial effort in bringing deliberative planning into standard BDI-style agent architectures, e.g., [39,91,103]. In fact, the necessity of studying more systematically planning in combination with acting and programming has been recently thoroughly advocated in [46]. Our proposal of agent planning programs goes exactly in this direction, thus contributing to both agent programming and planning research areas.

The rest of the paper is structured as follows. In Section 2, we formally define agent planning programs and the corresponding realization problem. In Section 3, we ground such notions on a full fledged example. Then, in Section 4, we look into the general solution for realizing agent planning programs by resorting to LTL synthesis via model checking of two-player game structures. We show that our solution is sound and complete, and characterize the worst-case computational complexity as EXPTIME-complete. In Section 5, we focus on the case where the domain is deterministic. We prove EXPTIME-completeness also in this case, and we provide our planning based technique and analyze it from the point of view of its correctness and optimizability. After that, in Section 6, we report on a set of experiments that provides evidence of the effectiveness in practice of such technique. Finally, we discuss related work and draw conclusions in Sections 7 and 8, respectively. For the sake of readability, a concrete encoding of planning programs in SMV (the input language of TLV, JTLV and NuGaT), as well as detailed proofs and additional experimental results, are given in appendices.

2. Agent planning programs

Agent Planning Programs are high-level representations of the behavior of agents acting in a domain. Essentially, they are transition systems, with states representing decision points, and transitions labelled by triples consisting of a guard, a maintenance goal and an achievement goal, over the domain. For instance, an agent planning program for a researcher's everyday-life routine is depicted in Fig. 1b, under which the agent (i.e., the researcher) can decide to move between home (state v_0), work (state v_1), and the pub (state v_2). So, at planning program state v_1, the researcher may decide to go back home (transition to state v_0) or instead go out to the pub (transition to state v_1). Once at the pub, the agent can only return to her home, where the routine iterates again for the next day.

Informally, in order for an agent planning program to be executable, each labeling goal requires a plan to bring it about. Importantly, those plans ought to be "synchronized" so that the final world state generated by each plan is a suitable initial

[3] We point out that, from an agency perspective, this work focuses only on the *means-end analysis* aspect of agent practical reasoning [15]. There are many other important aspects of agency and agent programming that are not addressed by agent planning programs, including deliberation of goals, intentions and commitments, etc.

state for the subsequent plans associated with the next goals. When this is the case, the planning program is *realized*. In general, however, computing a realization does not simply amount to matching program transitions with appropriate plans. The fact is that, as plans are executed, both the state of the planning program and that of the underlying domain evolve and, in general, the planning program may reach the same state in different domain states, so that there is no guarantee that a single plan would work in all such domain states. Thus, a more sophisticated solution concept is required.

Our framework consists of two basic components: *(i)* a *planning domain*, formalizing the environment that the agent acts in; and *(ii)* an *agent planning program*, providing a high-level representation of the agent's space of deliberation.

Definition 1 *(Planning domain).* A *planning domain* is a tuple $\mathcal{D} = \langle P, A, \tau \rangle$, where:

- P is a finite set of *domain propositions*; a *state* of the domain is a subset in 2^P;
- A is the finite set of *domain actions*; and
- $\tau \subseteq 2^P \times A \times 2^P$ is the *transition relation*; we freely interchange notations $\langle s, a, s' \rangle \in \tau$ and $s \xrightarrow{a} s'$ in $\mathcal{D}$. □

We say that an action a is *executable* in a state s, if there exists some state s' such that $s \xrightarrow{a} s'$ in $\mathcal{D}$. Notice that, in general, planning domains are nondeterministic, as their evolution is modeled by a transition relation. We next define what it means for a plan to achieve a goal in a planning domain $\mathcal{D}$. A *$\mathcal{D}$-history* h is a finite sequence $s^0 \xrightarrow{a^1} s^1 \cdots s^{\ell-1} \xrightarrow{a^\ell} s^\ell$, such that: *(i)* for each $i \in \{0, \ldots, \ell\}$, $s^i \in 2^P$; and *(ii)* for each $i \in \{0, \ldots, \ell - 1\}$, $s^i \xrightarrow{a^{i+1}} s^{i+1}$ in $\mathcal{D}$. Intuitively, $\mathcal{D}$-histories capture the possible evolutions of $\mathcal{D}$ from a state s^0. The set of all possible $\mathcal{D}$-histories is denoted by $\mathcal{H}$. Given a sequence $\eta = s^0 \xrightarrow{a^1} s^1 \cdots s^{\ell-1} \xrightarrow{a^\ell} s^\ell$, we define the *length of η*, denoted $|\eta|$, as $|\eta| = \ell + 1$, if η is finite (e.g., if it is a history), and $|\eta| = \infty$, otherwise. Moreover, for $0 < k < |\eta| + 1$, we denote the k-length finite prefix of η as $\eta|_k = s^0 \xrightarrow{a^1} \cdots \xrightarrow{a^{k-1}} s^{k-1}$ and its last state as $end[\eta] = s^\ell$.

Given a planning domain $\mathcal{D}$, a *history-based plan* (H-plan) for $\mathcal{D}$ is a partial function $\pi : \mathcal{H} \mapsto A$ such that, given a $\mathcal{D}$-history h, if π is defined on h, it returns an action $a = \pi(h)$ executable in $end[h]$. Intuitively, H-plans can be seen as "non-Markovian policies", i.e., functions that prescribe the action to execute next, given the current history (as opposed to the more commonly used "Markovian' policies", which prescribe actions based on the current state). A *trajectory* of an H-plan π (over $\mathcal{D}$), also called a π-trajectory, from a state $s^0 \in 2^P$, is a sequence $\eta = s^0 \xrightarrow{a^1} s^1 \xrightarrow{a^2} \cdots$ such that: *(i)* for all $0 < k < |\eta| + 1$, $\eta|_k$ is a $\mathcal{D}$-history; and *(ii)* for all $0 < k < |\eta|$, $a^k = \pi(\eta|_k)$. Observe that trajectories of H-plans can be either finite or infinite. A trajectory η is said to be *complete* (w.r.t. π) if it is infinite or such that $\pi(\eta)$ is undefined (thus η is finite and cannot be extended further, through π). An H-plan is said to be a *history-based terminating plan* (HT-plan, for a domain $\mathcal{D}$) if all of its complete trajectories are finite. Obviously, HT-plans induce only finite trajectories, which are in fact $\mathcal{D}$-histories. The set of all HT-plans over $\mathcal{D}$ is denoted by $HT_{\mathcal{D}}$.

A $\mathcal{D}$-history $h = s^0 \xrightarrow{a_1} s^1 \cdots s^{\ell-1} \xrightarrow{a_\ell} s^\ell$ *achieves* a goal ϕ, i.e., a propositional formula over the propositions of $\mathcal{D}$, if $s^\ell \models \phi$, where satisfaction is defined as usual in propositional logic. Similarly, h *maintains* a goal ψ if $s^i \models \psi$, for every $i \in \{0, \ldots, \ell - 1\}$. Observe that maintaining a goal ψ requires the goal to remain true up to the second last state $s^{\ell-1}$ of the history, while the goal ψ is allowed to become false in the *last* state s^ℓ (to make ψ remain true also in the last state we may simply require it to be not only maintained but also achieved). Such notions can be extended to HT-plans, as follows. We say that an HT-plan π *achieves* a goal ϕ from a state s, if all of its complete trajectories from s (which are histories) do so; also, we say that π *maintains* a goal ψ from s if all of its (complete or not) trajectories from s do so.

Next, we formally define agent planning programs as a *network*, constituted by the control flow of the program, of *declarative goal* assertions, the atomic instructions. Each of these instructions consist of a (potential) agent request, under a certain guard (i.e., precondition), to achieve a given achievement goal (i.e., postcondition) while maintaining certain condition (i.e., invariance).

Definition 2 *(Agent planning program).* An *agent planning program* is a tuple $\mathcal{P} = \langle P, V, v_0, \delta \rangle$, where:

- P is a finite set of *domain propositions*;
- V is the finite set of *program states*;
- $v_0 \in V$ is the *program initial state* of $\mathcal{P}$; and
- $\delta \subseteq V \times \Phi(P) \times \Phi(P) \times \Phi(P) \times V$ is the *transition relation* of $\mathcal{P}$, where $\Phi(P)$ stands for the set of all boolean formulas built from the set of propositions P. A transition $\langle v, \gamma, \psi, \phi, v' \rangle \in \delta$—also $\langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle \in \delta$ or $v \xrightarrow{\gamma:\psi,\phi} v'$ in $\mathcal{P}$ for legibility—is used to denote that whenever the *guard* γ holds (in the domain), the agent planning program $\mathcal{P}$ may legally move from state v to state v' by "*achieving ϕ while maintaining ψ*." □

The idea is that when the planning program and the domain are in states v and s (initially v_0 and s_0), respectively, the agent is allowed to pursue any *enabled* (i.e., whose guard holds true in s) planning program transition $v \xrightarrow{\gamma:\psi,\phi} v'$

in $\mathcal{P}$. However, being declarative assertions, such transition are *not* directly executable and actual *realizations* are required for them. A realization, then, must provide a concrete HT-plan π that brings about the achievement goal ϕ while guaranteeing maintenance of ψ and, furthermore, be compatible with further realizations for subsequent transitions (i.e., atomic instructions) of the planning program. The latter requirement is central to the approach, as the choice points in the planning program are resolved by decisions made by the agent *only at runtime*. It should be noted that only in special cases we can realize planning programs by simply annotating transitions with plans. In general, the annotation should be done on the (infinite) computation tree generated by the planning program. Indeed, a transition in the planning program may be pursued (i.e., requested by the agent) at different moments in time, from different states of the domain, and so different plans may be required.

With this intuition at hand, we are now prepared to formalize the notion of planning program realization, thus providing semantics to agent planning programs. We base such notion on a suitable variant of the formal notion of *simulation* [75], under which, loosely speaking, transitions are matched by plans, rather than by single actions.

Definition 3 *(Plan-based simulation).* Let $\mathcal{D} = \langle P, A, \tau \rangle$ be a planning domain and $\mathcal{P} = \langle P, V, v_0, \delta \rangle$ an agent planning program. A *plan-based simulation relation*, or *PLAN-simulation relation*, is a relation $R \subseteq V \times 2^P$ such that $\langle v, s \rangle \in R$ implies that, for every transition $v \xrightarrow{\gamma:\psi,\phi} v'$ in $\mathcal{P}$ such that $s \models \gamma$, there exists an HT-plan π such that:

1. π *achieves* ϕ and *maintains* ψ from s (in which case, plan π is said to <u>realize</u> the transition $v \xrightarrow{\gamma:\psi,\phi} v'$); and
2. for all complete trajectories h of plan π from domain state s, it is the case that $\langle v', end[h] \rangle \in R$ (in which case plan π is said to <u>preserve</u> R from $\langle v, s \rangle$ for $v \xrightarrow{\gamma:\psi,\phi} v'$).

A $\mathcal{P}$-state $v \in V$ is said to be *PLAN-simulated* by a $\mathcal{D}$-state $s \in 2^P$, denoted $v \preceq_{PLAN} s$, if there exists a *PLAN*-simulation relation R such that $\langle v, s \rangle \in R$. □

As for standard simulation, relation $\preceq_{PLAN}$ is a *PLAN*-simulation relation itself and, in particular, the *largest* one (with respect to set inclusion)—it can be obtained by taking the union of all *PLAN*-simulation relations.

Definition 4 *(Agent planning program realization).* A <u>realization</u> of an agent planning program $\mathcal{P}$ in planning domain $\mathcal{D}$ from an initial $\mathcal{D}$-state $s_0 \in 2^P$ is a partial function $\Omega : 2^P \times \delta \mapsto HT_{\mathcal{D}}$ such that for some *PLAN*-simulation relation R, it is the case that:

- $\langle v_0, s_0 \rangle \in R$; and
- for all pairs $\langle v, s \rangle \in R$ and all transitions $d = \langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle$ in $\mathcal{P}$ such that $s \models \gamma$, an HT-plan $\Omega(s, d)$ is defined, realizes transition d, and preserves R from $\langle v, s \rangle$ for d. □

That is, a realization Ω is a function that given a domain state s and a transition request $v \xrightarrow{\gamma:\psi,\phi} v'$, whose guard is satisfied in s, outputs an HT-plan π that achieves ϕ while maintaining ψ from s and guarantees that all potential future requests from v' after π's execution can also be fulfilled by plans prescribed by the realization function Ω.

Our first result states if the initial state of $\mathcal{P}$ is *PLAN*-simulated by the initial state of $\mathcal{D}$ then it is guaranteed that a realization exists (and obviously viceversa).

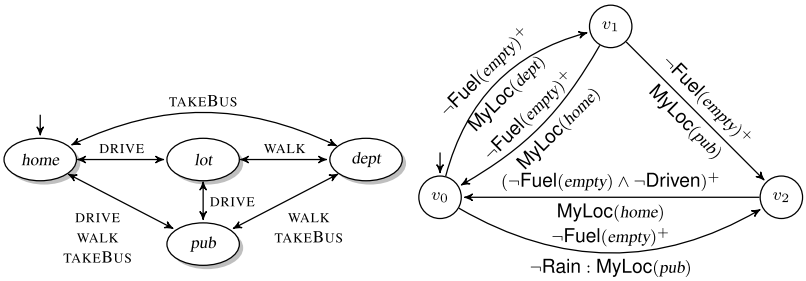
Theorem 1. *There exists a realization Ω of an agent planning program $\mathcal{P}$ in a planning domain $\mathcal{D}$ from $\mathcal{D}$-state s_0 if and only if $v_0 \preceq_{PLAN} s_0$.*

Proof. (If Part) If $v_0 \preceq_{PLAN} s_0$, then for all pairs $v \preceq_{PLAN} s$ and all transitions $d = \langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle$ in $\mathcal{P}$, there exists an HT-plan π that realizes d and preserves $\preceq_{PLAN}$ from $\langle v, s \rangle$ for d. Thus, we define $\Omega(s, d) = \pi$ by taking $\preceq_{PLAN}$ itself as the *PLAN*-simulation relation R.

(Only-If Part) Immediately follows from the definition of realization, which requires the existence of a *PLAN*-simulation R such that $\langle v_0, s_0 \rangle \in R$. Hence, $\preceq_{PLAN}$ being the largest *PLAN*-simulation, we have that $v_0 \preceq_{PLAN} s_0$. □

A planning program $\mathcal{P}$ is said to be <u>realizable</u> in a planning domain $\mathcal{D}$ if there exists a realization of $\mathcal{P}$ in $\mathcal{D}$ from $\mathcal{D}$'s initial state. When that happens, there exists a realization Ω such that *all* possible sequences of legal requests that the agent may issue starting from the initial configuration $\langle v_0, s_0 \rangle$ can be fulfilled by HT-plans returned by Ω.

In the next sections, we will devise techniques to *check* whether an agent planning program $\mathcal{P}$ is realizable in a domain $\mathcal{D}$ and, if so, to actually *build* a corresponding realization function Ω. Before doing so we illustrate the above notions with an example.



(a) Researcher's world map.

(b) Researcher's planning program. Each edge is annotated with its corresponding maintenance goal above, and guard plus achievement goal below. When missing, guards are assumed true.

Fig. 1. Dynamic domain and agent planning program modeling a researcher's everyday-life routine.

3. An example

In this section we illustrate the previous notions (and some of their subtleties) through a simple example on the everyday-life behavior of an academic. The researcher moves among four locations, namely: *home*, the academic *department* building, the department *parking lot*, and a *pub*. To move from one place to another, the researcher can drive a car, take a bus, or just walk. Due to highways, traffic restrictions, and distances, not all alternatives are available from every locations (e.g., it is too far to walk to work and campus circulation is restricted to buses only). In Fig. 1a, all allowed movements in the relevant domain are depicted.

Besides the location of the researcher, the domain includes other features not shown in the figure. For instance, some amount of fuel is consumed each time the car changes location and, at any point in time, it may rain (proposition Rain). In our first, deterministic, version of the domain, though, we assume that it is never raining (i.e., Rain is always false) and the fuel in the car's tank decreases by one level with each action DRIVE, that is, from *full* to *low* and from *low* to *empty*, with each trip (via action DRIVE). As expected, the car cannot be driven when the tank is empty. However, the tank can be brought to its full level by refueling (represented by action REFUEL). We shall later revisit this assumption in a non-deterministic variant of the example.

Let us formalize this planning domain $\mathcal{D} = \langle P, A, \tau \rangle$, as follows:

- $P = \{\mathsf{Fuel}(\mathit{full}), \mathsf{Fuel}(\mathit{low}), \mathsf{Fuel}(\mathit{empty}),$
  $\mathsf{MyLoc}(\mathit{home}), \mathsf{MyLoc}(\mathit{pub}), \mathsf{MyLoc}(\mathit{dept}), \mathsf{MyLoc}(\mathit{lot}),$
  $\mathsf{CarLoc}(\mathit{home}), \mathsf{CarLoc}(\mathit{pub}), \mathsf{CarLoc}(\mathit{lot}), \mathsf{Driven}, \mathsf{Rain}\}$.
- $A = A_{\textsc{drive}} \cup A_{\textsc{walk}} \cup A_{\textsc{takebus}} \cup \{\textsc{refuel}\}$, where:
  - $A_{\textsc{drive}} = \{\textsc{drive}(d) \mid d \in \{\mathit{home}, \mathit{pub}, \mathit{lot}\}\}$;
  - $A_{\textsc{walk}} = \{\textsc{walk}(d) \mid d \in \{\mathit{home}, \mathit{pub}, \mathit{lot}, \mathit{dept}\}\}$; and
  - $A_{\textsc{takebus}} = \{\textsc{takebus}(d) \mid d \in \{\mathit{home}, \mathit{pub}, \mathit{lot}, \mathit{dept}\}\}$.
- $\tau = \tau_{\textsc{drive}} \cup \tau_{\textsc{refuel}} \cup \tau_{\textsc{walk}} \cup \tau_{\textsc{takebus}}$, where:
  - $\tau_{\textsc{drive}} = \Big\{\langle s, \textsc{drive}(l), s' \rangle \mid$
    $l \in \{\mathit{home}, \mathit{pub}, \mathit{lot}\}, \mathsf{Fuel}(\mathit{empty}) \notin s, \mathsf{MyLoc}(\mathit{dept}) \notin s,$
    $\exists l', x, y \cdot l' \in \{\mathit{home}, \mathit{pub}, \mathit{lot}\},$
    $l' \neq l, (x, y) \in \{(\mathit{full}, \mathit{low}), (\mathit{low}, \mathit{empty})\},$
    $\mathsf{MyLoc}(l') \in s, \mathsf{CarLoc}(l') \in s, \mathsf{Fuel}(x) \in s,$
    $s' = (s \setminus \{\mathsf{MyLoc}(l'), \mathsf{CarLoc}(l'), \mathsf{Fuel}(x)\}) \cup$
    $\{\mathsf{MyLoc}(l), \mathsf{CarLoc}(l), \mathsf{Driven}, \mathsf{Fuel}(y)\}\Big\}$;
  - $\tau_{\textsc{refuel}} = \Big\{\langle s, \textsc{refuel}, s' \rangle \mid$
    $\mathsf{MyLoc}(l) \in s, \mathsf{CarLoc}(l) \in s,$
    $s' = (s \setminus \{\mathsf{Fuel}(\mathit{low}), \mathsf{Fuel}(\mathit{empty}), \mathsf{Driven}\}) \cup \{\mathsf{Fuel}(\mathit{full})\}\Big\}$; and
  - $\tau_{\textsc{walk}}$ and $\tau_{\textsc{takebus}}$ are the sets of $\mathcal{D}$'s transitions modeling the effects of actions WALK and TAKEBUS on the researcher's location, resp., as per Fig. 1a; additionally, both actions set Driven to false, to capture that the car has not been driven.

The intended meaning of the propositions in P is self-explanatory: $\mathsf{Fuel}(l)$ denotes the tank level; $\mathsf{MyLoc}(l)$ and $\mathsf{CarLoc}(l)$ denote the location of the researcher and car, respectively; Driven states that the car has *just* been driven in the previous

Table 1
A realization function for the deterministic variant of the researcher's everyday-life domain.

State	Transition	Plan
{MyLoc(*home*), CarLoc(*home*), Fuel(*full*)}	$\langle v_0, v_1\rangle$	⟨DRIVE(*lot*), WALK(*dept*)⟩
{MyLoc(*home*), CarLoc(*home*), Fuel(*full*)}	$\langle v_0, v_2\rangle$	⟨TAKEBUS(*pub*)⟩
{MyLoc(*home*), CarLoc(*lot*), Fuel(*low*)}	$\langle v_0, v_1\rangle$	⟨TAKEBUS(*dept*)⟩
{MyLoc(*home*), CarLoc(*lot*), Fuel(*low*)}	$\langle v_0, v_2\rangle$	⟨TAKEBUS(*pub*)⟩
{MyLoc(*dept*), CarLoc(*lot*), Fuel(*low*)}	$\langle v_1, v_0\rangle$	⟨WALK(*lot*), REFUEL, DRIVE(*home*), REFUEL⟩
{MyLoc(*dept*), CarLoc(*lot*), Fuel(*low*)}	$\langle v_1, v_2\rangle$	⟨WALK(*pub*)⟩
{MyLoc(*pub*), CarLoc(*home*), Fuel(*full*)}	$\langle v_2, v_0\rangle$	⟨TAKEBUS(*home*)⟩
{MyLoc(*pub*), CarLoc(*lot*), Fuel(*low*)}	$\langle v_2, v_0\rangle$	⟨TAKEBUS(*home*)⟩

action; and Rain states that it is raining. Also actions are self-describing. For instance, WALK(*dept*) is the action of the researcher walking to the department building (note there is no action DRIVE(*dept*), as driving in the campus is not allowed).

Executability and effects of actions are captured by the transition relation τ. For example, the set of transitions τ_{DRIVE} represents the transitions between states s and s' when the agent drives to destination location l. The action can only be executed when the car has fuel, and the agent and the car are co-located at l' (different from destination l). After the execution of the action, both the agent and the car are located in l, the car has just been driven, and its tank level has decreased by one unit (see τ_{REFUEL}).

Now imagine that the researcher wants to be able to go to work and, after work, maybe drop by the pub before heading back home. Sometimes, e.g., on weekends, she may want to go to the pub directly from home. For safety reasons, the researcher does *not* want to drive after having been at the pub. Also, a very natural requirement is that the car never runs out of fuel. Such desired behavior can be captured by the agent planning program depicted in Fig. 1b. Each transition is labeled with a triple $\langle \gamma, \psi, \phi\rangle$ encoding the required guard, maintenance and achievement goals, respectively γ, ψ and ϕ. We use the notation $v \xrightarrow{\gamma:\psi^+,\phi} v'$ to denote $v \xrightarrow{\gamma:\psi,(\phi\wedge\psi)} v'$, to abbreviate the common case where the maintenance goal needs to remain true also at the end. For instance, the maintenance goal $(\neg\mathsf{Fuel}(\mathit{empty}) \wedge \neg\mathsf{Driven})^+$ annotating the guard-free transition from state v_2 to state v_0, requires that both the car does not run out of fuel and that the researcher avoids driving after having been at the pub. Notice that such requirements need to hold also when the achievement goal MyLoc(*home*) is fulfilled.

The question is: *can the researcher carry out such a program, and if so, how?* As an example of positive answer, consider Table 1, which describes a possible realization for this program. The first column represents the current state of the domain; the second one contains the requested program transition; and the third one represents the plan to be executed from the current domain state to realize the requested transition. For simplicity, the second column includes only the source and target state of a program transition, while the corresponding guards, maintenance and achievement goals are specified in Fig. 1. Lastly, the third column reports the corresponding HT-plan, as a sequence of actions given that the domain is deterministic.

Consider the first line in the table. If, from the current domain state, the researcher chooses to go to the department (transition $\langle v_0, v_1\rangle$), the corresponding plan consists in driving first to the parking lot and then walking to the department. In the domain state resulting from executing this plan (fifth row in the table), the researcher is at the department and the car is at the parking lot with a low fuel level. From this state, when the researcher chooses to go back home, the corresponding plan consists in walking to the parking lot, refueling the car, driving home and finally refueling the car again. Observe that the first REFUEL action is required to prevent the car from running out of fuel, whereas the second one is not strictly required (the researcher could execute it as the first action of any future plan that includes driving the car). Notice that the state resulting from executing this plan is the one we initially started from. Thus, if the researcher needs to go to work again, the very same plan executed before is still available. Interestingly, this realization example associates the transition $v_0 \to v_1$ with two distinct plans (see third line of the table), depending on the current domain state.

Next consider a nondeterministic variant of this example, in which the fuel level and weather evolves nondeterministically. So, with each trip, the tank level may either stay the same, decrease from *full* to *low* or from *low* to *empty*, and the whether it is raining or not may change at every time-step (i.e., with every action performed). To model the new dynamics for fuel consumption, we replace τ_{DRIVE} for action DRIVE(l) with:

$$\Big\{\langle s, \text{DRIVE}(l), s'\rangle \mid l \in \{\mathit{home}, \mathit{pub}, \mathit{lot}\}, \mathsf{Fuel}(\mathit{empty}) \notin s, \mathsf{MyLoc}(\mathit{dept}) \notin s, \exists l', x, y \cdot l' \in \{\mathit{home}, \mathit{pub}, \mathit{lot}\}, l' \neq l, (x, y) \in \{(\mathit{full}, \mathit{full}), (\mathit{full}, \mathit{low}), (\mathit{low}, \mathit{low}), (\mathit{low}, \mathit{empty})\}, \mathsf{MyLoc}(l') \in s, \mathsf{CarLoc}(l') \in s, \mathsf{Fuel}(x) \in s, s' = (s \setminus \{\mathsf{MyLoc}(l'), \mathsf{CarLoc}(l'), \mathsf{Fuel}(x)\}) \cup \{\mathsf{MyLoc}(l), \mathsf{CarLoc}(l), \mathsf{Driven}, \mathsf{Fuel}(y)\}\Big\};$$

For this scenario, the realization needs to work *no matter* what the outcome of nondeterministic actions turns out to be. It can be seen that there exists a realization of the program for this variant that is similar to the one for the deterministic case, although the plans used are conditional. Observe also that, as a result of nondeterministic actions, the execution of plans may result in on of many states, instead of one only. For instance, take the plan in the first line of Table 1 and consider its execution from the corresponding domain state. As a result of the nondeterminism of DRIVE, after executing the first action, the tank level can be either *low* or *full*. Consequently, after the plan is executed (WALK is not affected by the fuel level), the domain can be in two possible states, i.e., either $\{\mathsf{MyLoc}(\mathit{dept}), \mathsf{CarLoc}(\mathit{lot}), \mathsf{Fuel}(\mathit{full})\}$ or $\{\mathsf{MyLoc}(\mathit{dept}), \mathsf{CarLoc}(\mathit{lot}), \mathsf{Fuel}(\mathit{low})\}$. Thus, in order to realize the planning program, a (HT-) plan must be defined for each of such states. For instance, to realize transition $v_1 \rightarrow v_0$, we need to define a plan that is executable from each of the states above. In our case, it is easy to see that the plan defined in the fifth line of the table can be executed from either state, as the maintenance goal $\neg\mathsf{Fuel}(\mathit{empty})$ (as well as executability of DRIVE) is guaranteed by the execution of REFUEL as the second action of the plan, and then again immediately after DRIVE. Finally, notice that when Rain is true, the program transition $\langle v_0, v_2\rangle$ is not executable, as $\neg\mathsf{Rain}$ is a guard for that transition, in fact simplifying the realization of the planning program.

An actual, more involved, example in the context of smart homes for disabled people is reported in [27], where an early version of agent planning programs is used.

4. General solution technique

In this section, we develop a general solution approach for realizing planning programs, based on the use of synthesis techniques via model checking of two-player game structures. Concretely, we show that checking the existence of a realization of an agent planning program is equivalent to checking whether a strategy exists to force a certain Linear-time Temporal Logic (LTL) formula in a suitable two-player game structure. Moreover from such strategy it is possible to extract an actual realization for the original problem. The main results of this section are a soundness and completeness theorem for the proposed technique, and the characterization of the computational complexity of the problem as EXPTIME-complete.

4.1. LTL synthesis based on two-player game structures

Linear-time Temporal Logic (LTL) is a well-known logic used to specify dynamic or temporal properties of programs [81,102]. *Formulas* of LTL are built from a set $\mathcal{Q}$ of atomic propositions and are closed under the boolean operators, the unary temporal operators $\bigcirc$ (*next*), $\Diamond$ (*eventually*), and $\Box$ (*always*), and the binary temporal operator $\mathcal{U}$ (*until*), which in fact can express both $\Diamond$ and $\Box$ (as $\mathtt{true}\mathcal{U}\phi$ and $\phi\mathcal{U}\mathtt{false}$, respectively). LTL formulas are interpreted over infinite sequences σ of propositional interpretations for $\mathcal{Q}$, i.e., $\sigma \in (2^{\mathcal{Q}})^{\omega}$.[4] The set of (true) propositions at position i is denoted by $\sigma(i)$, that is, $\sigma = \sigma(0), \sigma(1), \ldots$. Given an interpretation σ, a natural number i, and an LTL formula ϕ, we denote by $\sigma, i \models \phi$ the fact that ϕ holds in σ at position i. This is inductively defined as follows, for $p \in \mathcal{Q}$ a proposition, and ϕ, ψ LTL formulas:

$$
\begin{array}{lll}
\sigma, i \models p & \text{iff} & p \in \sigma(i);\\
\sigma, i \models \phi \vee \psi & \text{iff} & \sigma, i \models \phi \text{ or } \sigma, i \models \psi;\\
\sigma, i \models \neg\phi & \text{iff} & \sigma, i \not\models \phi;\\
\sigma, i \models \bigcirc\phi & \text{iff} & \sigma, i{+}1 \models \phi;\\
\sigma, i \models \phi\mathcal{U}\psi & \text{iff} & \text{there exists } k \geq i \text{ such that } \sigma, k \models \psi \text{ and } \sigma, j \models \phi, \text{ for all } j, i \leq j < k;\\
\sigma, i \models \Diamond\phi & \text{iff} & \text{there exists } j \geq i \text{ such that } \sigma, j \models \phi;\\
\sigma, i \models \Box\phi & \text{iff} & \text{for every } j \geq i \text{ we have } \sigma, j \models \phi.
\end{array}
$$

An interpretation σ satisfies ϕ, written $\sigma \models \phi$, if $\sigma, 0 \models \phi$. Standard logical tasks such as satisfiability or validity are defined as usual, i.e., a formula ϕ is *satisfiable* if there exists an interpretation that satisfies it, while it is said to be *valid* if it is satisfied by every possible interpretation. Checking satisfiability or validity of LTL formulas is PSPACE-complete [102].

Satisfiability and validity of LTL (and more in general of temporal) formulas are typical in *verification*. Here we are interested in a different kind of logical task, namely *reactive synthesis* [82,102]. This can be described as follows. Assume $\mathcal{Q}$ is partitioned into two sets $\mathcal{X}$ and $\mathcal{Y}$ of propositions, the former controlled by a module called *environment*, and the latter controlled by a module called *system*. Let the modules interact through the propositions in $\mathcal{Q}$, and have their own internal structure which defines the way they can change proposition values, based on the current assignments. When running, the environment and the system define a compound dynamic system whose evolutions stem from their interaction and that can be described in terms of sequences of assignments to $\mathcal{Q}$. Assume the environment is uncontrollable, that is, we have no way to change its internal structure, while the system is controllable, meaning that we can restrict its behavior. The problem then is: *can we restrict the system behavior to control the values of $\mathcal{Y}$ so that no matter which values the environment assigns to the propositions in $\mathcal{X}$, a desired LTL formula is satisfied?*

Interestingly, LTL synthesis is in general decidable and in fact 2EXPTIME-complete [82], but practically efficient procedures for it are still missing. For this reason, synthesis for special classes of LTL formulas has been investigated. Here we

[4] As standard, notation S^{ω} is used to denote the set of infinite sequences of elements of S.

focus on the class of so-called GR(1) LTL formulas studied in [11]. These include formulas that describe transition systems (those that describe the next state given the current one) and formulas like $\Box\phi$, $\Diamond\phi$, and $\Box\Diamond\phi$ where ϕ is propositional. In particular, for agent planning programs, we will use those in the latter class, which require that ϕ is satisfied infinitely many times. For such GR(1) LTL formulas, we can efficiently reduce synthesis to model checking of a so-called "two-player game structure" [1,11,49,33].

A *two-player game structure*, 2GS for short, is a tuple $G = \langle \mathcal{X}, \mathcal{Y}, I, \rho_e, \rho_s \rangle$, where:

- $\mathcal{X} = \{x_1, \ldots, x_m\}$ and $\mathcal{Y} = \{y_1, \ldots, y_n\}$ are the disjoint finite sets of environment and system propositional variables, respectively. We define the set of *game state variables* as $\mathcal{V} \doteq \mathcal{X} \uplus \mathcal{Y}$ (symbol $\uplus$ denotes disjoint union), and a *game state* as an interpretation of the variables in $\mathcal{V}$. We represent propositional interpretations $i : \mathcal{V} \mapsto \{\top, \bot\}$ as subsets $W \subseteq \mathcal{V}$, adopting the convention that $i(w) = true$ in W if and only if $w \in W$. Interpretations of $\mathcal{X}$ and $\mathcal{Y}$ variables are represented accordingly.
- $I \subseteq \mathcal{V}$ is the (unique) *initial state* of the game.
- $\rho_e \subseteq 2^{\mathcal{X}} \times 2^{\mathcal{Y}} \times 2^{\mathcal{X}}$ is the *environment transition relation*, which relates a game state to its possible successor *environment states*, i.e., $\mathcal{X}$-interpretations.
- $\rho_s \subseteq 2^{\mathcal{X}} \times 2^{\mathcal{Y}} \times 2^{\mathcal{Y}}$ is the *system transition relation*, which relates a current game state to the possible successor *system states*, i.e., $\mathcal{Y}$-interpretations.

Observe that an interpretation $W \subseteq \mathcal{V}$ is partitioned into two components $X \subseteq \mathcal{X}$ and $Y \subseteq \mathcal{Y}$. We often refer to a game state W as (X, Y), under the convention that X and Y represent the corresponding total assignments to $\mathcal{X}$ and $\mathcal{Y}$, respectively.

Intuitively, a 2GS captures the rules of a game where the environment and the system play as opponents. The game starts in the initial state $I = (X_I, Y_I)$, and the players alternate their moves, the environment moving first, by choosing their next state among those their transition relations enable. In details, when the current state of the game is $W = (X, Y)$, the environment chooses some $X' \subseteq \mathcal{X}$ such that $\rho_e((X, Y), X')$, and the system responds with some $Y' \subseteq \mathcal{Y}$ such that $\rho_s((X', Y), Y')$. Such moves lead the game to a new state $W' = (X', Y')$ from which a new round is played, which in turn leads the game to a new state, and so on. We define the game *successor relation* as the relation $\rho \subseteq (2^{\mathcal{X}} \times 2^{\mathcal{Y}}) \times (2^{\mathcal{X}} \times 2^{\mathcal{Y}})$ such that $\rho(W, W')$ if and only if, for $W = (X, Y)$ and $W' = (X', Y')$, $\rho_e((X, Y), X')$ and $\rho_s((X', Y), Y')$. An infinite sequence σ of legal moves starting from the initial state constitutes a *play* of the game, i.e., $\sigma = W_0 W_1 \cdots$ such that $\rho(W_i, W_{i+1})$, for $i \geq 0$. Without loss of generality, we make the assumption that ρ is *serial*, that is, for any finite sequence $\lambda = W_0 \cdots W_n$ such that, for $0 \leq i < n$, $\rho(W_i, W_{i+1})$ holds, there exists W' such that $\rho(W_n, W')$. This corresponds to the intuition that each player can always reply to the opponent, which in turn yields that 2GSs always admit a play.

A 2GS defines the constraints that players must respect when playing, but does not define the goal of the game, or the *winning condition*, i.e., the condition φ that a player needs to achieve in order to win a play. For this, we consider LTL formulas, in particular GR(1) formulas, over propositions in $\mathcal{V}$, and say that a play, which is an LTL interpretation over $\mathcal{V}$, is *winning for the system* if it satisfies φ. Notice that a play captures only a possible evolution of the game, while we are interested in defining when the system can force the game to evolve along a play winning for itself, no matter how the environment moves. To this end we introduce the following notion. Given a 2GS with set of game variables $\mathcal{V} = \mathcal{X} \uplus \mathcal{Y}$, a *strategy for the system* is a partial function $f : (2^{\mathcal{X}})^+ \mapsto 2^{\mathcal{Y}}$ such that: *(i)* $f(X_I) = Y_I$; and *(ii)* for $\ell \geq 0$, if $f(X_0 \cdots X_\ell) = Y_\ell$ is defined, with $X_0 = X_I$, then, for every X such that $\rho_e((X_\ell, Y_\ell), X)$, it is the case that $Y = f(X_0 \cdots X_\ell X)$ is defined and $\rho_s((X, Y_\ell), Y)$. Intuitively, a strategy represents the behavior that the system follows, after having observed a sequence of environment moves. Notice that, by the assumptions on ρ, a strategy for the system always exists. Furthermore, observe that the definition of strategy does not mention the system component explicitly. This is implicitly defined, at each step, by f on those plays where the system acts according to f, which are the only plays of interest to synthesis. Such plays are discussed next.

A play $\sigma = (X_0, Y_0)(X_1, Y_1) \cdots$ is said to be *compliant with a strategy f* if $Y_i = f(X_0 \cdots X_i)$, for all $i \geq 0$. That is, plays compliant with f capture the game evolutions where the system plays according to f. Obviously, given a sequence $X_0 X_1 \cdots$ of environment states from a play σ compliant with f, the system components of σ can be fully reconstructed by subsequent applications of f.

A strategy f is said to be *winning for the system* if for all plays $\sigma = (X_0, Y_0)(X_1, Y_1) \cdots$ compliant with f, it is the case that $\sigma \models \varphi$. When such a strategy exists, the game structure is said to be *winning for the system*, otherwise it is *winning for the environment*. As it turns out, when the system plays according to a winning strategy, all the plays that can stem from the game, and that correspond to different combinations of legal moves of the environment, are guaranteed to satisfy the winning condition. The *synthesis problem* is the problem of constructing, given a 2GS and a winning condition φ, a winning strategy for the system. The *realizability problem* is its decision version, i.e., the problem of checking whether such a strategy exists.

For our purposes, we focus on game winning conditions that belong to the class of so-called *weak-fairness* formulas, i.e., formulas of the form $\Box\Diamond\phi$, where ϕ is propositional, which in turn are in the GR(1) class. Specifically a 2GS together with this winning condition defines a so-called *Büchi game* [49], i.e., a game where the system wins if it can force visiting one of the states in an acceptance set $F \subseteq 2^{\mathcal{X}} \times 2^{\mathcal{Y}}$ infinitely often. In particular, we have $F = \{W \in 2^{\mathcal{X}} \times 2^{\mathcal{Y}} \mid W \models \phi\}$. For this class of games, we have the following result, based on the fixpoint computation of all states from which a play can be forced to achieve a state in F [49].

Theorem 2. *Given a 2GS $G = \langle \mathcal{X}, \mathcal{Y}, I, \rho_e, \rho_s \rangle$ and a winning condition $\varphi = \Box\Diamond\phi$, with ϕ propositional, the realizability and synthesis problems can be solved in time $O(n(n+m))$, where $\mathcal{V} = \mathcal{X} \cup \mathcal{Y}$, $n = 2^{|\mathcal{V}|}$ is the number of states in G, and $m = |\rho_e| + |\rho_s|$ is the number of transitions in G.*

Proof. Direct consequence of the construction of the winning region in [49, Theorem 2.22]. □

4.2. Solving agent planning programs

We now show how to compute a realization of an agent planning program $\mathcal{P}$ in a dynamic domain $\mathcal{D}$ from an initial state s_0, by reduction to synthesis for a 2GS with an LTL weak-fairness formula as winning condition. In the resulting game structure, the environment captures the joint evolution of the domain and the planning program, and the system represents an executor whose available moves are those enabled by the domain. The environment, besides keeping track of the current domain state, requests the next transition to be realized, while the system generates the actions to fulfill the request. Whenever a request is fulfilled, a flag is raised and a new transition is requested by the environment, after which the flag is reset. The winning condition for the system is to make the flag raise infinitely many times, that is, to guarantee that every time a transition is requested, it is eventually realized.

We start by building the 2GS G. We first specify the sets of environment and system propositions $\mathcal{X}$ and $\mathcal{Y}$, then we describe the initial state I of the game structure, and finally we build the transition relations ρ_e and ρ_s. We assume that the planning domain $\mathcal{D}$ starts in the initial state s_0.

Environment and system propositions We define the set of *environment* propositions $\mathcal{X}$ as the disjoint union of the following sets:

- $\mathcal{X}_\mathcal{D} = P$, containing the propositions of the planning domain $\mathcal{D}$;
- $\mathcal{X}_V = V$, containing the states of the planning program $\mathcal{P}$;
- $\mathcal{X}_r = \{req^{v,v'}_{\gamma:\psi,\phi} \mid \langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle \in \delta\}$, containing one proposition per program transition, with $req^{v,v'}_{\gamma:\psi,\phi}$ stating that $\mathcal{P}$'s transition $v \xrightarrow{\gamma:\psi,\phi} v'$ is currently requested.
- $\mathcal{X}_{lr} = \{req^{v,v}_{\alpha:\top,\top} \mid v \in V, \alpha = \bigwedge_{\langle v,\langle\gamma,\psi,\phi\rangle,v'\rangle\in\delta} \neg\gamma\}$, containing one dummy looping request proposition per program state v that can only be requested (i.e., whose guard is true) when no other transition request from v can (i.e., all their guards are false).

Notice that, although the same syntactic symbols are used, the states of $\mathcal{P}$ are interpreted as propositions in the game structure.

The set of *controlled* propositions is defined as $\mathcal{Y} = \mathcal{Y}_A \uplus \{\text{WAIT}, \mathit{init}, \mathit{last}, \mathit{violated}\}$, where $\mathcal{Y}_A = A$. Similarly as above, each action $a \in \mathcal{Y}_A \cup \{\text{WAIT}\}$ is interpreted as a proposition in the game structure denoting the action execution. Distinguished proposition WAIT stands for a no-op action. Proposition *init* is used to mark the initial state, *last* is a special proposition stating that the last action performed has completed the HT-plan under execution, and *violated* represents the fact that some maintenance goal violation has occurred, either in the current or in some past state.

The following syntactic shortcuts will be useful in the following:

- for every $\mathcal{D}$-state $s \in 2^P$ we define a propositional formula $\varsigma_s = \bigwedge_{i=1}^{n} l_i$, where $l_i = p_i$, if $p_i \in s$, and $l_i = \neg p_i$, otherwise. That is, ς_s states the fact that $\mathcal{D}$ is in state s;
- for every $\mathcal{P}$-state $v \in V$ we define a propositional formula $\varsigma_v = v \wedge \bigwedge_{v' \in V, v' \neq v} \neg v'$. That is, ς_v states that $\mathcal{P}$ is in state v; and
- for every program state $v \in V$, we define a propositional formula $req_v = \bigvee_{\langle v,\langle\gamma,\psi,\phi\rangle,v'\rangle\in\delta} req^{v,v'}_{\gamma:\psi,\phi}$. That is, req_v states that (at least) one transition among those available in the state v of the planning program is currently requested.

Initial state The initial (dummy) state is simply defined as $I = \{\mathit{init}\}$, i.e., $X_I = \emptyset$ and $Y_I = \{\mathit{init}, \mathit{last}\}$. Notice that neither the agent planning program nor the domain are in their initial state. However, as it will be clear shortly, this configuration is achieved after the first game transition occurs.

Environment transition relation We describe the transition relations ρ_e and ρ_s declaratively, using simple LTL formulas of the form $\Box\varphi_e$ and $\Box\varphi_s$, respectively, where φ_e and φ_s refer only to the current and the next state (the only temporal operator allowed in these formulas is $\bigcirc$). We adopt the convention that a pair $\langle W, W' \rangle$, with $W \subseteq \mathcal{V}$ and $W' \subseteq \mathcal{X}$ (respectively, $W' \subseteq \mathcal{Y}$), is in the transition relation of the environment (resp., of the system) if and only if, for some sequence σ starting with the prefix W, W', i.e., $\sigma = WW' \cdots$, it is the case that σ satisfies φ_e (resp., φ_s). For instance, if $\varphi_e = p \wedge \neg\bigcirc p$ is the formula defining ρ_e, then $\langle \{p\}, \emptyset \rangle \in \rho_e$, as $\sigma \models \varphi_e$ for any sequence $\sigma = \{p\}\emptyset \cdots$, while $\langle \{p\}, \{p\} \rangle \notin \rho_e$, as for no sequence $\sigma = \{p\}\{p\} \cdots$, it is the case that $\sigma \models \varphi_e$.

The transition relation ρ_e is captured by the formula $\varphi_e = trans_{\mathcal{D}} \wedge trans_{\mathcal{P}}$, where $trans_{\mathcal{D}}$ and $trans_{\mathcal{P}}$ capture the transition relations of the domain and the planning program, respectively. In words, φ_e encodes the synchronous execution of the domain and the planning program, taking into account, when needed, the value of the auxiliary variables *init* and *last*.

Technically, $trans_{\mathcal{D}}$ is obtained as a conjunction of the following formulas:

E1 $init \rightarrow \bigcirc \varsigma_{s_0}$, encoding that the domain is in its initial state, after the initial (dummy) move.
E2 $\bigwedge_{s \in 2^P} (\varsigma_s \wedge \textsc{wait} \rightarrow \bigcirc \varsigma_s)$, expressing that the domain remains still on action WAIT.
E3 $\bigwedge_{s \in 2^P, a \in \mathcal{Y}_A} \left(\varsigma_s \wedge a \rightarrow \bigcirc \bigvee_{\langle s,a,s' \rangle \in \tau} \varsigma_{s'}\right)$, expressing that if the domain is in state s, action a is to be executed next (which can happen only if the current game state is not I), then all possible successor states of s reachable through τ by executing a can occur next (we assume that an empty set of disjuncts equals false).

Notice that we use the formulas above to encode transitions only for simplicity. In practice, it is not needed to list all of them explicitly, but a compact representation can be used. An example of this appears in Section Appendix A, where we report the encoding in the concrete language SMV used (in slightly different variants) by the systems TLV, JTLV and NuGaT.

As to $trans_{\mathcal{P}}$, it is the conjunction of the following formulas:

E4 $init \rightarrow \bigcirc v_0$, which encodes that the planning program is initially in its initial state.
E5 $\bigvee_{v \in \mathcal{X}_V} \bigcirc [v \wedge \bigwedge_{v' \in \mathcal{X}_V \setminus \{v\}} \neg v']$, which encodes that the planning program can move to exactly one of its states.
E6 $\bigwedge_{v \in \mathcal{X}_V} \bigcirc [v \rightarrow req_v]$, which encodes that at least one transition available in the state the planning program moves to must be requested next.
E7 $last \rightarrow \bigwedge_{\langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle \in \delta} \bigcirc [req^{v,v'}_{\gamma:\psi,\phi} \rightarrow \gamma]$, which expresses that a new transition $v \xrightarrow{\gamma:\psi,\phi} v'$ can be requested only if, at the time of issuing the request (i.e., after *last* holds), guard γ is satisfied.
E8 $\bigwedge_{req, req' \in \mathcal{X}_r, req \neq req'} \bigcirc [req \rightarrow \neg req']$, that is, at most one program transition can be requested at a time.
E9 $\bigwedge_{\langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle \in \delta} [req^{v,v'}_{\gamma:\psi,\phi} \wedge last \rightarrow \bigcirc v']$, capturing that if transition $v \xrightarrow{\gamma:\psi,\phi} v'$ is currently requested and the last action performed has completed the current HT-plan, then the planning program moves to v'.
E10 $\bigwedge_{v \in \mathcal{X}_V} [(v \wedge \neg last) \rightarrow \bigcirc v]$, which expressing that the program remains still if the current HT-plan has not been completed.
E11 $\bigwedge_{req \in \mathcal{X}_r} [(req \wedge \neg last) \rightarrow \bigcirc req]$, capturing that the agent remains requesting the same transition if the current HT-plan has not been completed.

Notice that the environment can always make a move. In particular, when the game represents a program state v for which no actual transition can be requested in the current domain state—all guards are false—the environment can play the dummy transition request included in set $\mathcal{X}_{lr}$ for state v. This, together with the fact that every executable action yields at least one next domain state, guarantees that ρ_e is *serial*, that is, every state has a successor. Observe also that the last two formulas of $trans_{\mathcal{D}}$ and the last three formulas for $trans_{\mathcal{P}}$ trivially evaluate to true in the initial game state I—they do not constrain the first move of the environment.

System transition relation We now build φ_s, the formula that captures system player's transition relation ρ_s, i.e., the capabilities of the system. In other words, formula φ_s shall capture when actions can be executed and when the HT-plan under execution can be declared to be completed (via proposition *last*). The system also keeps track of maintenance goal violations (via proposition *violated*). The formula φ_s is the conjunction of the following subformulas:

S1 $\varphi_{\text{init}} = \bigcirc \neg init$, which states that *init* holds only in the initial state.
S2 $\varphi_{\text{act}} = \bigvee_{a \in \mathcal{Y}_A \cup \{\textsc{wait}\}} \bigcirc [a \wedge \bigwedge_{a' \in \mathcal{Y}_A, a' \neq a} \neg a']$, that is, *exactly* one domain action, or no-op WAIT action, is executed at each step.
S3 $\varphi_{\text{pre}} = \bigwedge_{a \in \mathcal{Y}_A} \bigcirc [a \rightarrow \bigvee_{\langle s,a,s' \rangle \in \tau} \varsigma_s]$, which requires that domain action a can be executed only if the domain is in a state s where the action is executable, i.e., its precondition is fulfilled.
S4 $\varphi_{\text{wait}} = \bigcirc [\textsc{wait} \leftrightarrow (last \vee \bigwedge_{\langle s,a,s' \rangle \in \tau} \neg \varsigma_s)]$, which requires that the no-op action WAIT is executed if and only if *last* holds or no domain action can be performed (i.e., the precondition of every action is false).
S5 $\varphi_{\text{last}} = \bigwedge_{\langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle \in \delta} \bigcirc [(req^{v,v'}_{\gamma:\psi,\phi} \wedge last) \rightarrow (\phi \wedge \neg violated)]$, expressing that an HT-plan can be declared completed only if the achievement goal ϕ of the transition currently requested is indeed achieved and no violation of a maintenance goal has (ever) occurred.
S6 $\varphi_{\text{maint}} = \bigcirc \big[\bigwedge_{\langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle \in \delta} req^{v,v'}_{\gamma:\psi,\phi} \wedge \neg\psi \wedge \neg last \rightarrow violated\big] \wedge \big[\neg violated \wedge \bigcirc (last \vee \bigwedge_{\langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle \in \delta} (req^{v,v'}_{\gamma:\psi,\phi} \rightarrow \psi))) \rightarrow \bigcirc \neg violated\big]$, expressing that a violation occurs if and only if the maintenance goal ψ of the requested transition is not satisfied. Note that non-satisfaction of the maintenance formula in the final step of a plan's execution (i.e., when *last* holds) is not considered a violation (refer to definition of a plan maintaining a goal in page 67).
S7 $\varphi_{\text{violated}} = (violated \rightarrow \bigcirc violated)$, which expresses that violations, once occurred, are recorded forever.

The behavior of the resulting 2GS can be summarized as follows. The environment initially sets the agent planning program and the domain in their respective initial states, and nondeterministically picks a program transition to be realized (E1, E4–E8). At every step, the system can reply to the environment by either following a plan to realize the current transition, thus choosing a domain action whose precondition holds in the current domain state, or by announcing the end of the current plan, that is the realization of the transition, by setting special proposition *last* to true and selecting special action WAIT (S1–S5). In the former case, the environment replies by simply executing the action, thus progressing the domain to one of its possible successor states (given the current state and the action chosen by the system) and keeping the planning program in its current state, with same transition request (E3, E10, E11). If, instead, a transition realization is announced, i.e., the last action of the plan has been executed (proposition *last*), then the domain remains still ("waits"), while the agent planning program is progressed, according to the current transition requested, to the successor state, and a new transition, outgoing from the new state, is selected for realization (E2, E9). Notice that in order for the system to set *last* true, the achievement goal must be fulfilled (S5). Also, when selecting a domain action, the system may choose one that violates the maintenance goal of the requested program transition. In this case, as soon as the violation occurs, proposition *violated* becomes true and remains so forever (S6, S7). Finally, observe that proposition *last* can be set true by the system only if no violation has occurred (S5). As a result, the system can declare a transition realized only if the corresponding achievement goal has been actually achieved and its maintenance goal has not been violated.

We note that because the system can always play WAIT when no domain action is executable, analogously to ρ_e also the transition relation ρ_s is serial—there is always a next available system move. This implies that the game successor relation ρ—built from ρ_e and ρ_s—is in turn serial, thus every game state has at least one successor.

Once the 2GS is defined, we can use a weak-fairness formula to encode the synthesis goal. Formally, we have:

$$\varphi_{\text{goal}} = \Box\Diamond \mathit{last}\,.$$

It can be seen that, as a consequence of the constraints implied by φ_{last}, φ_{maint}, and $\varphi_{\text{violated}}$, φ_{goal} is satisfied by a play if and only if the achievement goal of every request is eventually satisfied and no maintenance requirement is ever violated. Indeed, the current program transition is eventually realized if and only if *last* is eventually set to true. When this happens, a new transition request is issued, which requires *last* to eventually hold again, after which a new program transition will be requested, and so on and so forth.

In Appendix A we present an actual encoding, obtained by following the construction above, of the nondeterministic variant of the example presented in Section 3.

The following result shows correctness of the above construction, by linking the existence of a winning strategy for the system in the 2GS defined above with the existence of a realization of the agent planning program.

Theorem 3 *(Soundness & completeness). There exists a realization of an agent planning program $\mathcal{P}$ in a planning domain $\mathcal{D}$ from a state s_0 if and only if, for the 2GS G and the winning condition φ_{goal} defined above, there exists a strategy that is winning for the system.*

Proof. The proof consists in showing how from a strategy for the 2GS that is winning for the system, one can derive a realization for the agent planning program and, viceversa, how from a realization of the agent planning program, one can derive a winning strategy for the game. See Appendix B for full details. □

That is, computing a winning strategy for the synthesis problem defined by the 2GS and the winning condition above is equivalent to realizing the planning program $\mathcal{P}$ in $\mathcal{D}$.

Next we analyze the worst-case computational complexity of the problem. By Theorem 2, we have that a winning strategy for φ_{goal} in G can be computed in time $O(n(n+m))$, with n the number of states in G and $m = |\rho_e| + |\rho_s|$. Since $|\rho_e|, |\rho_s| \leq n^2$, we get a polynomial bound $O(n^3)$. However, $n \leq 2^{|\mathcal{V}|}$, thus checking the existence of a solution (and actually constructing it) can be done in time $O(2^{3|\mathcal{V}|})$. Considering the definition of ρ_e (conjuncts E6 and E9 of $trans_{\mathcal{P}}$) and ρ_s, the number of states n is $O(2^{|P|} \cdot |\delta| \cdot |A|)$, as $|\mathcal{X}_r| = |\delta|$. In other words, our technique is exponential in the number of domain propositions, while polynomial in the size of the planning program and number of domain actions.

For the lower bound, we observe that checking the existence of a conditional plan for an achievement goal in a nondeterministic planning domain with full observability is EXPTIME-hard [69,90]. Such a form of planning is a special case of our problem, where we have a planning program consisting of a single transition labelled with an achievement goal only. Hence, the technique presented here is giving us a tight complexity characterization for solving agent planning problems.

Theorem 4 *(Complexity). Checking whether an agent planning program is realizable in a planning domain from a given initial state is EXPTIME-complete.*

Proof. Direct consequence of the discussion above. □

Interestingly, in spite of the additional sophistication of agent planning programs, the complexity of realizing them is essentially the same as that of conditional planning (with full observability). In other words, at least from the worst-case

complexity point of view, realizing agent planning programs does not require any additional computational effort with respect to conditional planning.

Finally, we observe that the kind of solution based on the above technique shares several commonalities with the notion of *universal plan* [92], in the sense that from every configuration (of planning program state and domain state) a way to fulfill the winning condition by winning the game is provided. Obviously, the class of winning conditions considered here is not reachability (of a state satisfying the goal, as for universal plans), but a more sophisticated one expressing the ability to reach infinitely often a state where *last* holds. It should be clear, however, that such a solution shares the same criticality of universal plans, including its practical cost (see also [48]).

5. Planning programs in deterministic domains

In this section, we focus on the notable case in which the agent acts in a deterministic domain. A *deterministic planning domain* [46] is a special case of planning domain $\mathcal{D} = \langle P, A, \tau \rangle$, where the transition relation τ is a function $\tau : 2^P \times A \mapsto 2^P$. We call this case the "deterministic case" and for it we develop an alternative realization technique deeply rooted in the planning technology which consists of suitable calls to a classical planner, careful iterated till the entire planning program is realized. This iterative method is similar to the one implemented in planner NDP to solve non-deterministic planning problems [64], which constructs policies by iterative calls to a classical planner. However, we use some specific planning techniques that are not present in NDP. The kind of solution that our realization algorithm for deterministic domains devises has not the "universal plan" nature of the general procedure presented above, and empirically proves to be quite effective especially for agent planning programs over planning domains that have limited or no deadends in the search space, as shown later.

Before going on, we characterize the computational complexity of the deterministic case. Obviously, the general EXPTIME technique shown above applies to the deterministic case as well so this gives us an EXPTIME upper-bound. However the reduction from conditional planning with full observability that we use for the lower bound only gives us a PSPACE-hardness lower-bound for deterministic domains. So the question is: *is the problem EXPTIME-hard even in the deterministic case or does it admit a PSPACE algorithm?*

We answer this question by showing the EXPTIME-hardness also in the deterministic case. To do so we resort to a reduction from the *composition problem of deterministic agent behaviors*, which is known to be EXPTIME-complete [35,76].

Theorem 5. *Checking whether an agent planning program is realizable in a deterministic planning domain is EXPTIME-complete.*

Proof. It can be shown that composition of deterministic agent behaviors can be polynomially encoded into realization of agent planning programs. Details are in Appendix B. □

Next we detail the specific planning-based technique that we propose to handle the deterministic case.

5.1. Realizing planning programs for the deterministic case

In the rest of this section, for technical convenience, an action is represented as a triple $\langle Pre,\ Eff^+,\ Eff^- \rangle$ where Pre is a set of propositions representing the action preconditions, and $Eff^{+/-}$ is a set of propositions representing the action positive/negative effects. Like in classical planning [46], under the closed world assumption, a state is specified by a set of propositions, an action $a = \langle Pre,\ Eff^+,\ Eff^- \rangle$ is said to be *executable* in a domain state s if $Pre \subseteq s$, and the domain state s' obtained by executing a in state s is $s \setminus Eff^- \cup Eff^+$. The domain transition function τ is (implicitly) defined by the execution of the domain actions from all the possible domain states.

Observe that, because in a deterministic planning domain the execution of an HT-plan produces only a single history, an HT-plan can be simply represented as a sequence π of actions: the corresponding HT-plan function can be obtained by associating each action a of π with the sequence of states obtained by executing, from the initial state of the domain, all the actions that precede a in π. Thus, since in this section we deal with deterministic domains only, for simplicity we represent HT-plans in this form.

A state s of $\mathcal{D}$ is said to be *reachable* from an initial state s_0 if there exists a plan π such that s is the final state obtained by executing π from s_0. In the following, for a domain $\mathcal{D}$, we use $S \subseteq 2^P$ to denote the set of all states of a domain that are reachable from an initial state s_0. Further, $\pi(s)$ denotes the sequence of states obtained by executing π from state s and $last(\pi(s))$ denotes the final state of such sequence.

We address the problem of *effectively* constructing planning program realizations for deterministic domains by exploiting plan generation techniques for planning problems with *preferred end-states* (shortly, PESs) and *tabu end-states* (TESs). A PES is a desired end state for a plan realizing a planning program transition, while a TES is a forbidden plan end state. As will be described, PESs and TESs are generated by the proposed iterative algorithm for realizing agent planning programs, and they are important to guarantee its correctness and efficiency.

Algorithm: RealizePlanProg($\mathcal{P}, \mathcal{D}, s_0$)
Input: a planning program $\mathcal{P} = \langle P, V, v_0, \delta \rangle$, a deterministic planning domain $\mathcal{D} = \langle P, A, \tau \rangle$, and an initial state s_0;
Output: a realization of $\mathcal{P}$ in $\mathcal{D}$ from s_0 (Function Ω), or `failure`.

1. $\forall s, d \cdot \Omega(s, d) \leftarrow$ `noPlan`;
2. $States(v_0) \leftarrow \{s_0\}$; $\forall v \neq v_0 \cdot States(v) \leftarrow \emptyset$;
3. $\forall v \cdot Tabu(v) \leftarrow \emptyset$;
4. $Open \leftarrow \{\langle s_0, v_0 \rangle\}$;
5. **while** *Open* is *not* empty **do**
6. **extract** an open pair $\langle s, v \rangle \in Open$;
7. $\pi \leftarrow$ `noPlan`;
8. **foreach** program transition $d = \langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle \in \delta$ **do**
9. **if** $\Omega(s, d) =$ `noPlan` **and** $s \models \gamma$ **then**
10. $\pi \leftarrow \mathsf{Plan}(s, A, \psi, \phi, States(v'), Tabu(v'))$;
11. **if** π is `failure` **then break**;
12. **else**
13. $\Omega(s, d) \leftarrow \pi$;
14. **if** $last(\pi(s)) \notin States(v')$ **then**
15. **add** $\langle last(\pi(s)), v' \rangle$ **to** *Open*;
16. **add** $last(\pi(s))$ **to** $States(v')$;
17. **add** $\langle s, d \rangle$ **to** $Source(last(\pi(s)), v')$;
18. **if** π is `failure` **then**
19. **if** $\langle s, v \rangle = \langle s_0, v_0 \rangle$ **then return** `failure`;
20. **else**
21. **add** s **to** $Tabu(v)$;
22. **remove** s **from** $States(v)$;
23. **foreach** $\langle s'', d'' \rangle$ *s.t.* d'' is a $\mathcal{P}$ transition from v'' to v and $\langle s'', d'' \rangle \in Source(s, v)$ **do**
24. $\Omega(s'', d'') \leftarrow$ `noPlan`;
25. $Open = Frontier(\Omega, \tau, s_0, v_0)$;
26. **return** Ω.

Fig. 2. Algorithm for realizing a planning program $\mathcal{P}$ in a deterministic planning domain $\mathcal{D}$ from state s_0.

Definition 5. A *planning problem with PESs and TESs* is a tuple $\langle \mathcal{D}, s_0, \psi, \phi, S_P, S_T \rangle$ where $\mathcal{D} = \langle P, A, \tau \rangle$ is a deterministic planning domain, s_0 is the initial state, $\psi \in \Phi(P)$ is a maintenance goal, $\phi \in \Phi(P)$ is an achievement goal; $S_P \subseteq 2^P$ is a set of PESs; and, finally, $S_T \subseteq 2^P$ is a set of TESs. □

Given a planning problem $\Pi = \langle \mathcal{D}, s_0, \psi, \phi, S_P, S_T \rangle$ with PESs and TESs, an executable plan π for $\mathcal{D}$, and state $s' = last(\pi(s_0))$, we say that π is *valid* for Π iff π maintains ψ, $s' \models \phi$ and $s' \notin S_T$. Moreover, given two valid plans π_1 and π_2 for Π, we say that π_1 is *preferred* to π_2 iff $last(\pi_1(s_0)) \in S_P$ and $last(\pi_2(s_0)) \notin S_P$.

Fig. 2 shows the pseudo-code of RealizePlanProg, an algorithm for building planning program realizations. Starting from an open configuration (called *open pair* in the algorithm) $\langle s, v \rangle$, where s is a domain state and v is a planning program state (initially $s = s_0$ and $v = v_0$), for each transition d outgoing from v such that the guard of d holds in s, RealizePlanProg constructs a plan π realizing d from s. Then, the algorithm progresses the states of $\mathcal{D}$ and $\mathcal{P}$ (according to $\pi(s)$ and d, respectively), possibly generating a new open pair $\langle s', v' \rangle$ to process similarly. For each generated pair $\langle s, v \rangle$ and transition $d = \langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle$ such that $s \models \gamma$, function $\Omega(s, d)$ associates with s a plan constructed to achieve ϕ from s while maintaining ψ. If the algorithm generates an open pair $\langle s, v \rangle$ such that for some transition outgoing from v no realizing plan can be computed from s, backtracking is required, i.e., the plans generating $\langle s, v \rangle$ need to be removed from Ω. The algorithm terminates when no more open pairs are left, or it is the case that no realization can be found, i.e., for at least a transition $d = \langle v_0, \langle \gamma, \psi, \phi \rangle, v \rangle$ outgoing from the initial planning program state v_0, and such that γ holds in the initial domain state s_0, there exists no plan π constructed from s_0 such that π maintains ψ, $last(\pi(s_0)) \models \phi$ and $last(\pi(s_0))$ is in the set of domain states from which a transition outgoing from v can be realized.

The specification of function Ω under construction implicitly defines the set of open pairs, also called the *realization frontier*, which is denoted in the algorithm as *Open*. This set is obtained by considering all possible planning program executions, starting from $\langle s_0, v_0 \rangle$, using Ω to realize the transitions, and putting in the set all those pairs $\langle s, v \rangle$ such that for some transition d from v, the guard of d holds in s and $\Omega(s, d)$ is currently undefined. Essentially, this corresponds to a straightforward visit of the planning program graph from v_0 and s_0 using the current (partially defined) Ω. The frontier of this visit is the set of pairs $\langle s, v \rangle$, of domain and planning program state, such that there is a transition d outgoing from v whose guard holds in s, but for which there is no plan achieving and maintaining the corresponding goal, i.e., $\Omega(s, d)$ is undefined. Such a frontier is denoted by $Frontier(\Omega, \tau, s_0, v_0)$ and defines the open pairs for the current Ω stored in *Open*.

For example, assume that the current specification of Ω is defined by the first two lines of Table 1. Then, the realization frontier is the set of open pairs

$$\big\{\langle\{\mathsf{MyLoc}(dept), \mathsf{CarLoc}(lot), \mathsf{Fuel}(low)\}, v_1\rangle, \langle\{\mathsf{MyLoc}(pub), \mathsf{CarLoc}(home), \mathsf{Fuel}(full)\}, v_2\rangle\big\}.$$

The former pair, for instance, is reached by executing the first plan in Table 1 that realizes transition $\langle v_0, \langle \emptyset, \neg\mathsf{Fuel}(empty), \mathsf{MyLoc}(dept)\rangle, v_1\rangle$ from the initial state $s_0 = \{\mathsf{MyLoc}(home), \mathsf{CarLoc}(home), \mathsf{Fuel}(full)\}$, and it is in the frontier because (a) the

transition has no guard (hence, it needs to be realized) and (b) the current specification of Ω is still undefined for the domain state $\{\mathsf{MyLoc}(dept), \mathsf{CarLoc}(lot), \mathsf{Fuel}(low)\}$ and transitions $\langle v_1, \langle \top, \neg\mathsf{Fuel}(empty), \mathsf{MyLoc}(home) \wedge \neg\mathsf{Fuel}(empty)\rangle, v_0\rangle$ and $\langle v_1, \langle \top, \neg\mathsf{Fuel}(empty), \mathsf{MyLoc}(pub) \wedge \neg\mathsf{Fuel}(empty)\rangle, v_2\rangle$.

Algorithm RealizePlanProg maintains three auxiliary functions $States : V \to 2^S$, $Tabu : V \to 2^S$ and $Source$: $S \times V \to 2^{S\times\delta}$. Intuitively, $States(v)$ records all domain states reached when $\mathcal{P}$ is in v, for some $\mathcal{P}$ execution, according to the current Ω; $Tabu(v)$ indicates the states of $\mathcal{D}$ that are forbidden when v is reached; and $Source$ associates each open pair $\langle s', v'\rangle$ with those pairs $\langle s, d\rangle$ such that d is a program transition from v to v' and, for $\pi = \Omega(s, d)$, $last(\pi(s)) = s'$. Essentially, function $Source$ says how an open pair was generated by the current Ω.

Initially (lines 1–4), Function Ω is completely undefined (through the special value `noPlan`), $States(v) = \emptyset$ for every $v \neq v_0$, $States(v_0) = \{s_0\}$, $Tabu(v) = \emptyset$ for every v, and $Open = \langle s_0, v_0\rangle$. At each iteration of the external loop (lines 5–25), an arbitrary open pair $\langle s, v\rangle$ is extracted from $Open$ and processed by:

(i) computing, for each transition $d = \langle v, \langle \gamma, \psi, \phi\rangle, v'\rangle$ such that $s \models \gamma$ and $\Omega(s, d) = \texttt{noPlan}$ (i.e., d has not been processed for s yet), a plan π that maintains ψ, achieves ϕ from s with an acceptable end state, i.e., $last(\pi(s_0)) \notin Tabu(v')$ (lines 8–10);
(ii) appropriately updating Ω, $Open$, and the auxiliary functions (lines 11–25).

When $Open$ becomes empty, the external loop terminates and the algorithm returns Ω (line 26).

Task *(i)* is accomplished by executing function Plan, which computes a plan for the planning problem with PESs and TESs $\langle \mathcal{D}, s, \psi, \phi, States(v'), Tabu(v')\rangle$. Intuitively, the domain states in $States(v')$ are used as preferred end states to minimize the number of generated open pairs, while the domain states in $Tabu(v')$ are used as tabu end states to prevent next iterations from generating unrealizable open pairs. Details about how to achieve this behavior in Plan are given in Section 5.2.

For task *(ii)*, assume that $\langle s, v\rangle$ is an open pair, and d is a program transition from program state v to program state v', whose guard holds in s. If a plan π realizing d from s is found, then the algorithm updates $\Omega(s, d)$, $States(v')$ and $Source(s', v')$ as follows: function Ω is updated by setting $\Omega(s, d)$ to π; if $s' = last(\pi(s))$ is not already in $States(v')$, the set of open pairs is extended with $\langle s', v'\rangle$; state s' is added to $States(v')$; and $\langle s, d\rangle$ is added to $Source(s', v')$ (lines 13–17). If for some program transition d outgoing from v such that its guard holds in s, procedure Plan is unable to find a plan achieving/maintaining the goals of d from s, then open pair $\langle s, v\rangle$ cannot be realized. In the special case $s = s_0$ and $v = v_0$, no realization of $\mathcal{P}$ can be built, and hence RealizePlanProg terminates returning failure (lines 18–19). Otherwise ($s \neq s_0$ or $v \neq v_0$), backtracking is performed on Ω (lines 21–25): s is added to $Tabu(v)$; s is removed from $States(v)$, as clearly no longer preferred; for all pairs $\langle s'', v''\rangle \in Source(s, v)$, $\Omega(s'', d'')$ is set undefined ($\Omega(s'', d'')$ becomes `noPlan`), as the corresponding plans need to be recomputed in order to avoid generating the configuration $\langle s, v\rangle$; and, finally, $Frontier(\Omega, \tau, s_0, v_0)$ defines the new set of open pairs ($Open$).

Interestingly, RealizePlanProg is parametric with respect to the specific planning procedure used to implement Plan, thus allowing us to generate different version of our algorithm based on different planning approaches and heuristics.

The following results demonstrate the fundamental properties of RealizePlanProg.

Lemma 1. *Algorithm RealizePlanProg terminates provided that subroutine Plan terminates.*

Theorem 6 *(Soundness). The function computed by Algorithm RealizePlanProg is a realization of the input agent planning program $\mathcal{P}$, deterministic planning domain $\mathcal{D}$ and initial domain state s_0, provided that subroutine Plan is sound to solve planning problems with achievement and maintenance goals.*

Theorem 7 *(Completeness). Assume that subroutine Plan is complete. Algorithm RealizePlanProg returns a realization of the input planning program $\mathcal{P}$, if it exists; otherwise it returns* `failure`.

Proofs for all three claims can be found in Appendix B.

5.2. Encoding preferred and tabu end-states into actions with costs

A planning problem with PESs and TESs can be expressed in PDDL3 [43]. In particular, a TES s can be specified by an "`at end`" constraint (an additional goal formula constraining the goal state) imposing that the disjunction of the negation of the propositions that are true in s and the propositions that are false in s hold at the end of the plan. (Under the closed world assumption, a proposition p is true in s if $p \in s$, while it is false in s if $p \notin s$, assuming s formalized as a set of propositions.) Similarly, a PES s can be specified by a preferred goal (also called *soft goals*) imposing that the conjunction of the propositions that are true in s and the negation of the propositions that are false in s preferably holds at the end of the plan.

A planning problem with soft goals and constraints can be translated into a classical planning problem with action costs, that can be solved by classical planners supporting real-valued fluents [43]. Keyder and Geffner [62] show that classical planners can solve the problems obtained by their translation scheme for compiling soft goals away more quickly than what it takes to solve the original problems with soft goals.

In this section, we propose a scheme to transform a problem with PESs and TESs into a problem with action costs, that is much simpler than the one proposed in [43], as it considers only a special case of the planning problem with soft goals and constraints studied in [43]. Concerning the compilation of PESs, our scheme also differs from the one by Keyder and Geffner both in its purpose and compilation technique. Our compilation is designed for the particular context in which it is used (realizing program transitions involved in a loop) and the type of soft goals that are relevant in this context (preferred end states). We do not propose a method that provides a general translating scheme for compiling soft goals away, as in [62]. Instead, our scheme constructs a planning problem Π' with action costs from a planning problem Π with PESs so that, if a planner finds a solution plan of Π' with the lowest cost, such a plan can be easily transformed into a solution plan of Π ending in one of the PESs of Π. Moreover, in our context any valid plan can satisfy at most one preference, and our scheme compiles also TESs, while the compilation scheme described in [62] handles only soft goals.

Definition 6. A *planning problem with action costs* is a tuple $\langle \mathcal{D}, s_0, \psi, \phi, c \rangle$ where $\mathcal{D} = \langle P, A, \tau \rangle$ is a deterministic planning domain, s_0 is the initial state, $\psi \in \Phi(P)$ is a maintenance goal, $\phi \in \Phi(P)$ is an achievement goal; and $c : A \mapsto \mathbb{R}$ is an action cost function. □

A planning problem with PESs and TESs $\Pi = \langle \mathcal{D}, s_0, \gamma, \psi, \phi, S_P, S_T \rangle$ where $\mathcal{D} = \langle P, A, \tau \rangle$ can be translated into a planning problem with action costs $\Pi' = \langle \mathcal{D}', s_0', \psi, \phi', c \rangle$ such that[5]:

- $\mathcal{D}' = \langle P', A', \tau' \rangle$;
- $P' = P \cup P_M \cup P_T$;
- $A' = A^+ \cup A_P \cup A_T$;
- τ' is implicitly defined by the preconditions/effects of the actions in A';
- $s_0' = s_0 \cup \{\texttt{normal-mode}\}$;
- $\phi' = \phi \wedge \texttt{check-pref} \wedge \bigwedge_{p_t \in P_T} p_t$;
- $c(o) = \begin{cases} 1 & \text{if } o = \texttt{Ignore-pref}, \\ 0 & \text{otherwise;} \end{cases}$

where

- $P_M = \{\texttt{normal-mode}, \texttt{end-mode}, \texttt{check-pref}\}$;
- $P_T = \{\texttt{not-tabu}(s) \mid s \in S_T\}$;
- $A^+ = \{\langle \mathit{Pre} \cup \{\texttt{normal-mode}\}, \mathit{Eff}^+, \mathit{Eff}^- \rangle \mid \langle \mathit{Pre}, \mathit{Eff}^+, \mathit{Eff}^- \rangle \in A\}$;
- $A_P = \texttt{Ignore-pref} \cup \{\texttt{Sat-pref}(s) \mid s \in S_P\}$, where $\texttt{Ignore-pref}$ is the action $\langle \{\texttt{normal-mode}\}, \{\texttt{end-mode}, \texttt{check-pref}\}, \{\texttt{normal-mode}\} \rangle$ and $\texttt{Sat-pref}(s)$ is the same as $\texttt{Ignore-pref}$ but with the additional set of preconditions $\{p \mid p \in s\} \cup \{\neg p \mid p \in P \wedge p \notin s\}$;
- $A_T = \{a \mid a \in \texttt{Act-tabu}(s) \wedge s \in S_T\}$, where $\texttt{Act-tabu}(s)$ is the set of actions $\{\langle \{\texttt{end-mode}, \neg p\}, \{\texttt{not-tabu}(s)\}, \emptyset \rangle \mid p \in P \wedge p \in s\} \cup \{\langle \{\texttt{end-mode}, p\}, \{\texttt{not-tabu}(s)\}, \emptyset \rangle \mid p \in P \wedge p \notin s\}$.

It is easy to see that the structure of any plan for the translated problem is $\langle \pi_{A^+}, a, \pi_T \rangle$, π_{A^+} and π_T are two (possibly empty) sub-plans of actions in A^+ and A_T, respectively, and $a \in A_P$. The (possible) presence of action $\texttt{Sat-pref}(s)$, for some s in the plan, indicates that $\mathit{last}(\pi_{A^+})$ is the preferred domain state s. The (required) presence of an action of $\texttt{Act-tabu}(s)$ in π_T, for some tabu state s, indicate that the end state generated by subplan π_{A^+} is different from $s \in S_T$. Note that since the conjunction goal formula ϕ' contains a conjunct $\texttt{not-tabu}(s)$ for each tabu state $s \in S_T$, subplan π_T must contain an action of A_T for each $s \in S_T$.

The cost of a plan π is the sum of the cost of the actions executed in π. Since there can be at most one occurrence of action $\texttt{Ignore-pref}$ in any valid plan, by definition of cost function c, the cost of every valid plan is either 0 or 1. The plans with cost equal to 0 are the best plans.

Theorem 8 *(Plan validity and equivalence). Let Π be a solvable planning problem with PESs and TESs, and Π' a planning problem with action costs derived from Π by the translating scheme defined above. Then, (1) there exists a valid plan π' for Π'; and (2) for every plan π' solving Π', the plan obtained by removing the actions in $A_T \cup A_P$ from π' and precondition $\texttt{normal-mode}$ from every action in π' is a valid plan for Π.*

Proof. See Appendix B. □

[5] For the sake of simplicity, in the compilation we use actions with negative preconditions. They can be easily translated into actions with only positive preconditions [63], although ruling out them can make the specification of the world state considerably larger.

Theorem 9 *(Plan preference). Let Π be a planning problem with PESs and TESs that has a solution plan ending in a PES, and Π' a planning problem with action costs obtained from Π by our translating scheme presented above. Then, (1) there exists a plan π' solving Π' such that $c(\pi') = 0$, and (2) for every plan π' solving Π' such that $c(\pi') = 0$, the plan obtained by removing the actions in $A_T \cup A_P$ from π' and precondition* `normal-mode` *from every action in π' is a valid plan solving Π and ending in a PES of Π.*

Proof. See Appendix B. □

Finally, note that there is a difference between the classical definition of planning problems with action costs and ours. In our context, the achievement goal is an arbitrary boolean formula, instead of a conjunction of atomic propositions, and our definition also includes the maintenance goal. Maintenance goals can be compiled away as described in the next section. The formula representing the problem achievement goal can be compiled away into action preconditions (the goal formula of the compiled problem is a conjunction of propositions) [63]. Specifically, first the goal formula is transformed into a DNF formula. Then, for every disjunct δ of the (DNF) achievement goal formula, the set of actions is augmented by an action with a dummy effect and the set of conjuncts of δ as the precondition set of the action. The dummy effect is a conjunct of the new problem goal formula and the original goal formula is removed.

5.3. Encoding maintenance goals into action preconditions

It is known that planning problems with maintenance goals can be translated into propositional planning problems [6,19,40,43]. A very simple translation for the planning problem associated with a program transition having a maintenance goal consists in adding the maintenance goal formula of the transition to the precondition formula of every domain action. The negative side of such a translation is that many planners transform the precondition and goal formulas into disjunctive normal form before planning, and thus the transformation of the formulas obtained by ruling out maintenance goals may blow up. Investigating efficient encodings of maintenance goals is out of the scope of this paper. For our experimental analysis, we considered only planning programs with goal formulas stated as conjunctions, which can be normalized without a blowup of the resulting compiled problem.

It is important to note that the end $\mathcal{D}$-state $last(\pi)$ of any plan π realizing an incoming transition of a $\mathcal{P}$-state v is the initial $\mathcal{D}$-state of any plan realizing a transition outgoing from v. If the computation of π ignored the interdependency between π and the plans realizing the outgoing transitions of v, it could happen that the maintenance goal formula of an outgoing transition is not satisfied in $last(\pi)$. In this case, the planning problem derived to realize such an outgoing transition would be unsolvable, and therefore algorithm RealizePlanProg would backtrack. In order to reduce the amount of these backtracks the original *achieving* goals of the program transitions incoming to a $\mathcal{P}$-state v can be augmented as follows. Let $\{\langle v, \langle \gamma_i, \psi_i, \phi_i \rangle, v_i \rangle \mid 1 \leq i \leq m\}$ be the set of outgoing transitions of v, where γ_i, ψ_i and ϕ_i are the guard, maintenance goal, and achieving goal formula, respectively. Every incoming transition $\langle v', \langle \gamma, \psi, \phi \rangle, v \rangle$ of v is changed to

$$\langle v', \langle \gamma, \psi, \phi \wedge \bigwedge_{i=1}^{i=m} \gamma_i \rightarrow \psi_i \rangle, v \rangle.$$

As we will see in Section 6.4, indeed such a transformation can reduce significantly the amount of backtracking of RealizePlanProg, and hence considerably improve the performance of RealizePlanProg.

5.4. Enhancing the program realization by plan adaptation

Planning programs may represent routines that include cycles to carry on in the domain. In this case, computing the realization requires to reach at least one goal situation more than once. Assume that at the i-th iteration of the loop 5–25 of RealizePlanProg a transition is processed by invoking subroutine Plan with problem Π_i, and, subsequently, at the j-th iteration ($j > i$) such a transition is processed again by invoking Plan with problem Π_j. In order to solve Π_j, subroutine Plan could re-use and modify the plan previously computed for Π_i, instead of planning from scratch.

From a theoretical point of view, in the worst case, adapting an existing plan is not more efficient than a complete re-generation of the plan from scratch [77]. However, in practice, plan adaptation can be much more efficient than generating, when few changes of the existing plan are necessary to adapt it. In the context of the planning program realization, the achievement and maintenance goals of problems Π_i and Π_j are the same (i.e., the achievement and maintenance goal formulas associated with the transition processed both at the i-th and j-th iteration), and hence adapting the plan previously computed for Π_i can be extremely promising when solving Π_j.

In principle, a transition in a cycle can be processed by RealizePlanProg more than twice, and hence the number of previously computed plans that can be re-used for realizing such a transition may be greater than one. Assume that transition $d = \langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle$ has been already realized $n > 1$ times. Let Π_k be the planning problem associated to the k-th realization of d with initial state s_k ($k \in \{1, \ldots, n\}$), and π_k the solution plan computed for Π_k so that $\Omega(s_k, d) = \pi_k$. Suppose now that, at the current iteration of loop 5–25 of RealizePlanProg, transition d is processed again with an open pair $\langle s', v \rangle$ and planning problem Π'. The differences between every Π_k and Π' concern their initial states and sets of PESs and TESs.

Algorithm: BestPlan(s, d)
Input: a domain state s, and a program transition d from v to v' involved in a planning program cycle;
Output: a (possibly empty) plan.

1. $best \leftarrow \infty$; $bestplan \leftarrow \emptyset$;
2. **foreach** $s^v \in States(v)$ **do**
3. **if** $\Omega(s^v, d) \neq \texttt{noPlan}$ **do**
4. $n \leftarrow |\mathsf{RelaxedPlan}(s, s^v)|$;
5. **if** $n < best$ **then**
6. $best \leftarrow n$;
7. $bestplan \leftarrow \Omega(s^v, d)$;
8. **return** *bestplan*.

Fig. 3. Algorithm for selecting the best plan to re-use for processing a transition d in a planning program cycle. RelaxedPlan is a plan reaching s^v from s constructed using the domain action without their negative effects.

However, for $k = 1 \ldots n$, $last(\pi_k)$ is still in $States(v')$, and hence it is a PES of Π'. (If $last(\pi_k)$ were not in $States(v')$, then $\Omega(s_k, d)$ would not be π_k.) In this case, a plan π' solving Π' may be constructed as a sequence of two subplans: a subplan reaching s_k from s' (if exists), followed by π_k for some $k \in \{1, \ldots, n\}$. The expected cost required to adapt plan π_k to solve Π' can be estimated as the number of actions in the relaxed plan constructed to achieve states s_k from s' [45,57]. The plan is relaxed, as it is constructed by ignoring the domain actions' negative effects.

Fig. 3 shows an algorithm, called BestPlan, for selecting the best plan to re-use for a doman state s and a transition d from v to v' involved in a cycle. For each state s^v in $States(v)$ such that a plan π realizing d from s^v has been already computed, BestPlan generates a relaxed plan π_R to reach s^v from s (lines 2–4); and, finally, BestPlan returns the plan with the expected lowest adaptation cost (lines 5–8). The selected best plan can then be re-used by algorithm RealizePlanProg invoking a different version of subroutine Plan, that we call AdaptPlan, with additional input the plan π_{best} returned by BestPlan(s, d). If π_{best} is equal to $\emptyset$, AdaptPlan plans from scratch; otherwise, it adapts π_{best} to a valid plan that realizes transition d from domain state s with a preferred end state in $States(v')$.

6. Experimental results

We present here the results of an experimental study with the following main goals:

- analyzing the effectiveness and the efficiency of our approach to realizing agent planning programs in deterministic domains;
- evaluating the usefulness of PESs for the performance of RealizePlanProg;
- evaluating the performance of RealizePlanProg using different incorporated planners that support PDDL3 preferences for representing PESs, or that can solve the planning problem with action costs obtained by compiling them away;
- evaluating our compilation of maintenance goals in the planning problems, and the usefulness of using plan adaptation techniques for realizing the planning program transitions.

In this experimental study, we focus on achievement and maintenance goals that are conjunctive. Moreover, we set all the transition guards to true. Note that realizing planning programs with all guards set to true does not represent a simplification in experimenting our algorithm, since it forces the algorithm to realize all outgoing transitions, even those that guards would rule out. In other words, by considering only planning programs without guards in our experiments, we are not restricting the analysis to planning programs that are computationally easier (than those with guards) to solve for our technique.

6.1. Experimental settings

Algorithm RealizePlanProg has been tested using three well-known incorporated planners: Hplan-P [6], LAMA [89], and LPG [45]. In the following, before describing the used benchmark domains and problems, we give a very brief description of each of them. More detailed information is available from the relative referred papers. In the rest of the paper, notation RealizePlanProg[x] denotes RealizePlanProg incorporating planner x.

Hplan-P [6] is a heuristic search planner built on top of the TLPlan system [3]. Hplan-P handles PDDL3 constraints and preferences by transforming these into parameterized finite state automata. Essentially, it uses an incremental best-first search planning algorithm, guided by a prioritized sequence of heuristics, which combines estimates of the cost of reaching the goals, the cost of satisfying preferences, and different estimates of the final plan metric value. With RealizePlanProg[Hplan-P], a planning problem with PESs and TESs is encoded into a PDDL3 problem as described at the beginning of section 5.2, except that the disjunction representing the TESs is part of the Hplan-P's problem goal formula instead of a PDDL3 `at end` constraint.

LAMA [89] translates the PDDL problem specification into the multi-valued state variable representation "SAS+" [4] and searches for a plan in the space of the world states using a heuristic derived from the *causal graph* [53], a particular graph representing the causal dependencies of SAS+ variables. Its core feature is the use of a pseudo-heuristic derived

from landmarks, propositions that for every solution of a planning task must be true in some state reached by the solution. Moreover, a weighted $A^\star$ search is used with iteratively decreasing weights, so that the planner continues to search for plans of better quality. While LAMA does not support reasoning over PDDL3 preferences and constraints, it supports planning with action costs through the usage of real-valued fluents. With RealizePlanProg[LAMA], planning problems with PESs and TESs are encoded by using the translation scheme described in Section 5.2, plus the real-valued fluent "`cost`": the initial state of each problem assigns value zero to `cost`; the problem metric function requires to minimize the value of `cost`; and, finally, each action of the translated problem with cost equal to 1 is encoded with the additional PDDL effect "`(increase (cost) 1)`" increasing the value of fluent `cost` by 1 unit.

LPG [45] is based on a stochastic local search procedure that explores a space of partial plans represented through *linear action graphs* (shortly, LA-graphs) [45], which are variants of the very well-known planning graph [12]. The search steps are certain graph modifications transforming a LA-graph into another one. LPG's search algorithm selects the successor LA-graph according to a heuristic evaluation function and a "noise parameter". The heuristic function estimates the number of additional search steps required to find a solution from the graphs obtained by applying the possible modifications. The noise parameter introduces some randomization in the choice of the successor, which is useful to escape from search states corresponding to local minima. When a solution is found, the LA-graph is modified by applying some graph modifications that improve the quality of the represented plan according to the problem plan metric, and the search is restarted to reach a new solution from the resulting LA-graph. LPG is the only planner considered in our experimental analysis that supports plan adaptation, as its initial search state can be either an empty LA-graph (in planning from scratch) or the LA-graph representing an input plan (in plan adaptation). The encoding of PESs and TESs used with RealizePlanProg[LPG] is the same as in RealizePlanProg[LAMA].

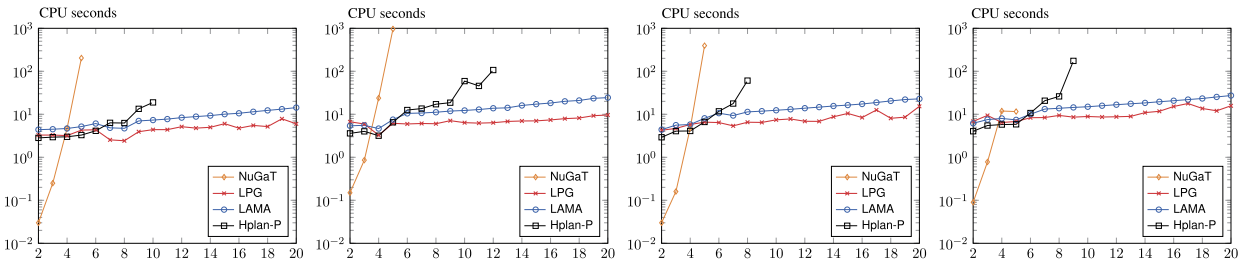
In our experimental analysis, we have also considered the realization of planning programs using NuGaT, an optimized game solver built on top of NuSMV [21], as a baseline for evaluating the performance of RealizePlanProg. It should be clear that since NuGaT is a solver more general than RealizePlanProg, it is expected that it performs worse than our proposed approach for deterministic domains. Nevertheless, we believe it is a useful baseline for evaluating the performance of RealizePlanProg.

In the experiments, planning programs are constructed over 8 benchmark domains and with 6 different program structures defined by the planning program transition relation δ. Seven of the chosen domains were also used in past international planning competitions (IPCs) [2,43,54,56,59,70,71]. They are: `Logistics` (IPC-1), `Blocksworld` (IPC-2 typed version), `Zenotravel` (IPC-3 typed STRIPS version), `Pipesworld` (IPC-4 propositional "no-tankage" version), `Storage` (IPC-5 propositional version), `Elevators` (IPC-6 sequential satisficing version without real-valued fluents), and `Barman` (IPC-7 sequential satisficing track version without real-valued fluents). All these planning domains have no deadend in their state space. To study the behavior of our approach for planning domains with deadends, we also designed and used an additional "directed" version of `Logistics` that will be described when we present the results of this experiment. The considered planning program structures are: a single cycle with only achievement goals (shortly, `1C`), a single cycle with both achievement and maintenance goals (shortly, `1C+M`), multiple binary cycles in sequence (`MC`), a random sparse directed graph (`RS`), and a complete directed graph (`CG`). Moreover, we consider a variant of `1C` with one cycle and one external node connected to the cycle by a single edge (shortly `1E1C`). More formally, these structures are defined as follows.

- `1C`$[n]$: $\delta = \{\langle v_i, G_i, v_{((i \bmod n)+1)}\rangle \mid v_i \in V,\ 1 \le i \le n\}$;
- `1E1C`$[n]$: $\delta = \langle v_1, G_1, v_2\rangle \cup \{\langle v_{i+1}, G_i, v_{((i \bmod (n-1))+2)}\rangle \mid v_i \in V,\ 1 \le i < n\}$;
- `1C+M`$[n]$: $\delta = \{\langle v_i, \langle G_i, M_i\rangle, v_{(i \bmod n)+1)}\rangle \mid v_i \in V,\ 1 \le i \le n\}$;
- `MC`$[n]$: $\delta = \{\langle v_i, G_i, v_{i+1}\rangle, \langle v_{i+1}, G_{i+n-1}, v_i\rangle \mid v_i \in V, 1 \le i < n\}$;
- `RS`$[n]$: $\delta = \{\langle v_i, G_i, w_i\rangle \mid (v_i, w_i) \in E_{Rand}, 1 \le i \le |E_{Rand}| = \lceil n \cdot log_2 n\rceil\}$;
- `CG`$[n]$: $\delta = \{\langle v_i, G_{i\cdot n+j}, v_j\rangle, \langle v_j, G_{j\cdot n+i}, v_i\rangle \mid v_i, v_j \in V,\ 1 \le i \le n,\ 1 \le j \le n,\ i \neq j\}$;

where V is the set of program states, $n = |V|$, E_{Rand} is a set of $\lceil n \cdot \log_2 n\rceil$ randomly selected pairs of program states, M_i denotes the i-th set of maintenance goals, and G_x denotes the x-th set of (randomly generated) achievement goals. Unless differently specified, the sets of achievement goals were obtained by using the existing problem generators. Maintenance goals were hand coded because there exists no automatic generator for them, and developing a tool to generate them which guarantees that the obtained problems are solvable is not trivial. Overall, we constructed 1223 planning programs with a randomly generated initial state and $|\delta|$ problem goal sets. Specifically, we constructed the following five benchmarks:

- SM_6. For `Blocksworld`, 80 planning programs with the domain size ranging from small to middle-size (the domain involves from 2 to 21 blocks) and program transition relation yielding structures `1C`[6], `MC`[4], `RS`[4], and `CG`[3] ($|\delta| = 6$);
- SM_{50}. For each considered domain, 80 planning programs with the domain size ranging from small to middle-size (the domain involves from 3 to 30 objects) and program transition relation yielding structures `1C`[50], `MC`[26], `RS`[14], and `CG`[8] ($|\delta| \approx 50$);
- $SM{+}M_{50}$. For domains `Logistics` and `Storage`, 40 planning programs obtained by the programs of benchmark SM_{50} by adding maintenance goals to the program transitions;



(a) 1C[6]. (b) RS[4]. (c) MC[4]. (d) CG[3].

Fig. 4. CPU time of RealizePlanProg using LPG, LAMA, Hplan-P and NuGaT for planning programs with domain Blocksworld and δ equal to 1C[6], MC[4], RS[4] and CG[3] (s.t. $|\delta| = 6$). The x-axis refers to the number of blocks in the planning domain.

ML_{2-12}. For domains Blocksworld and Zenotravel, 33 planning programs with the domain size ranging from middle-size to large (the domain involves from 40 to 76 objects) and program transition relation yielding structures CG[2-4] ($|\delta|$ ranges from 2 to 12);

S_{5-100}. For each considered domain, 67 planning programs with the same small domain size (the number of domain objects ranges from 2 to 18) and program transition relation yielding structures 1C[5-100], MC[4-51], RS[3-23], and CG[3-11] ($|\delta|$ ranges from about 5 to 100).

The considered evaluation criteria are the CPU time used to realize the planning program and the program realization size (number of generated plans in the computed realization). The latter measures the quality of the realization: the lower the program realization size is, the simpler and, we believe, more desirable the realization is. An alternative criterion for measuring the realization quality can be the amount of resources used or produced by the plans forming the program realization (e.g., fuel, money, time, space, etc.). However, in the analysis we did not consider this, since the paper is focused on planning programs where the domain states are sets of propositions, which are unsuitable to effectively encode amounts of resources.

The tests were conducted on an Intel Xeon(tm) 3 GHz machine, with 2 Gbytes of RAM. Unless otherwise indicated, the CPU-time limit used by RealizePlanProg to realize planning programs was 1000 seconds. The termination of the incorporated planner was forced after 60 seconds or when two different solution plans (with increasing quality) were computed. Note that in this latter case, the second plan necessarily achieves a PES. Moreover, the second plan computed by every planner incorporated into RealizePlanProg is an optimal solution (in terms of satisfied PESs). This is because (i) Hplan-P maximizes the number of achieved PESs and at most one PES can be reached; (ii) LAMA and LPG minimize the total cost of the plan solving the problem obtained by compiling PESs and TESs away, and, by construction of the compiled problems, at most one action with positive cost can be executed in a valid plan (the cost of every other action is equal to zero).

6.2. *Performance of* RealizePlanProg *with different planners*

In this section, we experimentally evaluate the performance of RealizePlanProg with planners Hplan-P, LAMA and LPG using benchmarks SM_6, SM_{50} and S_{5-100}.

Fig. 4 shows the CPU time of RealizePlanProg and NuGaT (our baseline) for domain Blocksworld in benchmark SM_6. As expected, the gap between the performance of RealizePlanProg using any incorporated planner and NuGaT is huge, since NuGaT can realize Blocksworld planning program with only very few blocks within the CPU-time threshold. We think that the (not surprising) poor performance of NuGaT is merely due to the lack of heuristic-based search techniques (for plan construction) in this general purpose reasoning system.

Moreover, the results in Fig. 4 indicate that RealizePlanProg using either LAMA or LPG realizes all the planning programs, while using Hplan-P it realizes only the planning programs with small domain instances. For these planning programs, the CPU times of RealizePlanProg using LPG, LAMA and Hplan-P are similar, but for planning programs with larger domain instances (number of blocks) the use of LPG or LAMA makes realizing the programs at least 1–2 orders of magnitude faster.

Fig. 5 shows the program realization size of RealizePlanProg for SM_6. The size of the program realization computed by RealizePlanProg using LAMA is always the best; the program realization size of RealizePlanProg[LPG] is slightly larger than or equal to RealizePlanProg[LAMA]; finally, for the planning programs with small domain instances, the program realization size of RealizePlanProg[Hplan-P] and of RealizePlanProg using either LAMA or LPG are the same, but for the other planning programs RealizePlanProg[Hplan-P] computes much larger realizations.

The results in Fig. 5 indicate that, for large planning domain instances, the plans computed by Hplan-P do not usually achieve PESs. Figs. 4 and 5 also show that the larger the program realization is, the slower RealizePlanProg is. This is because for the considered planning programs the number of open pairs generated by RealizePlanProg is usually similar to the program realization size, and the incorporated planner is run at least once for every open pair.

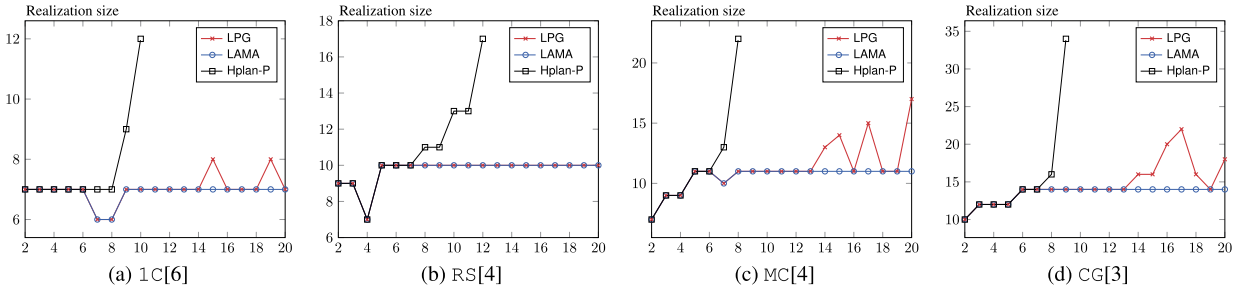


Fig. 5. Realization size of RealizePlanProg using LPG, LAMA and Hplan-P for planning programs with domain `Blocksworld` and δ equal to `1C`[6], `MC`[4], `RS`[4] and `CG`[3] (s.t. $|\delta| = 6$). The x-axis refers to the number of blocks in the planning domain.

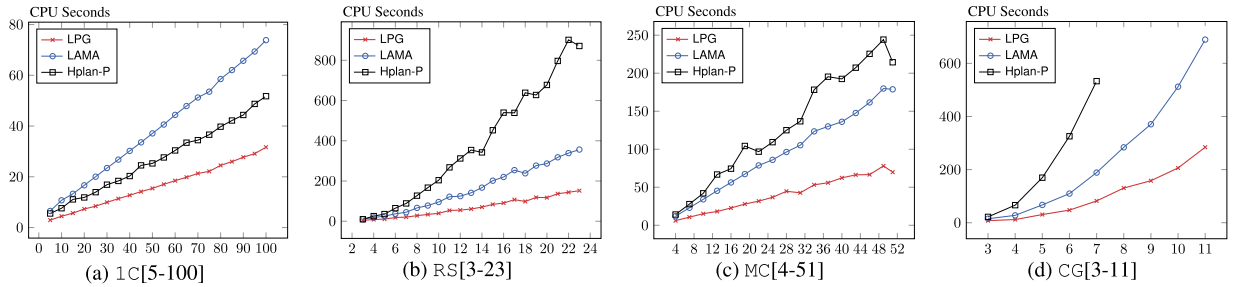


Fig. 6. CPU time of RealizePlanProg using LPG, LAMA, and Hplan-P for planning programs with domain `Zenotravel` and δ over four different values and $|\delta|$ ranges from about 5 to 100. The *x*-axis refers to the number of program states.

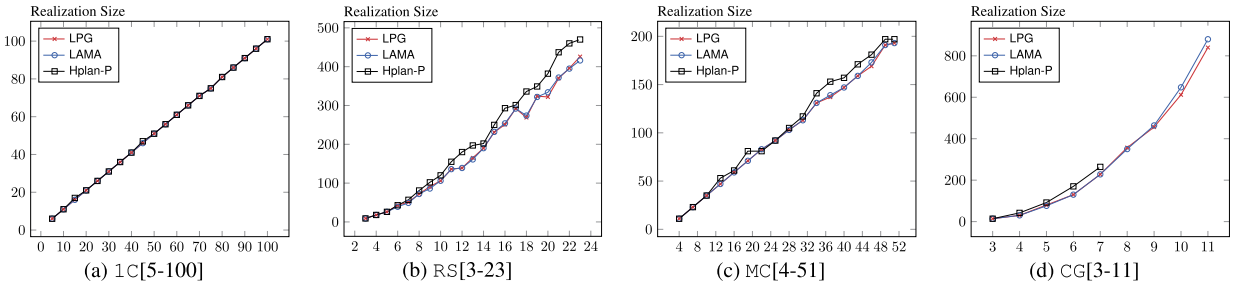


Fig. 7. Realization size of RealizePlanProg using LPG, LAMA, and Hplan-P for planning programs with domain `Zenotravel` and δ over four different values and $|\delta|$ ranges from about 5 to 100. The *x*-axis refers to the number of program states.

For domains different from `Blocksworld` and program transition relations larger than those in SM_6, we obtained similar results. Appendix A shows the performance of RealizePlanProg for domains `Logistics` and `Pipesworld` with benchmark SM_{50}. The appendix gives no result for NuGaT, because it realizes no planning program of this benchmark.

Figs. 6 and 7 compare the CPU time and the program realization size of RealizePlanProg using Hplan-P, LAMA and LPG for domain `Zenotravel` with benchmark S_{5-100}. For all planning programs of S_{5-100} with domain `Zenotravel`, the number of involved domain objects is the same and equal to 11. These results indicate that for program structures `1C`, `MC`, and `RS`, both the CPU time and the program realization size grow roughly linearly w.r.t. the number of program states; for structure `CG`, they grow quadratically. Therefore, for the experimented program structures, the performance grows linearly w.r.t. the size of the program transition relation. We experimentally observed that for planning programs with domain `Zenotravel`, the average total number of open pairs generated by every considered incorporated planners is: about $1 \cdot |\delta|$ if the planning-program structure is `1C`; about $1.5 \cdot |\delta|$ if the planning-program structure is `MC`; about $2 \cdot |\delta|$ if the planning-program structure is `RS`; and, finally, about $2.5 \cdot |\delta|$ if the planning-program structure is `CG`.

Appendix B shows the results of this analysis for two other domains of benchmark S_{5-100}: `Elevators` and `Storage`. The results are similar, except for Hplan-P, which fails to realize many planning programs. The rationale for this behavior is that, with domains `Elevators` and `Storage`, even for domain instances involving few objects, the size of the domain state can be large, and consequently achieving PESs can be very hard.

Table 2
Average CPU time and number of realized planning programs (in parenthesis) of RealizePlanProg using Hplan-P, LAMA and LPG with/out PESs for planning programs of benchmark SM_{50}. Gray boxes indicate very significant performance gap.

Domain	using Hplan-P		using LAMA		using LPG	
	with PESs	w/out PESs	with PESs	w/out PESs	with PESs	w/out PESs
Barman						
1C[50]	76.92 (18)	94.64 (18)	56.88 (20)	109.76 (20)	43.76 (20)	67.29 (20)
MC[26]	467.91 (9)	305.67 (13)	151.44 (20)	511.38 (14)	231.19 (12)	606.46 (7)
RS[14]	370.39 (5)	258.13 (6)	338.46 (16)	760.89 (5)	310.65 (7)	886.56 (2)
CG[8]	393.87 (5)	177.05 (5)	395.99 (10)	678.94 (5)	435.21 (4)	940.81 (1)
Total	277.60 (37)	193.13 (42)	205.18 (66)	475.55 (44)	175.93 (43)	432.38 (30)
Blocksworld						
1C[50]	30.17 (4)	60.99 (4)	63.53 (20)	99.89 (20)	24.46 (20)	161.68 (19)
MC[26]	116.95 (3)	586.70 (2)	138.02 (20)	459.41 (20)	74.32 (20)	945.02 (2)
RS[14]	180.45 (8)	731.15 (4)	165.82 (20)	913.42 (8)	101.18 (20)	871.73 (4)
CG[8]	420.55 (8)	865.03 (2)	290.52 (20)	960.11 (2)	162.76 (20)	928.75 (2)
Total	229.54 (23)	642.33 (12)	164.47 (80)	608.21 (50)	90.68 (80)	726.80 (27)
Elevators						
1C[50]	34.68 (20)	66.97 (20)	43.52 (20)	82.81 (20)	38.01 (20)	60.24 (20)
MC[26]	354.82 (8)	562.88 (10)	102.24 (20)	813.36 (10)	102.64 (20)	766.48 (10)
RS[14]	286.26 (1)	773.37 (1)	204.70 (20)	1000 (0)	300.74 (11)	961.66 (1)
CG[8]	606.18 (3)	1000 (0)	364.72 (20)	1000 (0)	501.28 (5)	1000 (0)
Total	186.67 (32)	315.93 (31)	178.80 (80)	724.04 (30)	154.06 (56)	573.44 (31)
Logistics						
1C[50]	72.27 (20)	113.46 (20)	33.78 (20)	80.68 (20)	13.16 (20)	41.95 (20)
MC[26]	177.88 (5)	936.96 (2)	77.90 (20)	899.85 (4)	44.91 (20)	828.03 (7)
RS[14]	292.62 (2)	1000 (0)	162.59 (20)	1000 (0)	78.15 (20)	1000 (0)
CG[8]	557.73 (2)	1000 (0)	277.23 (19)	1000 (0)	182.36 (20)	1000 (0)
Total	139.15 (29)	377.72 (22)	136.11 (79)	741.90 (24)	79.65 (80)	717.49 (27)
Pipesworld						
1C[50]	126.55 (20)	168.05 (20)	69.63 (20)	120.97 (20)	14.70 (20)	65.86 (20)
MC[26]	393.92 (7)	675.43 (4)	164.54 (20)	930.32 (4)	53.56 (20)	867.32 (4)
RS[14]	– (0)	– (0)	350.47 (20)	1000 (0)	111.34 (20)	1000 (0)
CG[8]	– (0)	– (0)	629.40 (19)	1000 (0)	256.76 (20)	1000 (0)
Total	195.87 (27)	299.59 (24)	299.38 (79)	759.82 (24)	109.09 (80)	733.29 (24)
Storage						
1C[50]	139.50 (2)	196.47 (2)	105.25 (18)	273.28 (16)	19.02 (20)	101.86 (20)
MC[26]	– (0)	– (0)	156.26 (17)	894.51 (4)	59.95 (20)	1000 (0)
RS[14]	732.76 (1)	292.12 (3)	164.13 (20)	909.93 (6)	83.13 (19)	1000 (0)
CG[8]	1000 (0)	516.40 (1)	278.37 (20)	951.80 (4)	114.58 (17)	1000 (0)
Total	579.55 (3)	297.62 (6)	178.68 (75)	764.81 (30)	67.20 (76)	763.64 (20)
Zenotravel						
1C[50]	190.72 (11)	349.73 (8)	95.95 (20)	151.06 (20)	16.43 (20)	50.62 (20)
MC[26]	79.70 (2)	453.51 (2)	209.82 (20)	930.13 (2)	120.24 (20)	902.39 (4)
RS[14]	276.12 (1)	1000 (0)	194.13 (20)	1000 (0)	86.91 (20)	1000 (0)
CG[8]	856.03 (1)	1000 (0)	313.72 (20)	1000 (0)	186.70 (18)	1000 (0)
Total	225.97 (15)	450.27 (10)	203.40 (80)	770.30 (22)	100.41 (78)	731.54 (24)

6.3. Importance of using preferred end states

In order to evaluate the impact of using PESs on the performance of RealizePlanProg, we compared RealizePlanProg using PESs and ignoring them. Table 2 gives the number of realized planning programs and the average CPU time for the planning programs of benchmark SM_{50}. The average CPU time is computed using the CPU-time limit (1000 seconds) for the planning programs that RealizePlanProg does not realize (within the CPU time limit); the average realization size is computed over the planning programs that RealizePlanProg can solve both with and without using PESs.

The results in Table 2 show that planning with PESs has a high positive impact on the number of realized planning programs and the average speed of RealizePlanProg. Using either LAMA or LPG, planning with PESs always allows RealizePlanProg to realize a larger set of planning programs, and makes it (on average) faster than when planning without

Table 3
Average realization size and percentage of computed plans reaching PESs (in parenthesis) of RealizePlanProg using Hplan-P, LAMA and LPG with/out PESs for planning programs of benchmark SM_{50}. Gray boxes indicate very significant performance gap. "–" means that there is no data to compute the average.

Domain	using Hplan-P		using LAMA		using LPG	
	with PESs	w/out PESs	with PESs	w/out PESs	with PESs	w/out PESs
Barman						
1C[50]	56.3 (26.9)	56.9 (27.5)	51.1 (48.3)	62.1 (16.6)	53.5 (38.8)	60.2 (23.1)
MC[26]	126.3 (64.5)	164.7 (63.5)	87.1 (72.4)	179.7 (62.7)	92.0 (68.4)	310.4 (54.3)
RS[14]	128.0 (80.5)	134.4 (80.2)	116.8 (82.5)	165.6 (80.4)	120.5 (82.1)	435.5 (75.7)
CG[8]	175.0 (89.9)	198.8 (89.5)	168.0 (90.1)	275.8 (88.4)	252.0 (88.6)	609.0 (86.9)
Total	101.4 (51.8)	112.8 (51.7)	83.3 (64.6)	135.6 (46.7)	73.5 (50.3)	161.9 (36.0)
Blocksworld						
1C[50]	51.0 (50.0)	52.3 (33.3)	51.0 (50.0)	51.9 (37.2)	51.1 (49.1)	158.9 (11.0)
MC[26]	91.5 (68.6)	467.0 (52.7)	98.3 (66.4)	278.6 (54.4)	91.5 (68.7)	949.0 (50.8)
RS[14]	143.8 (81.2)	570.8 (76.1)	166.5 (80.1)	660.9 (75.8)	143.5 (81.4)	727.3 (75.8)
CG[8]	238.0 (88.7)	623.0 (86.8)	238.0 (88.7)	623.0 (86.8)	238.0 (88.7)	623.0 (86.8)
Total	119.8 (70.0)	389.3 (59.7)	95.9 (63.0)	262.9 (52.3)	81.6 (58.4)	336.0 (29.2)
Elevators						
1C[50]	52.5 (40.9)	55.2 (19.6)	50.9 (55.0)	54.5 (22.5)	50.9 (55.0)	59.0 (16.4)
MC[26]	98.4 (66.1)	588.3 (51.7)	94.6 (67.4)	500.1 (52.0)	97.1 (66.8)	787.8 (51.2)
RS[14]	140.0 (79.5)	1094 (72.3)	– (–)	– (–)	151.0 (80.4)	1140 (74.2)
CG[8]	– (–)	– (–)	– (–)	– (–)	– (–)	– (–)
Total	68.2 (49.2)	238.0 (30.3)	65.5 (59.1)	203.0 (32.3)	69.0 (59.6)	328.9 (29.5)
Logistics						
1C[50]	54.7 (32.4)	65.5 (8.4)	50.8 (60.0)	61.0 (10.4)	50.8 (62.5)	61.4 (10.4)
MC[26]	84.0 (71.2)	1035 (50.7)	84.3 (71.2)	474.3 (51.9)	84.4 (70.9)	991.1 (51.4)
RS[14]	– (–)	– (–)	– (–)	– (–)	– (–)	– (–)
CG[8]	– (–)	– (–)	– (–)	– (–)	– (–)	– (–)
Total	57.3 (36.0)	153.6 (12.2)	56.4 (61.9)	129.9 (17.3)	59.5 (64.7)	302.4 (21.0)
Pipesworld						
1C[50]	54.9 (26.8)	58.8 (13.4)	51.1 (48.3)	60.0 (12.6)	51.0 (50.0)	97.1 (8.5)
MC[26]	102.0 (65.6)	470.5 (51.9)	98.0 (66.4)	496.3 (51.7)	98.5 (66.3)	654.0 (51.2)
RS[14]	– (–)	– (–)	– (–)	– (–)	– (–)	– (–)
CG[8]	– (–)	– (–)	– (–)	– (–)	– (–)	– (–)
Total	62.8 (33.2)	127.4 (19.8)	58.9 (51.4)	132.7 (19.1)	58.9 (52.7)	189.9 (15.6)
Storage						
1C[50]	52.0 (33.3)	54.5 (19.6)	51.0 (50.0)	83.1 (5.9)	52.1 (41.9)	138.2 (2.5)
MC[26]	– (–)	– (–)	70.0 (77.3)	366.3 (52.6)	– (–)	– (–)
RS[14]	95.0 (82.9)	457.0 (74.3)	109.0 (83.2)	509.8 (75.9)	– (–)	– (–)
CG[8]	– (–)	– (–)	171.5 (89.9)	607.3 (86.9)	– (–)	– (–)
Total	66.3 (49.9)	188.7 (37.9)	81.2 (65.6)	276.1 (36.9)	52.1 (41.9)	138.2 (2.5)
Zenotravel						
1C[50]	72.4 (28.4)	90.4 (5.7)	51.0 (52.5)	63.2 (11.8)	51.1 (50.0)	69.3 (11.4)
MC[26]	90.0 (68.5)	598.0 (51.6)	86.0 (69.7)	267.5 (54.3)	87.8 (69.3)	988.8 (51.8)
RS[14]	– (–)	– (–)	– (–)	– (–)	– (–)	– (–)
CG[8]	– (–)	– (–)	– (–)	– (–)	– (–)	– (–)
Total	75.9 (36.4)	191.9 (14.8)	54.1 (54.0)	81.7 (15.6)	57.2 (53.2)	222.5 (18.1)

PESs. Interestingly, very often the algorithm realizes at least two times more planning programs, or is at least one order of magnitude faster (see gray boxes in Table 2). The performance gap is very large especially for planning programs with structures involving several cycles. Concerning RealizePlanProg[Hplan-P], planning with PESs often gives better performance than planning without them, but in some cases we observed a performance decrease. This happened for domain Barman and δ equal to MC[26], RS[14], or CG[8], domain Elevators and δ equal to MC[26], and domain Storage and δ equal to RS[14], or CG[8]. In these cases, Hplan-P often crashes when it attempts to solve planning problems with many preferences or with preferences involving many propositions.

Table 3 analyzes the program realization size (i.e., the total number of plans in the computed program realization) for the planning programs of benchmark SM_{50}. The results in this table indicate that planning with PESs is useful also in terms of the realization size. For any considered incorporated planner, exploiting planning with PESs allows RealizePlanProg to compute program realizations that are always smaller, and often at least two times smaller (see gray boxes in Table 3). Specifically, for every considered program structure involving several cycles, the performance gap obtained by planning

Table 4
Maximum number of objects in a planning problem, and maximum size of sets S_P, A_P, S_T and A_T for RealizePlanProg using Hplan-P, LAMA and LPG when solving the planning programs of benchmark SM_{50}.

Domain	#objects	using Hplan-P		using LAMA		using LPG	
		S_P	S_T (A_T)	S_P (A_P)	S_T (A_T)	S_P (A_P)	S_T (A_T)
Barman	24	10	31 (2542)	42 (43)	0 (0)	60 (61)	0 (0)
Blocksworld	17	20	28 (3640)	10 (11)	0 (0)	10 (11)	0 (0)
Elevators	17	8	58 (4640)	12 (13)	0 (0)	65 (66)	0 (0)
Logistics	19	10	71 (3976)	28 (29)	0 (0)	46 (47)	0 (0)
Pipesworld	42	9	29 (2001)	12 (13)	0 (0)	19 (20)	0 (0)
Storage	30	6	40 (2760)	8 (9)	18 (2268)	81 (82)	0 (0)
Zenotravel	22	24	28 (1428)	10 (11)	0 (0)	50 (51)	0 (0)

with/out PESs is almost always very large, except in domain Barman if the realization algorithm uses planners Hplan-P or LAMA. Using LPG, sometimes PESs are useful even when the program structure forms a single cycle. We think that in RealizePlanProg[LPG] PESs are very useful because of the randomization in the local search procedure of LPG: in LPG the choice of the actions for the plan under construction is randomized, and this can lead to generate different plans for the same problem goals, resulting in different plan end states; however, using PESs in LPG guides the search towards the same end states (the preferred ones), ameliorating the diversification determined by the randomization.

The data in Table 4 describes the behavior of RealizePlanProg in terms of: the maximum size of the sets S_P of PESs and S_T of TESs generated for a $\mathcal{P}$-state, and the maximum number of actions in sets A_P and A_T for benchmark SM_{50} (A_P for Hplan-P in not considered in the table, because with Hplan-P PESs are encoded as PDDL3 preferences). Sets A_P and A_T, defined in Section 5.2, are used for translating a planning problem with PESs and TESs into a planning problem with action costs; they have size $|S_P| + 1$ and $|S_T| \cdot |P|$, respectively, where $|P|$ is the set of problem fluents. While in principle the size of these sets can be exponential in the number of problem objects, the results in the table show that this is not the case for benchmark SM_{50}.

As for sets S_T and A_T, since the planning programs of benchmark SM_{50} have planning domains with no deadend, the program transitions are, in principle, realizable from any (reachable) $\mathcal{D}$-state, and so the sizes of S_T and A_T can be 0. On the contrary, Table 4 shows that often this is not the case: when using Hplan-P some states are added to S_T and A_T for every considered domains, and when using LAMA these sets have size greater than zero for domain Storage. This happens because sometimes Hplan-P and LAMA fail to solve (solvable) planning problems within the given CPU-time limit and amount of memory (each failure generates a tabu state).

It is worth noting that for planning program structures including loops, even when sets S_T and A_T are empty, the planning problems with PESs that are solved during the execution of RealizePlanProg are *inter*dependent in the sense that the solution of a planning problem associated with a transition incoming to a $\mathcal{P}$-state v takes into account the solution of the planning problems associated with the transition(s) outgoing from v, as it should (preferably) enable the reuse of the plans already computed for the outgoing transitions (this is the purpose of PESs). The number of *not* interdependent planning problems is always (at most) the number of program states, i.e., the number of planning problems with an empty set of PESs that are constructed during the execution of RealizePlanProg. Therefore, the average number of solved interdependent planning problems can be derived by subtracting the number of program states from the data in Table 3. For instance, with δ equal to CG[8], the number of program states is 8. For such δ and domain Barman, the average size of the planning program realization generated using LPG and preferred end states is about 252, and hence the number of generated interdependent planning problems is, on average, (at least) $252 - 8 = 244$. The results in Table 3 show that, except for programs with δ equal to 1C[50], most of the planning problems solved by RealizePlanProg are interdependent.

6.4. Planning programs with maintenance goals

The experimental analysis presented so far uses benchmarks formed by planning programs with only achievement goals. In this section, we also consider maintenance goals using benchmark $SM{+}M_{50}$, i.e., planning programs with δ equal to 1C+M[50] and domains Logistics and Storage. For Logistics, we designed program transitions with maintenance goals constraining all airplanes but one to stay at a particular airport (the headquarter of their airline), and forcing each of the airplanes to be used in turn (for the different transitions). Similarly, for Storage all hoists but one are constrained to stay at a particular location, and each of them is forced to be used in turn. In these programs, the transitions from any $\mathcal{D}$-state in which the maintenance goal formula is not satisfied are unrealizable. Moreover, having maintenance goals in a planning problem associated with a program transition can make solving it harder for a planner.

Since the planners used in our experimental analysis do not natively support maintenance goals, planning programs with maintenance goals have been translated into planning programs without them. We considered two related translation schemes:

T1: the basic schema adding the maintenance goal formula to the precondition formula of every domain action; and
T2: the same schema T1 extended as described in Section 5.3.

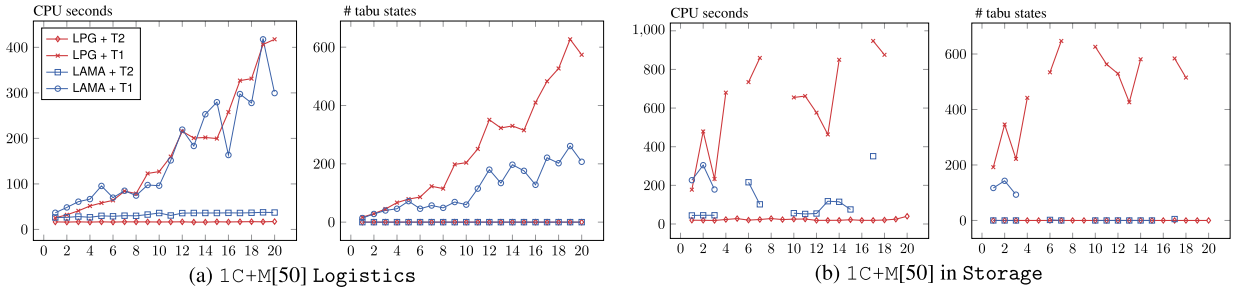


Fig. 8. CPU time and number of generated tabu states of RealizePlanProg using LPG and LAMA with/out achieving the next maintenance goals for planning programs with domains `Logistics` and `Storage` and δ equal to 1C+M[50]. The x-axis refers to the program number (the greater the number is, the greater the size of the planning domain is). The legend from the first chart applies in all four charts.

By using T2, the plans realizing the incoming transitions of v generate end states satisfying the formulas of all maintenance goals on the transitions outgoing from v.

Fig. 8 shows the performance of our approach using LAMA and LPG with the two considered translations for planning programs of benchmark SM+M_{50}. The results show that, with T1, the performance of RealizePlanProg decreases exponentially with the size of the planning programs; on the contrary, with T2 the performance does not degrade significantly, indicating that building plans achieving the maintenance goals on the next transitions is extremely useful. We observed that the performance gap with T1 and T2 using Hplan-P is even grater than when using LAMA and LPG. (These performance results using Hplan-P are omitted from Fig. 8 for the sake of its readability.)

The number of tabu states generated by RealizePlanProg with T2 is always zero using LPG and LAMA, except for only two problems (16 and 17) of `Storage` using LAMA, where LAMA exceeds the given CPU-time limit; on the contrary, using LPG and LAMA with T1, the number of generated tabu states is almost always very high. This happens because with T1 the plan computed to realize a transition incoming to a $\mathcal{P}$-state v usually reaches an end state that does not satisfy all the formulae of the maintenance goals on the outgoing transitions of v, making realizing at least one of such transitions impossible.

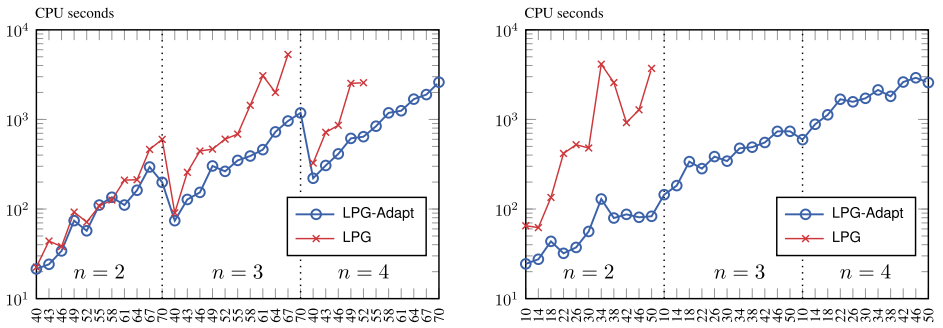
6.5. Usefulness of using plan adaptation techniques

In order to show that using plan adaptation techniques can be very useful to compute the program realization, we compared RealizePlanProg with and without using plan adaptation techniques. For this experiment, we considered the well-known domains `Blocksworld` and `Zenotravel`. Since plan adaptation can be especially useful when the program structure forms several cycles and the domain instance is large (i.e., when solving the planning problems can be quite hard), for this experiment we considered the planning programs in benchmark ML_{2-12}, which are planning programs with a structure forming a complete directed graph. The planning programs of benchmark ML_{2-12} have a number of program states ranging from 2 (program transition relation δ forms a single cycle) to 4 (δ forms 20 cycles). For the planning programs over domain `Blocksworld`, the number of blocks in the domain instances ranges from 40 to 70; for the planning programs over domain `Zenotravel`, concerning moving people in a network of locations by using aircrafts consuming levels of fuel, the number of aircrafts, cities and fuel levels is 5, 25 and 4, respectively, while the number of persons ranges from 10 to 50.

In this experiment, we used only planner LPG, since it is the only considered incorporated planner that supports plan adaptation. The CPU-time limit used by RealizePlanProg to realize a planning program was 2 hours, while the CPU-time limit for solving a planning problem by LPG was 10 minutes. In the following, LPG-Adapt denotes the version of LPG adapting the plan returned by procedure BestPlan described in Section 5.4.

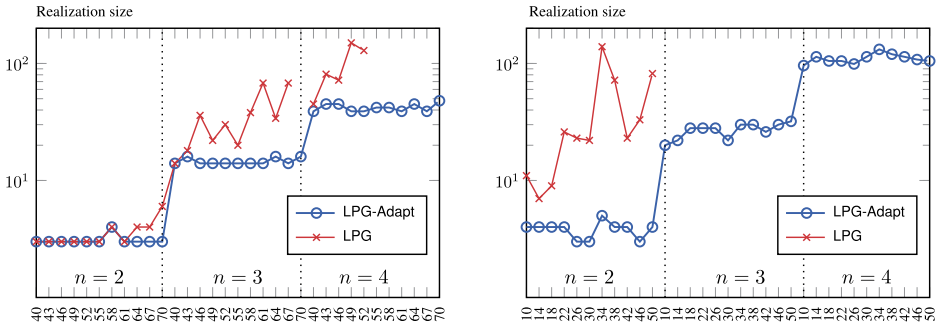
Fig. 9 shows the CPU time of RealizePlanProg using LPG and LPG-Adapt for planning programs in benchmark ML_{2-12}. RealizePlanProg[LPG-Adapt] realizes a larger set of planning programs, and is always faster than or similar to RealizePlanProg[LPG]: with LPG-Adapt every considered planning program is realized; without plan adaptation, when the planning programs have more than 2 states, for `Blocksworld` RealizePlanProg cannot realize the planning programs with the largest instances, while for `Zenotravel` it realizes no program. Moreover, RealizePlanProg[LPG-Adapt] is generally considerably faster than RealizePlanProg[LPG]. For `Zenotravel`, it is significantly faster even for relatively small domain instances and program structure forming a single cycle ($n = 2$ in Fig. 9).

Fig. 10 shows the program realization size of RealizePlanProg using LPG and LPG-Adapt for benchmark ML_{2-12}. These results indicate that, when using LPG-Adapt, the program realization can be much smaller. The rationale of this behavior is that achieving PESs by LPG-Adapt can be much easier than by LPG. This happens because *(i)* for benchmark ML_{2-12} the domain size is large, and hence achieving PESs can be very hard, *(ii)* for the considered domains, very often the last plan portion (completely) defines the plan end state, and often the last plan portion of the plan computed by LPG-Adapt is the same as of the input plan (because the goals are the same). Therefore, very often the end state of the plan computed by LPG-Adapt is the same as the end state of the input plan; hence, LPG-Adapt often easily generates plans ending in PESs.



(a) CG[2-4] in Blocksworld. (b) CG[2-4] in Zenotravel.

Fig. 9. CPU seconds of RealizePlanProg using LPG with/out plan adaptation for planning programs over domains Blocksworld, Zenotravel and δ equal to CG[2-4]. On the x-axis there is the number of blocks/persons involved in the domain instances. Finally, n is the number of program states.



(a) CG[2-4] in Blocksworld. (b) CG[2-4] in Zenotravel.

Fig. 10. Realization size of RealizePlanProg using LPG with/out plan adaptation for planning programs with domains Blocksworld, Zenotravel and δ equal to CG[2-4]. On the x-axis there is the number of blocks/persons involved in the domain instances. n is the number of program states.

6.6. On domains with many deadends

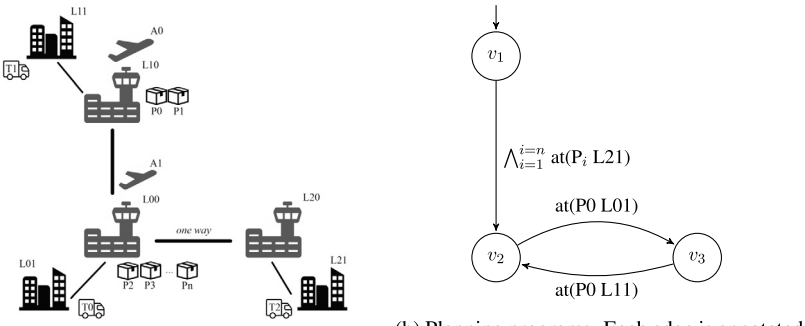
When the involved planning domain has many deadends in its domain state space, computing a realization of the planning program can be very hard also using the proposed planning-based approach. In this section, we study the performance of RealizePlanProg for domains with a large number of deadends, focusing on an interesting class of planning programs in which agent activities can be repeatedly done and undone indefinitely often.

It is worth noting that, when there are deadends, the planning programs for the experimental evaluation need to be very accurately designed in order to guarantee their realizability. For instance, consider a planning program with δ equal to 1C[50] (a single cycle) in which every transition goal requires moving an airplane in a version of domain Zenotravel without action refuel, so that the fuel level of the airplanes can never be restored after their use. Such planning programs can never be realized, even if the airplane movement were optimal (in terms of fuel consumption), because every $\mathcal{D}$-state generated by a plan realizing a transition is different from the $\mathcal{D}$-states generated by any plan previously computed for that same transition. Therefore, with an initial limited amount of fuel, there exists no realization for which the execution of cycle 1C[50] can be executed indefinitely often.

The planning programs that we designed for testing RealizePlanProg in domains with deadends use a directed (irreversible) version of domain Logistics, concerning the movement of packages among cities by airplanes and trucks, in which: only certain movements of airplanes are possible; the initial states are defined as depicted in Fig. 11a; the transition relations are modelled using program structure 1E1C; and the achieving goal formulas are defined as depicted in Fig. 11b.

The transition relation defined according to 1E1C models the agent behavior formed by a one-shot activity followed by a cyclic activity. The first activity regards the movement of all packages but one (package P0) from airports L00 and L10 to city L21; the cyclic activity regards the recurrent movement of P0 between city L01 and city L11. Airplanes can fly between airports L00 and L10 in both directions, and from L00 to airport L20 but *not* from L20 to L00. In order to realize a planning program in this class, the trick is moving all packages but P0 from L00 to L20 (and, subsequently, to L21) by using only *one* airplane. If both the airplanes were used for this movement, subsequently no airplane would be available to move P0 between L01 and L11. Let n be the number of packages to move. The number of deadend $\mathcal{D}$-state for the planning programs of this experiment is $8 \cdot 10^n$ over $72 \cdot 10^n$ possible reachable $\mathcal{D}$-states.

Fig. 12 shows the performance of RealizePlanProg for the described planning programs. With LPG and LAMA only the programs with few packages to move are realized; while with Hplan-P no program is realized. The results in the figure indicate that, when the planning domain has many deadends but the number of generated tabu states is not high,



(a) Logistics world map.

(b) Planning programs. Each edge is annotated with its corresponding achievement goal.

Fig. 11. Dynamic domain and agent planning programs with a directed version of domain `Logistics`.

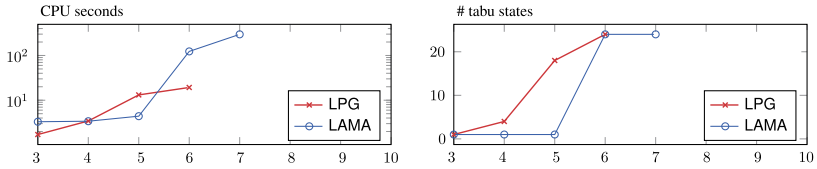


Fig. 12. CPU time and number of generated tabu states of RealizePlanProg using LPG and LAMA for planning programs with a directed version of domain `Logistics` and δ defined in Fig. 11b (`1E1C`[3-10] in `Logistics`). The x-axis refers to the number of packages in the planning domain.

RealizePlanProg can find a solution within the given CPU time limit. However, this happens only for small-size problems. When there are many packages in the domain, the number of deadends significantly increases and RealizePlanProg generates more tabu states, not only because the planning problems associated with the program transitions can be unsolvable, but also because they can be very hard to solve for the planners when they are solvable, leading the planner to fail within the given CPU time.

7. Related work

The work presented here can be related with two recent efforts to integrate agent-oriented programming and systems with declarative goals and lookahead planning. Efforts to integrate declarative goals (e.g., [24–26,58,91,100]) stem from the recognized need of providing development frameworks that are more faithful to the notion of rational agent behavior developed in agent theory [14,23,97], as well as to enhance those systems with more flexible and robust mechanisms for intelligent action selection. For example, the AgentSpeak-like language CANPlan [91] provides a construct Goal(ϕ_s, δ, ϕ_f) with the intended meaning of "achieve (success) goal ϕ_s by executing (procedural) plan δ, provided failing condition ϕ_f remains false" (similar constructs were proposed for other agent programming frameworks, such as AgentSpeak itself or 3APL/2APL). While Goal's constructs like the above one resemble planning program's transitions of the form "achieve ϕ_s while maintaining $\neg\phi_f$," they have some major differences. In particular, there is no effort from agent architectures to *proactively* enforce the satisfaction of the goals; their support remains at the *reactive* level (i.e., re-try δ if it has completed without achieving ϕ_s, and successfully drop it or abandon it with failure if ϕ_s or ϕ_f becomes true, resp.). In other words, no reasoning is performed to guarantee that plan δ is in fact executed in a way that would bring about the goal ϕ_s (while avoiding ϕ_f). The reason for this is one of efficiency: agent programs are meant to be executed online under soft real-time constraints, and hence rely on the assumption that the given program δ is designed to achieve the goal ϕ_s on-the-fly, under normal circumstances. Solving planning programs, instead, requires building plans that will not only achieve each local goal (in transitions), but that are also mutually "compatible" within the whole network of goals. On the other hand, planning programs do not provide, at this point, ways of specifying (and using) available procedural domain information to build those plans, something that can arguably help to cope with the complexity of the problem (see below discussion on HGN planning).

Another related link between planning programs and agent systems is the integration of automated planning capabilities to the latter. There are indeed a number of platforms and architectures which mix, in some way or another, planning and program execution into a so-called *continual planning* approach, such as A-SHOP [39], Retsina [78], SRI's Cypress [105], Propice-Plan [37], CANPlan [91], and JADEX [103]. All these systems are able to do some type of lookahead planning within a typical reactive agent execution. In most cases, the type of planning considered is domain-tailored planning, similar to HTN-planning [46], rather than first-principle planning as in planning programs [91,104]. In addition, the underlying approach is to provide specific programming constructs (e.g., CANPlan's Plan(δ, ϕ) [91] or IndiGolog's $\Sigma(\delta; \phi?)$ [29] constructs to achieve ϕ using program δ) that allow for calling a planning module to synthesize a course of actions, which is then carried out by the agent execution engine. Roughly speaking, the difference with our work is that the core of continual

planning systems is driven by an online executor (which can however resort to local lookahead planning as necessary), whereas planning programs are meant to be fully solved offline in order to obtain execution guarantees for all possible agent behaviors modelled in the program.

From the planning perspective, the work on hierarchical goal network (HGN) planning [95,96] shares motivations and has technical similarities with planning programs, but they also have important differences. HGN planning aims at generalizing "classical" HTN planning to include goal networks, by using a different semantics for tasks and methods. In HGN planning, tasks correspond to classical goals and methods specify ways to decompose goals into sequences of subgoals. There has even been efforts to develop HGN-planning systems that work with partial decomposition knowledge [94]. Like planning programs, HGN planning has the ability to specify agent behaviors in a declarative manner using a network of goals. However, those networks amount to partially-ordered sets of goals (therefore not admitting indefinitely *looping* behaviors) whose total order of satisfaction is left to the solver to decide. Planning programs admit network with cycles and the ordering of goals is outside the control of the solver. Generally speaking, the behaviors that HGN planning aims to capture are the same as those of planning with temporally extended goals, producing a *single* plan to be executed. On the other hand, planning programs require generating a controller for multiple alternative synthesized plans that cover the whole space of deliberation of the agent (in order to execute the right plan according to the transition chosen by the agent at each step). The idea, though, of integrating goal networks with subgoal decomposition knowledge as well as the techniques based on landmark reasoning used in existing HGN systems are worth investigating in the context of planning programs, so as to better deal with the intrinsic computational difficulty of the task.

One particular agent paradigm that appears capable of encoding planning programs is that of Golog-like situation calculus-based high-level programming languages [28,29,68,5]. Indeed, because those languages offer standard programming constructs (including iteration, conditionals, and even parallel execution) as well as non-deterministic $\delta_1 \mid \delta_2$ (execute δ_1 or δ_2) and a test construct $\phi?$ (guarantee ϕ is true), one could imagine that planning programs could be encoded into a particular Golog-like program. This is actually not the case, at least if one considers the standard semantics of these programs [68], the so called "offline execution." First, Golog-like languages are typically meant to execute the given program to completion and cannot then handle continuous (cyclic) programs/controllers that are meant to run forever, as it is the case for planning programs. Second, the non-deterministic constructs have typically an "angelic" semantic: the planner has to find *one* that works. In planning programs, the controller has to guarantee executability for every possible choice. Finally, Golog-like languages do not come with sophisticated techniques for the actual synthesis of (iterated) successful executions. A different analysis needs to be carried out for IndiGolog [29]. This variant of Golog has the capabilities of representing our planning programs, by making use of standard constructs to represent the control structure given by the transition system and the special deliberation construct $\Sigma(\cdot)$ for representing "goal-oriented actions" labeling transition. Specifically, each goal-oriented assertion $[\gamma : \psi, \phi]$ can be represented as $[\textbf{if}\ \gamma\ \textbf{then}\ \Sigma((\pi a.\psi?; a)^*; \phi?)]$. Nonetheless, due to their online execution nature, the resulting IndiGolog program would account for a sort of continual planning approach as discussed above, under which goals assertions (modeled as Σ search blocks) are independent of each other.

Interestingly, in [5], a language based on Golog has been used to specify domain-control knowledge for solving classical planning problems, and a translation function has been proposed, which given a planning instance and a program described by a Golog-based language outputs a new planning instance that embeds the control stated by the program. This enables any planner to exploit search control specified by the program. We could see our work as an extension of that, where instead of specifying Golog transitions in terms of actions we specify them in terms of goals.

A synthesis problem tightly related to the work presented here is that of behavior composition [35]. In fact, such problem is one of the main starting points for our work here. The idea there is to realize (i.e., implement) a given desired, but non-existent, target module that a user is meant to operate (e.g., a home entertainment system) by suitably coordinating a set of existing available modules (e.g., video cameras, game consoles, automatic blinds and lights, etc.) The problem is in fact a generalization, within a broader AI context, of the well-known web-service composition problem [8,9,72] in which a target web-service is obtained by putting together a set of existing web-services. Like agents in planning programs, the target module user is assumed to operate a behavior specification by issuing requests that ought to be satisfied (by a smart controller, called the "composition"). However, in the composition task the request is for the execution of a particular action (e.g., play music) rather than the achievement of a state of affairs. Moreover, the challenges involve deciding which of the existing available modules will be able to fulfill such request. Rather than searching for adequate behavior delegations, in planning programs, we look for complex conditional programs that could be "stitched" together so as to guarantee declarative goal requests. Because actual domain actions will generally be executed in concrete devices and available modules, it makes sense to look for plans solving a given planning program that could actually be carried out by proper delegation to such modules. It is indeed possible to extend the planning-program framework to accommodate the "delegation" of plans to their actual performers, in the same way as done in behavior composition. This is done by compiling away all behaviors into the underlying dynamic domain; see [34] for details. What is more, it is possible to suitably encode a complete behavior composition task into a planning-program realization, along the line of the hardness proof of Theorem 5. It follows then that the framework for agent planning programs presented here subsumes that for behavior composition. Lastly, we note that similar techniques based on synthesis over specific game structures [35] or automated planning [86] were used to solve the composition task, among others.

The work on agent planning programs is related to generalized planning, in the sense that the result of the planning program realization can be seen as a form of generalized plan (e.g., [13,33,98]). Generalized plans are rich control structures

that include loops and parametrized or lifted actions whose arguments must be instantiated during execution. The work on generalized planning looks at synthesizing a plan that is general enough to realize the same goal on several planning scenarios. Instead, the work presented in this article looks at synthesizing a plan that realizes, within the control structure imposed by the agent program, a collection of interrelated goals over the same planning domain.

Planning programs can also be considered as a form of complex routines, modelling desired domain evolutions and typically including conditions and cycles, that an agent executes in the domain. In planning, similar routines can be specified by temporally extended goals (e.g., [3,6,36,43,61]), in the following abbreviated with TE-goals. Unlike simple achievement goals, which express required properties of the final state achieved by a plan, TE-goals express required properties or constraints on the whole (possibly cyclic) sequences of states traversed by all possible executions of a valid plan. For instance, TE-goals can be used to require that some state properties are achieved according to a certain sequence, that a property holds in every state generated by the plan execution, that a property is achieved periodically or within a certain number of plan steps from a state where another property holds, etc. Planning with a class of TE-goals can be compiled into classical planning by compilation schemes using additional domain predicates and actions (e.g., [6,43]).

TE-goals can also be used to specify domain-specific control knowledge that a planner can exploit to generate plans more efficiently. For deterministic domains, e.g., the forward search planner TLPlan [3] provides a logic-based platform supporting reasoning about search control knowledge, in the form of temporal logic formulae that promising plan prefixes must not violate. Moreover, TLPlan is capable of building cyclic plans modeling required domain evolutions [61] specified by LTL formulas expressing TE-goals. The planning method used by TLPlan relies on the construction and compilation of Büchi automata equivalent to the TE-goals [107], which recognize the language of (cyclic) execution sequences satisfying the goals. For non-deterministic domains, e.g., planner MBP [22] provides a framework to plan for TE-goals expressed using CTL formulas [41] that distinguishes temporal requirements on all possible and on some plan executions [79]. In order to deal with large search spaces, the planning approach used in MBP relies on symbolic model checking techniques and BDDs [16].

While TE-goals are declarative plan requirements, planning programs also provide a way of specifying procedural knowledge of the domain. MBP has been extended to support planning with requirements such as “it should do everything that is possible to achieve a given condition”, with failure situations of the form “try to reach a goal but, in case of failure, do reach a different goal” [65], and with procedural goals specified by constructs expressing conditional and iterative plans. [93]. However, the problems addressed by MBP are quite different from agent planning programs, and it is not clear whether the problem of realizing an agent planning program can be compiled into a MBP problem so that a plan satisfying the MBP's goals corresponds to a planning-program realization. The most significant differences between planning programs and the methods in [65,93] are that the MBP framework cannot cope with the executor decisions, which introduce a sort of nondeterminism in the planning program definition and is a distinguishing feature in our problem, and that MBP requires that at least one execution reaches a successful state. A consequence of the latter point is, e.g., that procedural goals expressed by loops need to terminate, while in planning programs this is not required.

8. Conclusions

The AI community is expressing the need to put more effort in investigating principled ways of integrating planning and acting (and hence programs) [47]. In this paper we have studied the notion of agent planning programs, which is much in line with this need. Agent planning programs are (finite-state) programs whose atomic instructions consist of precondition-invariance-postcondition assertions. These programs need to be compiled into executable ones by replacing such assertions with plans that, under the guarantee that the precondition is satisfied, maintain the invariance condition and achieve the postcondition. The key point is that these plans cannot be computed in isolation, since once a goal (postcondition) has been achieved, new precondition-invariance-postcondition triples need to be fulfilled as prescribed by the program. We have shown a general solution for such programs and characterized the complexity of the problem.

Interestingly, the general solution proposed, which is optimal from the computational complexity point of view, can be implemented directly using game-structure model-checking based synthesis tools such as the mentioned TLV, JTLV and NuGaT, but also Anzu [60] or Ratsy [10]. This general solution has the flavor of universal plans, but may involve more work than really needed. Focusing on deterministic domains, we have developed an iterated-classical-planning technique that exploits goal preferences and plan adaptation methods to speed up the realization of transitions in cycles. We have tested this technique through an array of experiments, demonstrating that the planning-based approach as a whole is an effective way to practically handle agent planning programs in deterministic domains (observe, though, that while we used some well-known domain-independent planners, the aim of such experiments was not to show the goodness of a specific planner or encoding, and other planners could have been used). This is especially the case with planning domains whose state spaces have limited deadends.

There are several further research avenues to explore related to this mix of planning and programming that agent planning programs provide. We mention here some of them at the extremes of the spectrum. On the one hand, a crucial issue that we did not address in this paper is devising convenient representation formalisms for agent planning programs. Indeed, we have simply used transition systems in the present work, which can be considered a general but possibly too pristine formalism for describing dynamic systems. When it comes to applications, better representation formalisms—in the style of those developed in reasoning about action—are preferred. For example, one could resort to variants of high-level agent programming languages like Golog/ConGolog/IndiGolog for expressing agent planning programs. Notice though that, as dis-

cussed in Section 7, one cannot simply adopt their standard computational semantics, and a new sort of off-line semantics would be needed that takes into account the realization of planning programs as discussed in this paper, possibly extended to deal with first-order representation of data giving rise to infinite-state domains. Pushing this line even further, one could consider allowing recursive procedure calls in the planning program, making the planning program itself infinite state (due to the need of, e.g., an unbounded stack for dealing with multiple procedure activations). Recent work on decidable verification of situation calculus [30–32] and other data-aware process formalisms [7,51] becomes very relevant for this kind of research.

At the other end of the spectrum, we are interested in improving and extending implementations based on planning techniques. First of all, we would like to generalize the technique presented here to nondeterministic domains (possibly using conditional or conformant planners), as well as to introduce measures and techniques to compute optimized program realizations. With respect to the latter, one would aim at obtaining plans that are not only good from the computational point of view, but also (or alternatively sometimes) from an engineering point of view by maximizing qualities such as understandability, robustness, and modifiability.

When it comes to planning programs over deterministic domains, we intend to optimize the performance of the algorithm proposed in Section 5 by including heuristics and techniques that take subsequent transitions into account more effectively (when realizing a particular transition), in order to reduce backtracking by avoiding plans that create open pairs from which a next transition cannot be realized. We expect these advanced techniques will be helpful especially for programs over planning domains where our current technique can incur in a high number of backtracks. We would also like to draw from the recent work on HGN-planning (and associated planning systems like GoDeL [94]), which, as mentioned above, exploits goal decomposition and landmarks to solve classical planning on a network of goals, albeit under a different semantics.

Finally, an interesting and challenging direction concerns addressing the realization of *dynamic* planning programs—programs in which states or transitions can be added or removed dynamically, and the preconditions-invariance-postconditions of the transitions can be incrementally revised—without always recomputing a new realization from scratch.

Acknowledgements

The authors would like to thank the anonymous reviewers for their suggestions and comments that helped improve the paper in significant ways. This research was partially supported by the EU Project FP7-ICT 318338 (OPTIQUE), the Sapienza Award 2013 Spiritlets project, the Ripartizione Diritto allo Studio, Università e Ricerca Scientifica of Provincia Autonoma di Bolzano–Alto Adige, under project VeriSynCoPateD (*Verification and Synthesis from Components of Processes that Manipulate Data*), the Australian Research Council (grant DP120100332), and an Australian Academy of Science "Scientific Visit to Europe" mobility award.

Appendix A. An encoding example

In this Appendix we show the actual encoding of the nondeterministic variant of the example presented in Section 3. The encoding is provided in the language SMV, which is a standard input language for some state-of-the art model checkers (such as NuSMV [20] and SMV [73]) that has been adopted also in the synthesis engines TLV [84], JTLV [83] and NuGaT [17]. The use of SMV also allows us to show how to express transitions in a compact way.

An excerpt of the listing is reported in Fig. A.1. The game is organized hierarchically in three modules. The topmost module, `main` (top of figure), encodes the whole game, and is composed of two submodules: `environment`, of type `environment_module` (defined in the bottom right section of the figure), which encodes the behavior of the environment, and `agent`, of type `system_module` (bottom left), which captures the behavior of the system. The module type defines what the module (formal) parameters are, i.e., how the module interacts with other modules, and how it behaves.

Module definitions have several sections. `VAR` is where local variables are declared. Variables can be either boolean, such as `last` of `system_module`, or of enumerated type, such as `fuel` of `environment_module`, which can assume the values `full`, `empty` and `low`. In fact, enumerated types are suitably represented using (arrays of) boolean values, so we can consider the game as defined over boolean variables only. In the case of `main` also the modules representing the players, although not being proper variables, are declared in section `VAR`, using the keyword `system`. According to the semantics of two-player games the state transitions of `main` are obtained by concatenating an `agent` transition to an `environment`'s, starting with both players in their initial state, with `environment` moving first.

When a module, such as `agent` or `environment` in `main`, is instantiated, its formal parameters are bound to variables (possibly module instances themselves), which can then be accessed by the instantiated module. For instance, the declaration of `environment` in `main` states that `environment` can access the module instance `agent`, and in particular all of its local variables, i.e., `agent.act`, `agent.last`, `agent.viol`.

The `VAR` section of `system_module` contains the declarations of system variables, i.e., the controlled variables. These include one enumerated variable `act` for actions, as well as boolean variables `last` and `viol`. In `environment_module`, some enumerated variables (`fuel`, `my_loc`, `car_loc`) are used to concisely capture exclusive propositions. For instance, since fuel can be at one level only, the `fuel` variable can assume values `full`, `low` and `empty`. The boolean variables

```
MODULE main
 VAR
  env : system environment_module(agt);
  agt : system system_module(env);
 DEFINE
  good := agt.last;
-- end of main


MODULE system_module(env)
VAR
  act: {start, wait, refuel, dr_home,...};
  last: boolean;
  viol: boolean;
 INIT
  act = start & last & !viol
 TRANS
  next(act) != start &
  case -- action preconditions
    next(act) = refuel:
     next(env.my_loc) = next(env.car_loc);
    next(act) = dr_home:
     next(env.fuel) != empty &
     next(env.my_loc) = env.car_loc;
    ...
    -- preconditions for remaining actions
  esac &
  case -- goal achievement
    next(env.pp_trans) = tr_1 & next(last):
     next(env.my_loc) = dept & next(!viol) &
     next(env.fuel) != empty;
    ...
    -- remaining transitions
  esac &
  case -- violations
    act = start: !next(viol);
    viol: next(viol);
    next(env.pp_trans) = tr_1 &
      next(env.fuel) = empty : next(viol);
    ...
    TRUE: next(viol) = viol;
  esac
-- end of system_module
```

```
MODULE environment_module(sys)
 VAR
  fuel: {full, low, empty};
  my_loc: {home, pub, lot, dept};
  car_loc: {home, pub, lot};
  driven: boolean;
  rain: boolean;
  pp_state: {start,v0,v1,v2};
  pp_trans: {start, tr_1,...,tr_5};
 INIT
  fuel = full & my_loc = home & !driven &
  car_loc = home & !rain &
  pp_state = start & pp_trans = start
 TRANS
  case -- propostion fuel
    sys.act = start: next(fuel) = full;
    sys.last: next(fuel) = fuel;
    sys.act = refuel: next(fuel) = full;
    sys.act in {dr_home, dr_pub, dr_lot}:
     fuel = full-> next(fuel) in {full,low} &
     fuel = low -> next(fuel) in {low,empty};
    TRUE : next(fuel) = fuel;
  esac &
   -- case blocks for remaining propostions
  ...
  case -- planning program requests
    next(pp_state) = v0 :
        next(pp_trans) in {tr_1,tr_4};
    ...
  esac &
  case -- guards on program transitions
    next(pp_trans) = tr_4 : next(!rain);
    TRUE : TRUE;
  esac  &
  case -- transition on goal achievements
    sys.act = start : next(pp_state) = v0;
    pp_trans = tr_1 & sys.last :
      next(pp_state) = v1;
    ...
    TRUE : next(pp_state) = pp_state &
           next(pp_trans) = pp_trans;
  esac
-- end of environment_module
```

Fig. A.1. Excerpt of SMV encoding for the example of Section 3.

`driven` and `rain` keep track, respectively, of whether the researcher has just driven and whether it is raining. The remaining variables, `pp_state` and `pp_trans`, record the current state of the planning program and the current transition requested, respectively. Notice that, for convenience, planning program transitions are named. For instance, `tr_1` represents the transition from v_0 to v_1.

In section `DEFINE`, propositional formulas can be defined. This feature is used only in `main`, where `good` is a reserved name for the formula representing the propositional part of the goal. The declaration `good:= agent.last` asserts that the winning condition of the game, i.e., the formula φ_{goal}, is $\Box\Diamond$`agent.last` (the temporal modalities $\Box$ and $\Diamond$ are implicit).

The remaining sections `INIT` and `TRANS` are used to define, respectively, the initial state and the transition relation of a module. The former contains a formula stating what the initial values of local variables are. For instance, the `INIT` section of `system_module` expresses the fact that, initially, `act` is assigned to `start`, `last` is true and `viol` is false (symbols `&`, `|` and `!` stand, respectively, for logical and, or and not). These are arbitrary default values, assigned so as to have only a single initial state. Action `start`, the only action that the system can select in the initial state, will set each variable (possibly nondeterministically) to its actual initial value (leaving unconstrained variables free to range over their definition domain). The corresponding section of `environment_module` has an essentially analogous structure.

Section `TRANS` contains a formula that relates the value of each variable at a state with the values of other variables at current or next state. This essentially defines the transition relation for the module as including all those pairs of (current and successor) states whose variable assignments satisfy the formula. Keyword `next` is used to refer to the value of a variable at next state. For instance, the expression `next(act) != start` expresses that the value of `act` at next state cannot be `start` (which is allowed only in the initial state). Typically, `TRANS` is a complex formula obtained as a conjunction of `case` blocks which consist of an ordered list of cases, each defined by a condition followed by a consequence, separated by

character ':'. The semantics of a case stipulates that if the condition holds, the consequence holds too. Cases are evaluated in the order they appear in the block: when a case is encountered whose condition holds, the case block assumes the value of the consequence associated with the selected case. For instance, in the first block of the `TRANS` section of `module_environment`, the first case states that after action `start` is executed (`sys.act = start`), the fuel level will be full: `next(fuel) = full` (lines starting with – are commented). The second case, instead, states that if `sys.last` holds in the current state, the fuel level does not change at next state. The second case, however, is considered only if the condition of the first one is not satisfied.

The `TRANS` sections of `system_module` and `enviroment_module` encode the transition relations of the system and environment players, respectively, for our example. Such relations are expressed through boolean formulas, in particular as conjunctions of `case` blocks.

Consider the `TRANS` section of `system_module`. The first `case` block captures action preconditions. In details, the first line expresses that action `refuel` can be selected for execution at next step only if, at the same step, it is the case that `env_myloc` and `car_loc` match, i.e., that the researcher and the car are in the same location. The second line, which is considered only if the condition of the previous case does not hold, requires that, in order for the researcher to be able drive home, the tank of the car must be not empty. In addition to preconditions for regular domain actions, this block includes also constraints and preconditions on action `wait`, which has to be executed whenever `last` holds, and can be selected (when `last` does not hold) only if none of the preconditions for the other actions is satisfied.

The second block encodes the same constraints on proposition *last* discussed in previous section, i.e., that *last* can hold only if the current achievement goal is indeed achieved and no violation has occurred. So, e.g., the first line of this block encodes that if `tr_1` is the current requested transition, `last` can be true only if the researcher is at the department and no violation has been recorded (in variable `viol`). The last block captures when violations occur and, in particular in the last line, that once occurred they are recorded forever.

As to `environment_module`, the `TRANS` section contains a first set of `case` blocks which capture the effects of actions. For instance, the first block captures how driving actions affect the fuel level of the car. Notice that the evolution of fuel is nondeterministic. For each variable used to encode the domain state there is a distinct `case` block, each considering all of the possible actions. After these, a block for the transition requests is present, which states what the transitions are that can be requested for each state that the planning program can be in. For instance, at state `v0` only `tr_1` or (symbol |) can be requested. Notice that if a variable is not constrained by `TRANS`, its value can freely range over its domain. For instance, variable `rain` can assume any value after the execution of any action. The next block captures the guards on program transitions. In particular, it requires that, in order for transition `tr_4` to be requested, `rain` must be false (this is the only transition where a guard other than `true` is present). Observe that the guard is required to be satisfied only when the transition request is *new*, i.e., immediately after an occurrence of *last*, which marks the realization of the current transition. Finally, the last block encodes the evolution of program states when the currently requested transition is realized. For example, the second line of this block states that when `tr_1` is requested and `sys.last` holds (meaning that the requested transition is realized), the next state will be `v1`, i.e., the destination state of the transition.

We observe that, in SMV, the use of propositional formulas allows one to refer to sets of states, without having to list them explicitly, by considering a formula as representative of those states that satisfy it. In the same way, through the use of the `next` operator one can compactly represent transitions between states. Interestingly, TLV, JTLV and NuGaT, as well as many other synthesis engines, take advantage of this *symbolic* representation [16], typically optimized using ordered binary decision diagrams, to efficiently manipulate sets of states and transitions.

Appendix B. Proofs

Proof of Theorem 3 (page 75). First of all, by the definition of G, each game state W encodes:

- the (current) domain state $W[s] = \{p \mid p \in W \cap P\}$, that is, the projection of W on the set of $\mathcal{D}$'s domain propositions P (recall that $\mathcal{X}_P = P$);
- the (current) planning program state $W[v] \in W \cap V$, which always exists and is unique due to definition of ρ_e (recall that $\mathcal{X}_V = V$);
- the transition $W[d] = \langle v, \gamma, \psi, \phi, v' \rangle$ such that $W[v] = v$, and $v, req^{v,v'}_{\gamma:\psi,\phi} \in W$, which always exists and is unique due to definition of ρ_e (see E6 and E8);
- the action $W[a] \in W \cap A$, which always exists and is unique by the definition of ρ_s (S2).

We will say that W *represents* domain state $W[s]$, planning program state $W[v]$, transition $W[d]$, and action $W[a]$ (or, alternatively, that these are represented in W).

(If Part) Assume that a strategy f winning for the system exists, and let $\sigma = W_0 W_1 \cdots$ be a play compliant with f. From now on, we use $s^i = W_i[s]$, $v^i = W_i[v]$, $d^i = W_i[d]$, and $a^i = W_i[a]$, for all $i \geq 0$.

We start by making two observations. First, transition relations ρ_e and ρ_s guarantee that:

- W_1 represents the initial domain state s_0 of $\mathcal{D}$ (E1) and the initial program state v_0 of $\mathcal{P}$ (E4), that is, $W_1[s] = s^1 = s_0$ and $W_1[v] = v^1 = v_0$;

- if $last \notin W_i$ and $i \geq 1$, then $v^i, req^{v^i,v'}_{\gamma:\psi,\phi} \in W_{i+1}$ (i.e., program state and current request remain unchanged; see E10 and E11), and $s^{i+1} \in \tau(s^i, a^i)$ (i.e., the domain state represented in the next game state is one resulting from the execution of the action executed; see E3);
- if $last \in W_i$ and $i > 0$, then $s^{i+1} = s^i$ (i.e., selected action a^i has no effect; see E2), and there exists $req^{v',v''}_{\gamma':\psi',\phi'} \in W_{i+1}$ such that v' is represented in W_{i+1} (E9), $req^{v^i,v'}_{\gamma:\psi,\phi} \in W_i$ and $s^{i+1} \models \gamma'$ (i.e., W_{i+1} represents a new and valid request transition from the new program state; see E6, E7, and E9).

Second, because f is a winning strategy and σ is compliant with f, we have that:

- for all $i \geq 0$, $violated \notin W_i$, that is, for $req^{v^i,v'}_{\gamma:\psi,\phi} \in W_i$, it is the case that either $W_i \models \psi$ or $W_i \models last$ (see S5, S6 and S7);
- if $last \in W_i$ and $req^{v^i,v'}_{\gamma:\psi,\phi} \in W_i$, then $W_i \models \phi$, that is, when the system "plays" *last*, the achievement goal of the current $\mathcal{P}$-transition requested is achieved (S5);
- *last* holds infinitely many times along σ, as required by φ_{goal}.

Let us prove that $\mathcal{P}$ is realizable in $\mathcal{D}$ from s_0. By Theorem 1, it is enough to prove the existence of a realization by showing a *PLAN*-based simulation R (Definition 3) such that $\langle v_0, s_0\rangle \in R$. To that end, consider the relation $R \subseteq V \times 2^P$ defined as:

$$\langle v, s\rangle \in R$$

if and only if

there exists an f-compliant play $\sigma = W_0 W_1 \cdots$ such that for some $i \geq 0$, it is the case that

$$last \in W_i,\ W_{i+1}[s] = s \text{ and } W_{i+1}[v] = v.$$

Informally $\langle s, v\rangle$ is in R if there is a winning play where s and v are represented in a state (W_{i+1}) just after a previous request has been completed (signaled by *last* being true in W_i). Observe that because $last \in W_0$, this includes the case when s and v are initial for $\mathcal{D}$ and $\mathcal{P}$, respectively, i.e., $W_1[s] = s_0$ and $W_1[v] = v_0$. In other words a new request in domain state s and agent planning program state v has just been initiated (in game state W_{i+1}).

The fact that $\langle v_0, s_0\rangle \in R$ is trivial given that $last \in W_0$ and, as observed above, $W_1[s] = s_0$ and $W_1[v] = v_0$ (this holds for any f-compliant play). So, it remains to show that R is a *PLAN*-based simulation (as defined in Definition 3). To that end, let $\langle v, s\rangle$ be a pair in R, and consider a transition $v \xrightarrow{\gamma:\psi,\phi} v'$ in $\mathcal{P}$ such that $s \models \gamma$. First of all we know that:

† Since $\langle v, s\rangle \in R$, there exists an f-compliant play $\sigma = W_0 W_1 \cdots W_\ell W_{\ell+1} \cdots$ such that $W_{\ell+1}[s] = s$ and $W_{\ell+1}[v] = v$, for some $\ell > 0$, and $last \in W_\ell$.

†† Given that game G accounts for every transition in the agent planning program (see E6), there is one such play σ such that $req^{v,v'}_{\gamma:\psi,\phi} \in W_{\ell+1}$ (i.e., $W_{\ell+1}[d] = \langle v, \gamma, \psi, \phi, v'\rangle$).

Next, let us define an HT-plan π for such transition such that π achieves ϕ while maintaining ψ from state s (first constraint in Definition 3), and π preserves R (second constraint in Definition 3). The idea is to define a general (conditional) plan that makes the same action selections, at every step, as those done by the winning strategy f. The key is that the winning strategy f is indeed encoding a valid HT-plan. More concretely, consider the general plan π such that for any history $h = s^0 \xrightarrow{a_1} s^1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s^n$ with $s^0 = s$, it is the case that:

- $\pi(s^0 \xrightarrow{a_1} s^1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s^n) = a_{n+1}$, if there exists a play $\hat{\sigma} = \sigma|_\ell W^0 W^1 \cdots W^n \cdots$ such that:
  - play $\hat{\sigma}$ is complaint with strategy f;
  - $W^0 = W_{\ell+1}$;
  - $W^i[s] = s^i$ and $W^i[a] = a_{i+1}$, for all $i \in \{0, \ldots, n\}$; and
  - $last \notin W^i$ for all $i \in \{0, \ldots, n-1\}$.
- $\pi(s^0 \xrightarrow{a_1} s^1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s^n)$ is left undefined, otherwise.

It is not difficult to see that, because of all four constraints and the fact that strategies are deterministic, all plays $\hat{\sigma}$ as above will coincide on all game states W^0 to W^n, and hence plan π is well-defined.

Let us first prove that π is an HT-plan, that is, that all executions of π are *finite*. Suppose, on the contrary, that there is an infinite execution $h = s^0 \xrightarrow{a_1} s^1 \xrightarrow{a_2} \cdots$ of π. This means that $\pi(s^0 \cdots s^k) = a_{k+1}$, for each $k \geq 0$, and because of the way π was built, this implies that there has to exist an (infinite) play $\hat{\sigma} = \sigma|_\ell W^0 W^1 \cdots$ such that each W^i corresponds to each state s^i (i.e., $W^i[s] = s^i$) and action a_{i+1} (i.e., $W^i[a] = a_{i+1}$). More importantly, since h is infinite and π is always defined, it has to be the case, due to the last constraint in definition of π, that $last \notin W^i$, for all $i \geq 0$—*last* does not hold true from

W^0 onwards. It follows then that $\hat{\sigma} \not\models \Box\Diamond \mathit{last}$. However, this is a contradiction, since $\hat{\sigma}$ is a play compatible with strategy f (by how σ was constructed above), and f is a winning strategy for a game whose goal is indeed $\varphi_{goal} = \Box\Diamond \mathit{last}$. Hence, the above infinite execution h of π may not exists, all executions of π are finite, and π is an HT-plan.

Next let us show that the two constraints in the *PLAN*-based simulation Definition 3 are satisfied. To that end, take any complete execution $h = s^0 \xrightarrow{a_1} s^1 \xrightarrow{a_2} \cdots s^{n-1} \xrightarrow{a_n} s^n$ of HT-plan π. Then:

- Because of the way that π was constructed we can infer that:
  - there exists an f-compliant play $\hat{\sigma} = \sigma|_\ell W^0 W^1 \cdots W^{n-1} W^n \cdots$ such that, among other things, $\mathit{last} \notin W^i$ and $W^i[a] = a^{i+1}$, for all $0 \le i \le n-1$;
  - $\pi(h)$ is undefined, since h is complete (i.e., h cannot be further extended with π);
  - it has to be the case that $\mathit{last} \in W^n$. Suppose, on the contrary, that this is not the case and $\mathit{last} \notin W^n$. Then, take any f-compliant play $\hat{\sigma}' = \sigma|_\ell W^0 W^1 \cdots W^n W' W'' \cdots$. There has to be at least one such play given that ρ is, by construction, serial (i.e., there are no deadends). Since $\pi(h)$ is undefined, it has to be the case that play $\hat{\sigma}'$ does not satisfy one of the four constraints in the definition of π above. However, $\hat{\sigma}'$ trivially satisfies the first three requirements, so $\mathit{last} \in W^i$, for some $i \le n$. We know, due to existence of $\hat{\sigma}$ as above, that $\mathit{last} \notin W^i$, for all $0 \le i \le n-1$. Then, $\mathit{last} \in W^n$.

  Now, recall from (††) above, that $req^{v,v'}_{\gamma:\psi,\phi} \in W^0 = W_{\ell+1}$. Due to (E11) and the fact that $\mathit{last} \notin W^i$, for all $i \le n$, such active request is propagated throughout the whole game play up to W^n included. Hence, $req^{v,v'}_{\gamma:\psi,\phi} \in W^n$. That, together with the fact that $\mathit{last} \in W^n$ and (S5), implies that:
  - $W^n[s] = s^n \models \phi$ and since h stands for *any* complete execution of π, π achieves ϕ from state $s^0 = s$.
  - $W^n \models \neg \mathit{violated}$. Due to axiom (S7), we conclude that $W^0 \models \neg \mathit{violated}$. This together with constraint (S6), implies that $W^i[s] = s^i \models \psi$, for all $i \in \{0, \ldots, n-1\}$. Thus, π maintains ψ from state $s^0 = s$.

  So, putting it all together, HT-plan π achieves ϕ while maintaining ψ from state s.
- Consider the play $\hat{\sigma} = \sigma|_\ell W^0 W^1 \cdots W^{n-1} W^n W^{n+1} \cdots$ from above. We already know that the play is complaint with strategy f, and that $\mathit{last}, req^{v,v'}_{\gamma:\psi,\phi} \in W^n$. Due to axiom (E9), it follows then that $W^{n+1}[v] = v'$.
  We also know that $W^n[s] = s^n$. Because $\mathit{last} \in W^n$, it follows due to axiom (S4) that $\textsc{wait} \in W^n$ (a no-op action is done at game state W^n). Because of axiom (E2), $W^{n+1}[s] = s^n$ has to hold—the domain remains still. Thus, by how R was defined, we conclude that $\langle v', s^n \rangle \in R$, that is, π preserves R from $\langle v, s \rangle$ for transition $v \xrightarrow{\gamma:\psi,\phi} v'$.

Summarizing, we have just demonstrated that for any transition $v \xrightarrow{\gamma:\psi,\phi} v$ in $\mathcal{P}$, the plan π is a conditional plan satisfying both requirements of Definition 3. Then, relation R is indeed a *PLAN*-based simulation and the existence of a realization is guaranteed by Theorem 1.

(Only-If Part) Let $\Omega : 2^P \times \delta \mapsto HT_{\mathcal{D}}$ be a realization of $\mathcal{P}$ in $\mathcal{D}$ from s_0. From Ω, we shall derive a winning strategy f for the system, by induction on the length of environment moves $X_0 X_1 \cdots X_n$. For the base case, we define $f(X_0 = X_I) = Y_I = \{\mathit{init}, \mathit{last}\}$.

Assume f is defined for ℓ moves, and let us define the system $\ell+1$'s move as per strategy f. To that end, consider a legal sequence of $\ell+1$ environment moves of the form $\lambda = X_0 X_1 \cdots X_\ell X_{\ell+1}$, with $\ell \ge 0$, such that $\sigma = W_0 W_1 \cdots W_\ell$ is a finite play compliant with λ and f, that is, $W_i = (X_i, f(X_0 \cdots X_i))$, for $0 \le i \le \ell$. Then, we define:

$$f(X_0 \cdots X_{\ell+1}) = \begin{cases} \{a\} & \text{if} \quad a = \pi(X_{k+1}[s] \cdots X_{\ell+1}[s]) \\ \{\mathit{last}, \textsc{wait}\} & \text{if} \quad \pi(X_{k+1}[s] \cdots X_{\ell+1}[s]) \text{ is undefined} \end{cases}$$

where:

- index $k \le \ell$ is the largest index such that $\mathit{last} \in W_k$ in σ. That is W_k represents the last state where a transition request was fulfilled and a new request has been issued at X_{k+1} and is still "active."
- HT-plan π is defined as $\pi = \Omega(X_{k+1}[s], X_{k+1}[d])$. Basically, plan π is the plan that was prescribed by realization Ω when the transition request $X_{k+1}[d]$ was issued at state $X_{k+1}[s]$. Since W_k is the last step where *last* holds true, such request is still "active." We prove below that π does exist.

Next, we are to prove that function f is indeed a strategy for G (in doing so, we show that plan π above always exists). We do so by induction on the length of λ:

- If $\ell = 0$, then $\lambda = X_0 X_1$ (and $\sigma = W_0$). Thus, $k = 0$ (recall $\mathit{last} \in W_0$ since $\mathit{last} \in Y_I$) and, by axioms E1 and E4, $X_1[s] = s_0$ and $X_1[v] = v_0$. Moreover, because of axioms E6 and E7, $req^{v_0,v'}_{\gamma:\psi,\phi} \in X_1$ denotes some legal request transition $X_1[d]$ from v_0 with its guard being true in s_0 (i.e., $s_0 \models \gamma$). Due to Definition 4, $\Omega(X_1[s], X_1[d])$ is defined, that is, $\Omega(X_1[s], X_1[d]) = \pi$ for some HT-plan π that achieves ϕ while maintaining ψ. We need now to consider two cases:

- if $\pi(X_1[s])$ is defined, then $Y_1 = f(X_0X_1) = \pi(X_1[s]) = \{a\}$. Because plan π achieves ϕ, action a is executable in domain state $X_1[s]$ and hence axiom S3 is satisfied. Also, since π maintains ψ, $X_1[s] = s_0 \models \psi$. This, together with the fact that *violated* $\notin Y_I = Y_0$ and *violated* $\notin Y_1$, implies that axioms S6 and S7 are also met by f's prescribed move. The other constraints on the system are trivially satisfied.
- if $\pi(X_1[s])$ is undefined, then $Y_1 = f(X_0X_1) = \{last, \text{WAIT}\}$. This is the case when the request is satisfied in s_0, without performing any (domain) action. Because plan π achieves ϕ, it is the case that $X_1[s] = s_0 \models \phi$. This, together with the fact that *violated* $\notin Y_1$, implies that axiom S5 is satisfied by f's move (i.e., WAIT), which requires the domain to remain in s_0, so that ϕ is still satisfied after action execution. Moreover, since *last* $\in Y_1$ and *violated* $\notin Y_1$, axioms S6 and S7 are met as well. Also the remaining constraints on the system can be easily checked to be satisfied.

- Next consider $\lambda = X_0X_1 \cdots X_\ell X_{\ell+1}$, for some $\ell \geq 0$. By induction hypothesis, $f(X_0X_1 \cdots X_\ell)$ is defined and is a legal system move. This means that there exists $k' < \ell$ and a plan π' as per definition of f above that was used to define $f(X_0X_1 \cdots X_\ell)$. We consider again two cases:
  - if $f(X_0X_1 \cdots X_\ell) = \{a\}$, for some domain action a, then the same request as one step before is still active and we can therefore take $k = k'$ and $\pi = \pi'$ to define $f(X_0X_1 \cdots X_\ell X_{\ell+1})$. We use an analogous reasoning as that for the base case:
    - if $\pi(X_{k+1}[s] \cdots X_{\ell+1}[s])$ is defined, then $Y_{\ell+1} = f(X_0 \cdots X_{\ell+1}) = \pi(X_{k+1}[s] \cdots X_{\ell+1}[s]) = \{a\}$. Because plan π achieves ϕ, action a is executable in domain state $X_{\ell+1}[s]$ and hence axiom S3 is satisfied. Also, since π maintains ψ, $X_{\ell+1}[s] \models \psi$, which, together with the fact that *violated* $\notin Y_\ell$ and *violated* $\notin Y_{\ell+1}$, implies that axioms S6 and S7 are also met by f's prescribed move. The other constraints on the system are trivially satisfied.
    - if $\pi(X_{k+1}[s] \cdots X_{\ell+1}[s])$ is undefined, then $Y_{\ell+1} = f(X_0 \cdots X_{\ell+1}) = \{last, \text{WAIT}\}$. This is the case in which the active request has just been met at $X_{\ell+1}$. Because plan π achieves ϕ, it is the case that $X_{\ell+1}[s] \models \phi$. This, together with the fact that *violated* $\notin Y_{\ell+1}$, implies that axiom S5 is satisfied by f's move (WAIT). Moreover, since *last* $\in Y_{\ell+1}$ and *violated* $\notin Y_{\ell+1}$, axioms S6 and S7 are met as well. The other constraints on the system are trivially satisfied.
  - if $f(X_0X_1 \cdots X_\ell) = \{last, \text{WAIT}\}$, then the latest request issued at game state $W_{k'+1}$ has just been fulfilled in the previous game state $W_\ell = (X_\ell, \{last, \text{WAIT}\})$. We therefore take $k = \ell$ in order to define $f(X_0X_1 \cdots X_{\ell+1})$. Because λ is a legal sequence of environment moves, $\rho_e(W_\ell, X_{\ell+1})$ applies, and hence all axioms of the environment are met. This means that move $X_{\ell+1}$ encodes the successor state v' for the planning program $\mathcal{P}$, the same domain state as X_ℓ (due to WAIT action), and a new legal transition request from v' (with its guard true at state $X_{\ell+1}$). Because Ω is a realization, plan π' preserves a *PLAN*-simulation relation R, for which, in particular, it is the case that $R(X_{\ell+1}[v], X_{\ell+1}[s])$. Hence, by Definition 4, $\Omega(X_{\ell+1}[s], X_{\ell+1}[d])$ is defined, yielding an HT-plan π that realizes transition $X_{\ell+1}[d]$ (i.e., brings about $X_{\ell+1}[d]$'s achievement goal while respecting its maintenance goal). Therefore, $f(X_0X_1 \cdots X_{\ell+1})$ is defined and we can apply the same case reasoning as above, depending on whether π prescribes a domain action or not (i.e., WAIT), to show that f respects the rules of the game for the system player and is indeed a legal strategy. □

Finally, the fact that f is indeed a winning strategy follows from the fact that it is defined in terms of HT-plans that are *finite*. This means that, eventually, every HT-plan will complete, will be undefined, and f will eventually always play proposition *last*, thus meeting G's winning condition.

Proof of Theorem 5 (page 76). For EXPTIME membership we just observe that the general procedure works for this special case as well.

For the EXPTIME-hardness, we show a reduction from behavior composition problem for deterministic behaviors which is known to be EXPTIME-hard [76].

We define an (agent) *behavior* as a tuple $\mathcal{B} = \langle B, A, b_0, \varrho \rangle$, where:

- B is the finite set of behavior's states;
- A is the finite set of behavior's actions;
- $b_0 \in B$ is the behavior's initial state;
- $\varrho : B \times A \mapsto B$ is the behavior's partial transition function.

Notice behaviors are *deterministic* since ϱ is a partial function.

The behavior composition problem can be phrased as follows: check if a target behavior $\mathcal{T} = \langle T, A, t_0, \varrho_t \rangle$ can be *simulated* [75] by the asynchronous product of the available behavior $\mathcal{B}_1, \ldots, \mathcal{B}_n$ with $\mathcal{B}_i = \langle B_i, A, b_{i0}, \varrho_i \rangle$, with $i \in \{1, \ldots, n\}$. The problem is known to be EXPTIME-hard in the number n of available behaviors [76].

We reduce to the realization of the planning programs in deterministic domain as follows. First we define the dynamic domain $\mathcal{D} = \langle P, 2^P, A, \rho \rangle$ and an initial state S_0 as follows:

1. $P = (\bigcup_{i=1}^{n} P_i) \cup \{\mathsf{Exec}_a \mid a \in A\} \cup \{\mathsf{Exec}_{reset}\}$, where $P_i = \{b \mid b \in B_i\}$ is a set of new propositions representing the different states of available behavior $\mathcal{B}_i$; propositions Exec_a records "behavior action" a has just been executed; and proposition Exec_{reset} does this for an extra special action *reset* states whether.

2. $A' = A_b \cup \{\text{RESET}\}$ where $A_b = \{a_i \mid a \in A,\ i \in \{1, \ldots, n\}\}$, that is, the domain actions are formed by the behaviors' actions further annotated with the behavior that just did the action, plus the special action RESET.
3. $\rho \subseteq 2^P \times A' \times 2^P$, such that
   - $\langle S, \text{RESET}, S' \rangle \in \rho$ iff
     - $\mathsf{Exec}_{reset} \notin S$, that is, annotated behavior actions are not enabled and hence the only action enabled is RESET;
     - $S' = (S - \{\mathsf{Exec}_a \mid a \in A\}) \cup \{\mathsf{Exec}_{reset}\}$, that is, the only effect of RESET is making Exec_{reset} true and reset all $Exec_a$ to false;
   - $\langle S, a_i, S' \rangle \in \rho$ with $a_i \in A_b$ iff
     - $\mathsf{Exec}_{reset} \in S$, that is, the last action executed is RESET;
     - $b_i \in S$ and $b'_i \in S'$ for $\varrho_i(b_i, a) = b'_i$, that is, behavior i moves from state b_i to state b'_i according to its transition function ϱ_i;
     - for all $j \neq i$, it is the case that $S \cap B_j = S' \cap B_j$, that is, all other behaviors $j \neq i$ remain still;
     - $\mathsf{Exec}_{reset} \notin S'$, in this way all behavior actions are disabled after the transition in new state S'; and
     - $\mathsf{Exec}_a \in S'$, S' records the fact that behavior action a has just been performed.
4. $S_0 = \{b_{10}, \ldots, b_{n0}\}$, that is, the initial state of $\mathcal{D}$ denotes that all available behavior are in their respective initial states, but also that behavior actions are not enabled (only a RESET action can be executed initially).

We next build, based on target behavior $\mathcal{T} = \langle T, A, t_0, \varrho_t \rangle$, the planning program $\mathcal{P}$ for the dynamic domain $\mathcal{D}$ above as $\mathcal{P} = \langle P, V, v_0, \delta \rangle$, where

- P is the set f propositions of the dynamic domain $\mathcal{D}$;
- $V = \{t^0, t^1 \mid t \in T\}$, that is the planning program has as states, the states of $\mathcal{T}$ annotated with 0 and 1 (i.e., $\mathcal{P}$ doubles the states of the target behavior $\mathcal{T}$);
- $v_0 = t^0_0$, that is, the same initial state of $\mathcal{B}_0$ annotated with 0;
- δ is defined as follows:
  - $t^0 \xrightarrow{true:\mathsf{Exec}_{\text{RESET}}, \neg\mathsf{Exec}_{reset}} t^1$, that is RESET actions are used to move from the a state $t \in T$ annotated with 0 to the same state annotated with 1, with the only effect of enabling "regular actions" by making Exec_{reset} true, while maintaining that no other action has been executed (guards are not used in the encoding and are simply put to $\top$);
  - $t^1 \xrightarrow{true:\mathsf{Exec}_a, \mathsf{Exec}_{reset}} t'^0$ for $\varrho_{\mathcal{T}}(t, a) = t'$, that is, we mimic the actions in $\mathcal{T}$ but moving from state annotated with 1 to states annotated with 0.

  the result of this is that in states annotated with 0 the only transition allowed resets the executability of actions, and in the states annotated with 1 action requests according to the target behavior $\mathcal{T}$ are made.

The key point is that the only way to satisfy $[true : \mathsf{Exec}_{\text{RESET}}, \neg\mathsf{Exec}_{reset}]$ involves the execution of a *single* action RESET and to satisfy $[true : \mathsf{Exec}_a, \mathsf{Exec}_{reset}]$ we must use a *single* action within $\{a_1, \ldots, a_n\}$. It is immediate to verify that if the planning program $\mathcal{P}$ is *PLAN*-simulated by $\mathcal{D}$ iff $\mathcal{T}$ is simulated by the asynchronous product of $\mathcal{B}_1, \ldots, B_n$. Hence from the EXPTIME-hardness in [76] we get the EXPTIME-hard lower-bound for our case. □

Proof of Lemma 1 (page 78). Termination is guaranteed because the number of possible open pairs that can be generated by the algorithm is finite, at every iteration of the external loop (lines 5–25) an open pair $\langle s, v \rangle$ is extracted from *Open* (line 6), and the number of times any open pair is added by steps 15 and 25 to *Open* is finite. The latter point holds because:

- Step 15 never adds the same pair $\langle s, v \rangle$ to *Open* more than once, because it adds the pair to *Open* only if the end state s of the plan computed by Plan for a realization d incoming to v is not in $States(v)$ (step 14); moreover, when $\langle s, v \rangle$ is added to *Open*, $States(v)$ is extended with s (step 16), and, when s is removed from $States(v)$, $Tabu(v)$ is extended with s (step 21–22), preventing the generation of any plan achieving s.
- Step 25 adds a pair $\langle s, v \rangle$ to *Open* only if the realization of $\langle \overline{s}, \overline{v} \rangle$ fails, where $\langle s, d \rangle \in Source(\overline{s}, \overline{v})$ and d is a program transition from v to $\overline{v}$, and only if $\langle s, v \rangle$ becomes part of the realization frontier using Ω modified by removing the plan that realizes transition d. In the worst case, there exist $|V|$ transitions outgoing from v whose guard holds in s. Since we are assuming that the realization of $\langle \overline{s}, \overline{v} \rangle$ fails, at least one transition outgoing from $\overline{v}$ cannot be realized from $\overline{s}$. When the algorithm fails to realize such a transition, step 21 (permanently) adds $\overline{s}$ to $Tabu(\overline{v})$, and hence step 15 cannot add this pair again to *Open*. Therefore, the algorithm can realize transition d from state s at most $|S|$ times (the maximum number of different end states of a plan), and $\langle s, v \rangle$ is added to *Open* at most $|S| \cdot |V|$ times.

This guarantees that the condition of the external loop becomes false after a finite number of iterations and so that the algorithm terminates. □

Proof of Theorem 6 (page 78). Assume that the function Ω returned by the algorithm is not a valid realization for the input agent planning program $\mathcal{P}$. Then, by Definitions 3 and 4, there exists at least one pair $\langle s, v \rangle$ reached when $\mathcal{P}$ is

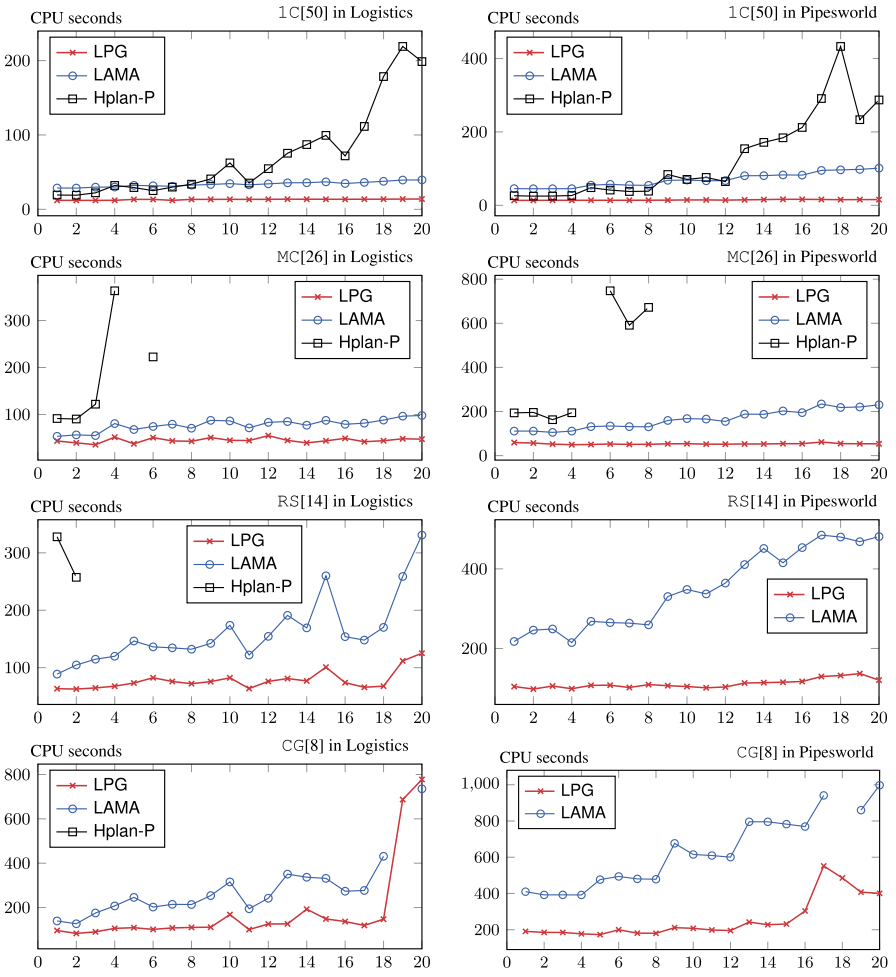


Fig. C.1. CPU time for benchmark SM_{50}.

executed according to Ω and a program transition $d = \langle v, \langle \gamma, \psi, \phi \rangle, v' \rangle$ with $s \models \gamma$ such that either (1) $\Omega(s, d) = \texttt{noPlan}$, or (2) $\Omega(s, d) = \pi$ and π does not maintain ψ or $last(\pi(s)) \not\models \phi$. Case (1) cannot hold because Ω is returned only if *Open* is empty and all pairs $\langle s, v \rangle$ that are reachable according to Ω are added to *Open* (steps 13–16), to be then removed from *Open* when (1.a) every transition d outgoing from v is either correctly realized by $\Omega(s, d)$ or the guard of d does not hold in s, or (1.b) an outgoing transition whose guard holds in s cannot be realized. However, pair $\langle s, v \rangle$ cannot be removed from *Open* because of (1.a), since we are assuming that $\Omega(s, d) = \texttt{noPlan}$ and $s \models \gamma$; $\langle s, v \rangle$ can neither be removed because of (1.b), since, when a transition outgoing from v whose guard holds in s cannot be realized from state s, Ω is set undefined for all $\langle s'', d'' \rangle$ that are sources of $\langle s, v \rangle$ (steps 23–24), while we are assuming that $\langle s, v \rangle$ is reached when $\mathcal{P}$ is executed according to Ω. Case (2) cannot hold because we are assuming that procedure Plan is sound. □

Proof of Theorem 7 (page 78). Assume that there exists a realization Ω for $\mathcal{P}$, and let $\langle s_0, v_0 \rangle$ be the initial open pair. For every transition $d = \langle v_0, \langle \gamma, \psi, \phi \rangle, v \rangle$ outgoing from v_0 such that $s_0 \models \gamma$, there exists a plan π such that $s' = last(\pi(s_0))$, $\Omega(s_0, d) = \pi$, π maintains ψ, and $s' \models \phi$. By construction of $Tabu(v)$ in RealizePlanProg (lines 18–21), $s' \notin Tabu(v)$, since any domain state s can be in $Tabu(v)$ only if there exists a transition outgoing from v, with its guard holding in s, that cannot be realized from s, which, by Definition 4, cannot be the case for $s = s'$. Since the usage of $Tabu(v)$ in subroutine Plan prevents the generation of any plan reaching an end state $s \in Tabu(v)$, $s' \notin Tabu(v)$, and Plan can generate a valid plan for every solvable planning problem in the input domain (Plan is complete), Plan cannot generate `failure` when it realizes transition d from $\langle s_0, v_0 \rangle$ (lines 10–11 of RealizePlanProg). Thereby, in line 18 of RealizePlanProg, $\pi \neq \texttt{failure}$ when $\langle s, v \rangle = \langle s_0, v_0 \rangle$, and RealizePlanProg cannot terminate returning `failure` (line 19). Then, by Lemma 1, RealizePlanProg terminates returning a realization for $\mathcal{P}$.

Assume that there exists no valid realization for the underlying planning program. By Lemma 1, RealizePlanProg terminates. By Theorem 6, if RealizePlanProg terminated returning a realization it would be valid, but this would contradict the assumption that there exists no valid realization. Thereby, RealizePlanProg terminates returning `failure`. □

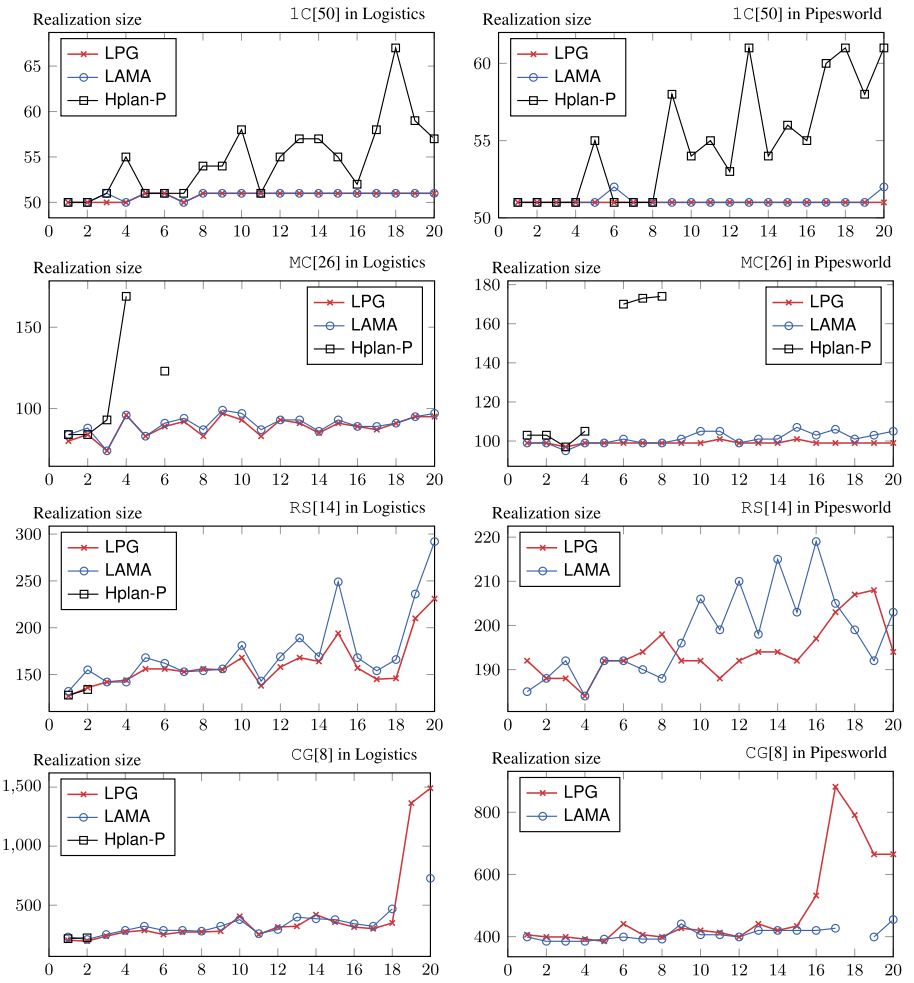


Fig. C.2. Realization size for benchmark SM_{50}.

Proof of Theorem 8 (page 79). (1) Let π be a valid plan for Π. A valid plan π' for Π' can be obtained by appending to π action `Ignore-pref` in A_P and, subsequently, a sequence of actions formed by one action in `Act-tabu`(s) for each TES s. The goal ϕ of Π achieved by subplan π of π' remains satisfied at the end state of π' because actions in A_P and A_T do not delete any proposition in the proposition set P of the domain of Π. Action `Ignore-pref` $\in A_P$ is executable at the end of subplan π, because its precondition `normal-mode` holds in the initial state of Π', and it is not deleted by the actions in A^+ forming π; `Ignore-pref` satisfies goal `check-pref` of Π', because it is an additive effect of the action, and it is not deleted by the actions in A_T. Since π is a valid plan, for each TES s, $s \neq last(\pi)$ holds, and hence there exists $p \in P$ such that either p is false in s and true in $last(\pi)$, or p is true in s and false in $last(\pi)$. Therefore, for each conjunct $g =$ `not-tabu`(s) of the achievement goal formula ϕ' of Π', there exists an action a in `Act-tabu`(s) that is executable after the execution of `Ignore-pref` in $last(\pi)$ and achieves g because *(i)* precondition `end-mode` of a is added by `Ignore-pref` and it is deleted by no action in A_T, and *(ii)* by construction of A_T, the other precondition of a holds in $last(\pi)$ and no other action in A_T can delete such a precondition. Moreover, plan π' maintains the maintenance goal ψ of Π and Π', because π maintains ψ and no action in $A_P \cup A_T$ can make it false. It follows that there exists a valid plan solving the translated problem Π' that is formed by π followed by an action in A_P (`Ignore-pref`) and a sequence of actions in A_T.

(2) Since π' is a valid plan for Π', $\phi' \models \phi$ and the actions in A_P and A_T do not add propositions of P, the subplan π of π' formed by the actions in A^+ satisfies the achievement goal of Π, as well as the maintenance goal of Π. Moreover, since π' is valid, for each TES s, plan π' contains at least one action $a \in$ `Act-tabu`(s) achieving conjunct `not-tabu`(s) of goal formula ϕ'. By construction of A_T and `Act-tabu`(s), since all actions in π' are executable and the actions in A_P and A_T do not add/delete propositions of P, $last(\pi)$ must be different from any TES of Π. Hence, by removing precondition `normal-mode` from the actions in π', we obtain a plan solving Π. □

Proof of Theorem 9 (page 80). (1) Let π be a valid plan for Π ending in a PES of Π. A valid plan π' for problem Π' such that $c(\pi') = 0$ can be obtained by appending to π action `Sat-pref`(s) and, subsequently, a sequence of actions formed by

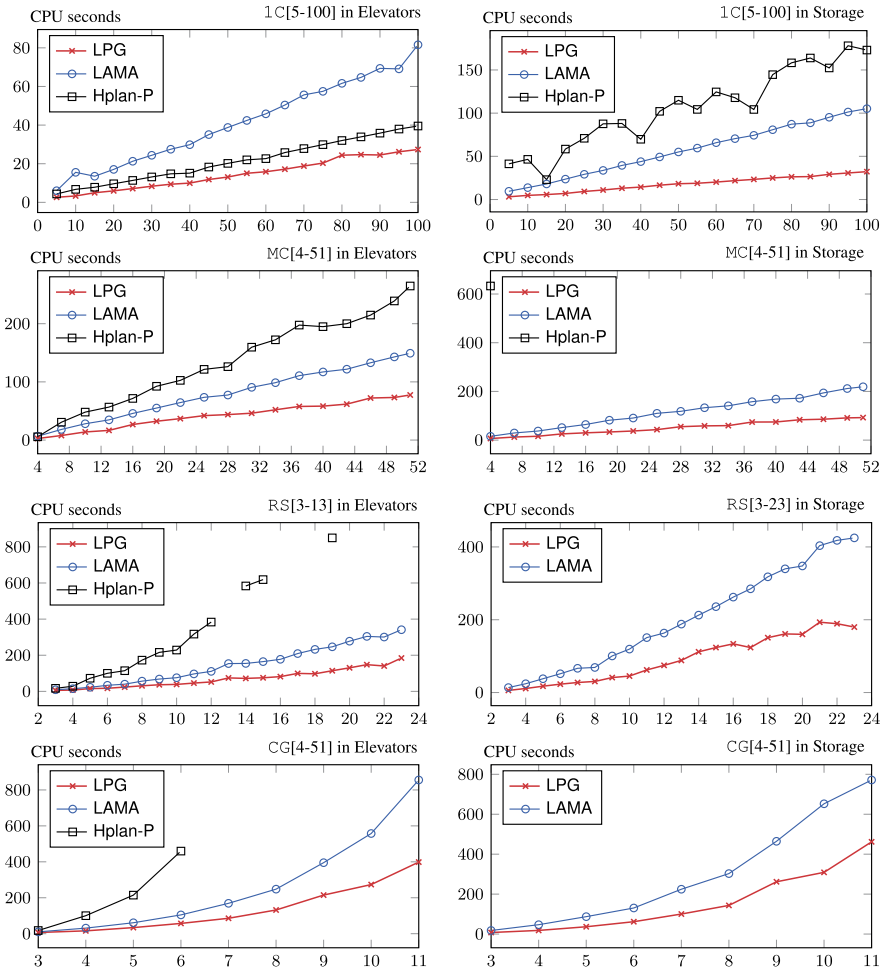


Fig. C.3. CPU time for benchmark S_{5-100}.

one action in Act-tabu(s) for each TES s. Action Sat-pref(s) preserves the maintenance goal of Π' and is executable at the end of subplan π, because Sat-pref(s) has no effect in the proposition set of the domain of Π, precondition normal-mode of Sat-pref(s) holds in the initial state of Π' and is not deleted by the actions in A^+ forming π, and, since s is a PES of Π, by construction of Sat-pref(s), the other preconditions of Sat-pref(s) hold in $last(\pi)$. Moreover, Sat-pref(s) satisfies conjunct check-pref of the achievement goal formula ϕ' of Π', because it is an additive effect of the action, and it is not deleted by the actions in A_T. For each conjunct $g =$ not-tabu(s) of ϕ', there exists an action $a \in$ Act-tabu(s) that achieves g and is executable after the execution of Sat-pref(s) in $last(\pi)$, because *(i)* precondition end-mode of a is added by Sat-pref(s) and is deleted by no action in A_T, and *(ii)* by construction of A_T, the other precondition of a holds in $last(\pi)$ and no other action in A_T can delete such a precondition. Moreover, every action in Act-tabu(s) preserves the maintenance goal of Π and Π'. Therefore, plan π' is valid, and, since the cost of every action of π' is zero, $c(\pi') = 0$.

(2) By Theorem 8, the subplan π obtained from π' by removing the actions in A_P and A_T and precondition normal-mode is valid for Π. Since $c(\pi') = 0$ and π' is valid, π' contains an action Sat-pref(s) achieving the goal conjunct check-pref of ϕ', for some PES s of Π. By construction of action set A^+ and action Sat-pref(s), Sat-pref(s) can be executed only as the first action after the end of subplan π. Moreover, by construction of Sat-pref(s) and since π' is valid, subplan π must end in a PES of Π. □

Appendix C. Additional experimental results

Figs. C.1 and C.2 show the CPU time and the program realization size of RealizePlanProg using LPG, LAMA and Hplan-P with PESs for planning programs with domain Logistics and Pipesworld and δ equal to 1C[50], MC[26], RS[14] and CG[8] (s.t. $|\delta|$ is about 50). The x-axis of the graphs in these appendixes refers to the program number (higher program numbers correspond to programs with domains that have larger sizes). Figs. C.3 and C.4 show the CPU time and the program realization size for planning programs with instances of domain Elevators based on 9 objects, instances of domain

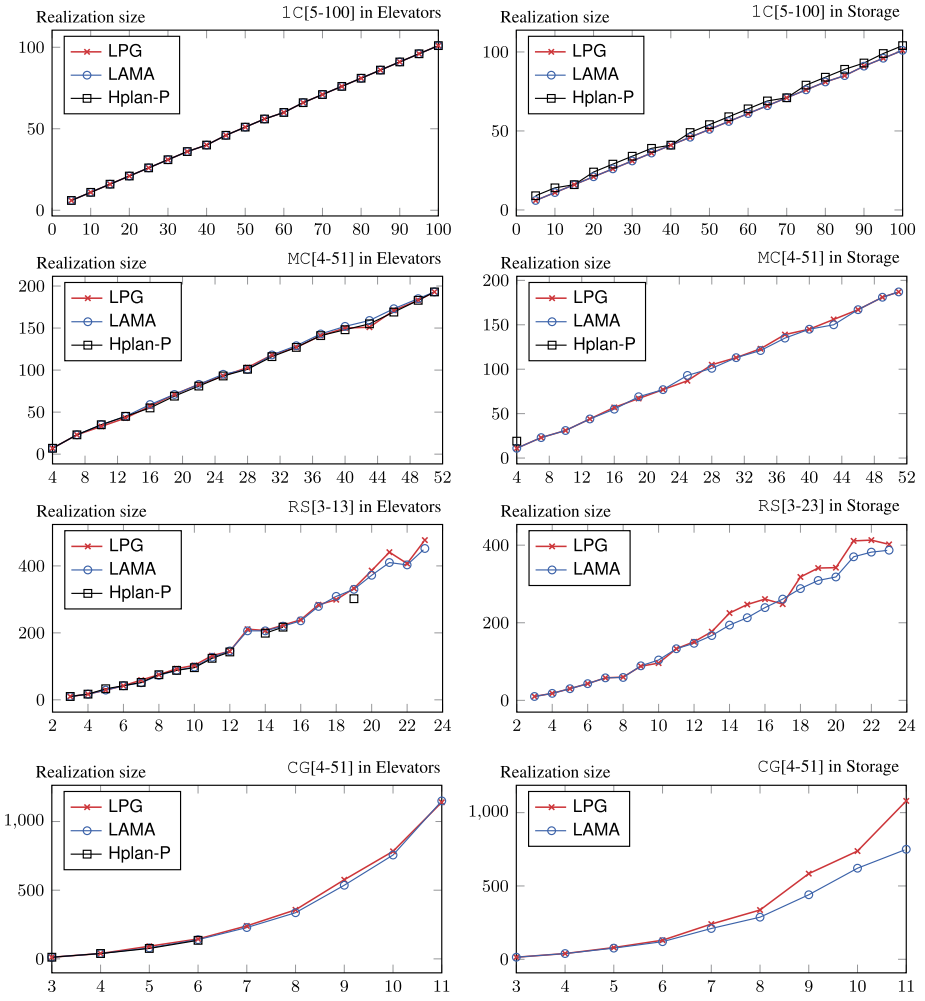


Fig. C.4. Realization size for benchmark S_{5-100}.

`Storage` based on 25 objects, and δ equal to `1C`[5-100], `MC`[4-51], `RS`[3-23] and `CG`[3-11] (s.t. $|\delta|$ ranges from about 5 to 100). The x-axis of the graphs in these latter appendixes refers the number of program states. The fact that Hplan-P does not appear in a graph means that it realizes no planning program among those evaluated in the graph.

References

[1] R. Alur, S.L. Torre, Deterministic generators and games for LTL fragments, ACM Trans. Comput. Log. 5 (1) (2004) 1–25.
[2] F. Bacchus, The AIPS'00 planning competition, AI Mag. 22 (2001) 47–56.
[3] F. Bacchus, F. Kabanza, Using temporal logics to express search control knowledge for planning, Artif. Intell. 116 (1–2) (2000) 123–191.
[4] C. Bäckström, B. Nebel, Complexity results for SAS+ planning, Comput. Intell. 11 (4) (1995) 1–34.
[5] J.A. Baier, C. Fritz, S.A. McIlraith, Exploiting procedural domain control knowledge in state-of-the-art planners, in: Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS, 2007, pp. 26–33.
[6] J.A. Baier, F. Bacchus, S.A. McIlraith, A heuristic search approach to planning with temporally extended preferences, Artif. Intell. 173 (5–6) (2009) 593–618.
[7] F. Belardinelli, A. Lomuscio, F. Patrizi, An abstraction technique for the verification of artifact-centric systems, in: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR), 2012, pp. 319–328.
[8] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella, Automatic composition of e-Services that export their behavior, in: Proceedings of the International Conference on Service Oriented Computing, ICSOC, 2003, pp. 43–58.
[9] P. Bertoli, M. Pistore, P. Traverso, Automated composition of web services via planning in asynchronous domains, Artif. Intell. 174 (3–4) (2010) 316–361.
[10] R. Bloem, A. Cimatti, K. Greimel, G. Hofferek, R. Könighofer, M. Roveri, V. Schuppan, R. Seeber, Ratsy – a new requirements analysis tool with synthesis, in: Proceedings of the International Conference on Computer Aided Verification, CAV, 2010, pp. 425–429.
[11] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, Y. Sa'ar, Synthesis of reactive(1) designs, J. Comput. Syst. Sci. 78 (3) (2012) 911–938.
[12] A.L. Blum, M.L. Furst, Fast planning through planning graph analysis, Artif. Intell. 90 (1–2) (1997) 281–300.
[13] B. Bonet, H. Palacios, H. Geffner, Automatic derivation of memoryless policies and finite-state controllers using classical planners, in: Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS, 2009, pp. 190–197.
[14] M.E. Bratman, Intentions, Plans, and Practical Reason, Harvard University Press, 1987.

[15] M.E. Bratman, D.J. Israel, M.E. Pollack, Plans and resource-bounded practical reasoning, Comput. Intell. 4 (3) (1988) 349–355.
[16] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang, Symbolic model checking: 10^{20} states and beyond, Inf. Comput. 98 (2) (1992) 142–170.
[17] R. Cavada, A. Cimatti, M. Roveri, V. Schuppan, A. Tchaltsev, NuGaT game solver home page, https://es.fbk.eu/technologies/nugat-game-solver, 2010.
[18] M. Cavazza, F. Charles, S.J. Mead, Character-based interactive storytelling, IEEE Intell. Syst. 17 (4) (2002) 17–24.
[19] L. Ceriani, A. Gerevini, Planning with always preferences and soft goals by compilation into STRIPS with action costs, in: Proceedings of the 8th International Annual Symposium on Combinatorial Search, 2015, pp. 161–165.
[20] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV 2: an opensource tool for symbolic model checking, in: Proceedings of the International Conference on Computer Aided Verification, CAV, 2002, pp. 359–364.
[21] A. Cimatti, E.M. Clarke, F. Giunchiglia, M. Roveri, NUSMV: a new symbolic model checker, Int. J. Softw. Tools Technol. Transf. (STTT) 2 (4) (2000) 410–425.
[22] A. Cimatti, M. Roveri, P. Traverso, Automatic OBDD-based generation of universal plans in non-deterministic domains, in: Proceedings of the National Conference on Artificial Intelligence, AAAI, 1998, pp. 875–881.
[23] P.R. Cohen, H.J. Levesque, Intention is choice with commitment, Artif. Intell. 42 (1990) 213–261.
[24] M. Dastani, 2APL: a practical agent programming language, Auton. Agents Multi-Agent Syst. 16 (3) (Jun. 2008) 214–248.
[25] M. Dastani, B. van Riemsdijk, J.-J. Meyer, Goal types in agent programming, in: Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS, 2006, pp. 1285–1287.
[26] F.S. de Boer, K.V. Hindriks, W. van der Hoek, J.-J. Meyer, A verification framework for agent programming with declarative goals, J. Appl. Log. 5 (2) (2007) 277–302.
[27] G. De Giacomo, C. Di Ciccio, P. Felli, Y. Hu, M. Mecella, Goal-based composition of stateful services for smart homes, in: On the Move to Meaningful Internet Systems: OTM 2012, Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2012, 2012, pp. 194–211.
[28] G. De Giacomo, Y. Lespérance, H.J. Levesque, ConGolog, a concurrent programming language based on the situation calculus, Artif. Intell. 121 (1–2) (2000) 109–169.
[29] G. De Giacomo, Y. Lespérance, H.J. Levesque, S. Sardina, IndiGolog: a high-level programming language for embedded reasoning agents, in: R.H. Bordini, M. Dastani, J. Dix, A.E. Fallah-Seghrouchni (Eds.), Multi-Agent Programming: Languages, Platforms and Applications, Springer, 2009, pp. 31–72 (ch. 2).
[30] G. De Giacomo, Y. Lespérance, F. Patrizi, Bounded situation calculus action theories and decidable verification, in: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR), 2012, pp. 467–477.
[31] G. De Giacomo, Y. Lespérance, F. Patrizi, S. Vassos, LTL verification of online executions with sensing in bounded situation calculus, in: Proceedings of the European Conference in Artificial Intelligence, ECAI, 2014, pp. 369–374.
[32] G. De Giacomo, Y. Lespérance, F. Patrizi, S. Vassos, Progression and verification of situation calculus agents with bounded beliefs, in: Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS, 2014, pp. 141–148.
[33] G. De Giacomo, F. Patrizi, P. Felli, S. Sardina, Two-player game structures for generalized planning and agent composition, in: Proceedings of the National Conference on Artificial Intelligence, AAAI, 2010, pp. 297–302.
[34] G. De Giacomo, F. Patrizi, S. Sardina, Agent programming via planning programs, in: Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS, 2010, pp. 491–498.
[35] G. De Giacomo, F. Patrizi, S. Sardina, Automatic behavior composition synthesis, Artif. Intell. 196 (2013) 106–142.
[36] G. De Giacomo, M.Y. Vardi, Automata-theoretic approach to planning for temporally extended goals, in: Proceedings of the European Conference on Planning, ECP, in: Lecture Notes in Computer Science, vol. 1809, Springer, 1999, pp. 226–238.
[37] O. Despouys, F.F. Ingrand, Propice-plan: toward a unified framework for planning and execution, in: Proceedings of the European Conference on Planning, ECP, in: Lecture Notes in Computer Science, vol. 1809, Springer, 1999, pp. 278–293.
[38] E.W. Dijkstra, A Discipline of Programming, Prentice Hall, 1976.
[39] J. Dix, H. Muñoz-Avila, D.S. Nau, L. Zhang, IMPACTing SHOP: putting an AI planner into a multi-agent environment, Ann. Math. Artif. Intell. 37 (4) (2003) 381–407.
[40] S. Edelkamp, On the compilation of plan constraints and preferences, in: Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS, 2006, pp. 374–377.
[41] E.A. Emerson, Temporal and modal logic, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B, MIT Press, 1990, pp. 995–1072.
[42] R.W. Floyd, Nondeterministic algorithms, J. ACM 14 (4) (1967) 636–644.
[43] A. Gerevini, P. Haslum, D. Long, A. Saetti, Y. Dimopoulos, Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners, Artif. Intell. 173 (5–6) (2009) 619–668.
[44] A. Gerevini, F. Patrizi, A. Saetti, An effective approach to realizing planning programs, in: Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS, 2011, pp. 323–326.
[45] A. Gerevini, A. Saetti, I. Serina, Planning through stochastic local search and temporal action graphs, J. Artif. Intell. Res. (JAIR) 20 (2003) 239–290.
[46] M. Ghallab, D.S. Nau, P. Traverso, Automated Planning: Theory and Practice, Morgan Kaufmann Publishers Inc., May 2004.
[47] M. Ghallab, D.S. Nau, P. Traverso, The actor's view of automated planning and acting: a position paper, Artif. Intell. 208 (2014) 1–17.
[48] M.L. Ginsberg, Universal planning: an (almost) universally bad idea, AI Mag. 10 (1989) 41–44.
[49] E. Grädel, W. Thomas, T. Wilke (Eds.), Automata, Logics, and Infinite Games: A Guide to Current Research, Lecture Notes in Computer Science (LNCS), vol. 2500, Springer, 2002.
[50] K.H. Hall, R.J. Staron, P. Vrba, Experience with holonic and agent-based control systems and their adoption by industry, in: Holonic and Multi-Agent Systems for Manufacturing, in: Lecture Notes in Computer Science (LNCS), vol. 3593, Springer, 2005, pp. 1–10.
[51] B.B. Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, M. Montali, Verification of relational data-centric dynamic systems with external services, in: ACM Symposium on Principles of Database Systems, PODS, 2013, pp. 163–174.
[52] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, E. Jansen, The Gator Tech Smart House: a programmable pervasive space, Computer 38 (3) (2005).
[53] M. Helmert, The fast downward planning system, J. Artif. Intell. Res. (JAIR) 26 (2006) 191–246.
[54] M. Helmert, M. Do, I. Refanidis (Eds.), Sixth International Planning Competition IPC6: Deterministic Part, 2008, http://ipc.informatik.uni-freiburg.de/.
[55] C.A. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969).
[56] J. Hoffmann, S. Edelkamp, The deterministic part of IPC-4: an overview, J. Artif. Intell. Res. (JAIR) 24 (2005) 519–579.
[57] J. Hoffmann, B. Nebel, The FF planning system: fast plan generation through heuristic search, J. Artif. Intell. Res. (JAIR) 14 (2001) 253–302.
[58] J.F. Hübner, R.H. Bordini, M. Wooldridge, Programming declarative goals using plan patterns, in: Proceedings of the International Workshop on Declarative Agent Languages and Technologies (DALT), in: Lecture Notes in Computer Science (LNCS), vol. 4327, Springer, 2006, pp. 123–140.
[59] S. Jiménez, A. Coles (Eds.), Seventh International Planning Competition IPC7: Learning Part, 2011, http://www.plg.inf.uc3m.es/ipc2011-learning.
[60] B. Jobstmann, S. Galler, M. Weiglhofer, R. Bloem, Anzu: a tool for property synthesis, in: Proceedings of the International Conference on Computer Aided Verification, CAV, 2007, pp. 258–262.
[61] F. Kabanza, S. Thiébaux, Search control in planning for temporally extended goals, in: Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS, 2005, pp. 130–139.

[62] E. Keyder, H. Geffner, Soft goals can be compiled away, J. Artif. Intell. Res. (JAIR) 36 (2009) 547–556.
[63] J. Koehler, B. Nebel, J. Hoffmann, Y. Dimopoulos, Extending planning graphs to an ADL subset, Technical Report 88, Institut für Informatik, Freiburg, Germany, 1997.
[64] U. Kuter, D.S. Nau, E. Reisner, R.P. Goldman, Using classical planners to solve nondeterministic planning problems, in: Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS, 2008, pp. 190–197.
[65] U.D. Lago, M. Pistore, P. Traverso, Planning with a language for extended goals, in: Proceedings of the National Conference on Artificial Intelligence, AAAI, 2002, pp. 447–454.
[66] Y. Lespérance, H.J. Levesque, F. Lin, D. Marcu, R. Reiter, R.B. Scherl, Foundations of a logical approach to agent programming, in: Proceedings of the International Workshop on Agent Theories, Architectures, and Languages, ATAL, 1995, pp. 331–346.
[67] H.J. Levesque, R. Reiter, High-level robotic control: beyond planning. A position paper, in: AIII 1998 Spring Symposium: Integrating Robotics Research: Taking the Next Big Leap, 1998, pp. 106–108.
[68] H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin, R.B. Scherl, GOLOG: a logic programming language for dynamic domains, J. Log. Program. 31 (1997) 59–84.
[69] M.L. Littman, Probabilistic propositional planning: representations and complexity, in: Proceedings of the National Conference on Artificial Intelligence, AAAI, 1997, pp. 748–754.
[70] D. Long, M. Fox, The 3rd international planning competition: results and analysis, J. Artif. Intell. Res. (JAIR) 20 (2003) 1–59.
[71] D. McDermott, The 1998 AI planning systems competition, AI Mag. 21 (2000) 35–55.
[72] S.A. McIlraith, T.C. Son, Adapting Golog for composition of semantic web service, in: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR), 2002, pp. 482–493.
[73] K.L. McMillan, Symbolic model checking – an approach to the state explosion problem, Ph.D. thesis, Carnegie Mellon University, 1992, TR CMU-CS-92-131.
[74] B. Meyer, Applying "design by contract", IEEE Comput. 25 (10) (1992) 40–51.
[75] R. Milner, An algebraic definition of simulation between programs, in: Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI, 1971, pp. 481–489.
[76] A. Muscholl, I. Walukiewicz, A lower bound on web services composition, in: Proceedings of the International Conference on Foundations of Software Science and Computation Structures, FoSSaCS, in: Lecture Notes in Computer Science (LNCS), vol. 4423, Springer, 2007, pp. 274–286.
[77] B. Nebel, J. Koehler, Plan reuse versus plan generation: a theoretical and empirical analysis, Artif. Intell. 76 (1–2) (1995) 427–454.
[78] M. Paolucci, D. Kalp, A. Pannu, O. Shehory, K. Sycara, A planning component for RETSINA agents, in: Proceedings of the International Workshop on Agent Theories, Architectures, and Languages, ATAL, 1999, pp. 147–161.
[79] M. Pistore, P. Traverso, Planning as model checking for extended goals in non-deterministic domains, in: Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI, 2001, pp. 479–484.
[80] N. Piterman, A. Pnueli, Y. Sa'ar, Synthesis of reactive(1) designs, in: Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI, in: Lecture Notes in Computer Science (LNCS), vol. 3855, Springer, 2006, pp. 364–380.
[81] A. Pnueli, The temporal logic of programs, in: Proceedings of the Annual Symposium on Foundations of Computer Science, FOCS, 1977, pp. 46–57.
[82] A. Pnueli, R. Rosner, On the synthesis of a reactive module, in: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL, 1989, pp. 179–190.
[83] A. Pnueli, Y. Sa'ar, L.D. Zuck, JTLV: a framework for developing verification algorithms, in: Proceedings of the International Conference on Computer Aided Verification, CAV, 2010, pp. 171–174.
[84] A. Pnueli, E. Shahar, A platform for combining deductive with algorithmic verification, in: Proceedings of the International Conference on Computer Aided Verification, CAV, 1996, pp. 184–195.
[85] J. Porteous, M. Cavazza, F. Charles, Narrative generation through characters' point of view, in: Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS, 2010, pp. 1297–1304.
[86] M. Ramirez, N. Yadav, S. Sardina, Behavior composition as fully observable non-deterministic planning, in: Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS, 2013, pp. 180–188.
[87] A.S. Rao, Agentspeak(L): BDI agents speak out in a logical computable language, in: Proceedings of the European Workshop on Modeling Autonomous Agents in a Multi-Agent World (Agents Breaking Away), in: Lecture Notes in Computer Science (LNCS), vol. 1038, Springer, 1996, pp. 42–55.
[88] R. Reiter, Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems, The MIT Press, 2001.
[89] S. Richter, M. Westphal, The LAMA planner: guiding cost-based anytime planning with landmarks, J. Artif. Intell. Res. (JAIR) 39 (2010) 127–177.
[90] J. Rintanen, Complexity of planning with partial observability, in: Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS, 2004, pp. 345–354.
[91] S. Sardina, L. Padgham, A BDI agent programming language with failure recovery, declarative goals, and planning, Auton. Agents Multi-Agent Syst. 23 (1) (2011) 18–70.
[92] M.J. Schoppers, Universal plans for reactive robots in unpredictable environments, in: Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI, 1987, pp. 1039–1046.
[93] D. Shaparau, M. Pistore, P. Traverso, Fusing procedural and declarative planning goals for nondeterministic domains, in: Proceedings of the National Conference on Artificial Intelligence, AAAI, 2008, pp. 983–990.
[94] V. Shivashankar, R. Alford, U. Kuter, D. Nau, The GoDeL planning system: a more perfect union of domain-independent and hierarchical planning, in: Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI, AAAI Press, 2013, pp. 2380–2386.
[95] V. Shivashankar, U. Kuter, D. Nau, R. Alford, Hierarchical goal-based formalism and algorithm for single-agent planning, in: Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS, 2012, pp. 981–988.
[96] V. Shivashankar, U. Kuter, D. Nau, Hierarchical goal network planning: initial results, Technical Report CS-TR-4983, University of Maryland, Freiburg, Germany, 2011.
[97] Y. Shoham, An overview of agent-oriented programming, in: J.M. Bradshaw (Ed.), Software Agents, The MIT Press, 1997, pp. 271–290.
[98] S. Srivastava, N. Immerman, S. Zilberstein, A new representation and associated algorithms for generalized planning, Artif. Intell. 175 (2) (2011) 615–647.
[99] W.M.P. van der Aalst, A.H.M. ter Hofstede, M. Weske, Business process management: a survey, in: Proceedings of the International Conference on Business Process Management, BPM, 2003, pp. 1–12.
[100] B. van Riemsdijk, M. Dastani, F. Dignum, J.-J. Meyer, Dynamics of declarative goals in agent programming, in: Proceedings of the International Workshop on Declarative Agent Languages and Technologies (DALT), in: Lecture Notes in Computer Science (LNCS), vol. 3476, Springer, 2005, pp. 1–18.
[101] B. van Riemsdijk, M. Dastani, J.-J. Meyer, Semantics of declarative goals in agent programming, in: Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS, 2005, pp. 133–140.
[102] M.Y. Vardi, An automata-theoretic approach to linear temporal logic, in: Logics for Concurrency: Structure Versus Automata, in: Lecture Notes in Computer Science (LNCS), vol. 1043, Springer, 1996, pp. 238–266.
[103] A. Walczak, L. Braubach, A. Pokahr, W. Lamersdorf, Augmenting BDI agents with deliberative planning techniques, in: Proceedings of the Programming Multiagent Systems Languages, Frameworks, Techniques and Tools Workshop, PROMAS, 2006, pp. 113–127.

[104] D.E. Wilkins, K.L. Myers, A common knowledge representation for plan generation and reactive execution, J. Log. Comput. 5 (6) (1995) 731–761.
[105] D.E. Wilkins, K.L. Myers, J.D. Lowrance, L.P. Wesley, Planning and reacting in uncertain and dynamic environments, J. Exp. Theor. Artif. Intell. 7 (1) (1995) 197–227.
[106] B.C. Williams, M.D. Ingham, S.H. Chung, P.H. Elliott, Model-based programming of intelligent embedded systems and robotic space explorers, Proc. IEEE 91 (1) (2003) 212–237, Special Issue on Modeling and Design of Embedded Software.
[107] P. Wolper, On the relation of programs and computations to models of temporal logic, in: Temporal Logic in Specification, 1987, pp. 75–123.