Agent Programming via Planning Programs

Giuseppe De Giacomo and Fabio Patrizi
Dipartimento di Informatica e Sistemistica
Sapienza Università di Roma
Roma, Italy
{patrizi,degiacomo}@dis.uniroma1.it

Sebastian Sardina
School of Computer Science and IT
RMIT University
Melbourne, Australia
sebastian.sardina@cs.rmit.edu.au

ABSTRACT

We imagine agent "planning" programs as programs built from achievement and maintenance goals. Their executions require the ability to meet such goals while respecting the programs' control flow. The question then is: can we always guarantee the execution of such programs? In this paper, we define this novel planning-programming problem formally, and propose a sound, complete and optimal wrt computational complexity technique to actually generate a solution by appealing to recent results in LTL-based *synthesis* of reactive systems.

Categories and Subject Descriptors

I.12.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods

General Terms

Theory, Verification, Algorithms

Keywords

Agent programming, planning, synthesis, LTL, model checking

1. INTRODUCTION

Agent planning programs are agent programs whose atomic instructions are requests for *achieving* a goal while *maintaining* some conditions. They come from merging two traditions in AI research: automated planning and agent-oriented programming.

Automated planning [4] allows the specification of behavior in a *declarative* manner, thus providing an abstract, flexible, and powerful mechanism that caters for *flexible* behavior: any conceivable way of achieving the desired outcome may be constructed. On the other hand, agent-oriented programming [18, 8, 9, 13] accommodates useful "*know-how*" domain knowledge encoding the typical operations of the domain, allowing agent systems to better focus their reasoning and "act as they go." Interestingly, the advantages of each of the two approaches are the weaknesses of the other. Automated planning is intrinsically difficult computationally, especially in its advanced forms [4, 14], since plans are built from first-principle, and it is not tailored for long-term behavior in changing domains, where the actual behavior depends on contingencies. Agent-oriented approaches, on the other hand, typically rely entirely on procedural knowledge that ought to be crafted at design time: no "new" plans can be generated.

Cite as: Agent Programming via Planning Programs, De Giacomo, G., Patrizi, F., and Sardina, S., *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lespérance, Luck and Sen (eds.), May, 10–14, 2010, Toronto, Canada, pp. XXX-XXX.

In this paper, through the notion of *agent planning programs*, we propose a novel account that mixes programming with planning, thus leveraging on the advantages of both approaches. In concrete, we assume a dynamic domain $\mathcal{D}$ describing the dynamics of the world as usual in planning and reasoning about actions, and an agent planning program $\mathcal{T}$ that is meant to be *"realized"* in $\mathcal{D}$. Technically, the program $\mathcal{T}$ is a transition system in which states represent *choice points* and transitions specify pairs of *maintenance* and *achievement* goals that the agent may decide to follow at each step of the program. Namely, at any point in time, the domain and the program are in some states, and the agent decides, autonomously, which program transition to request, thus specifying a goal to achieve next (e.g., be at work), and at the same time some maintenance constraints (e.g., never run out of fuel).

In order to actually realize a transition in the program, a plan needs to be synthesized. At execution time, each requested transition activates the execution of the corresponding plan. The key point is that after the execution of the plan, a new request is issued and a plan for it must be available. In other words, in synthesizing the plan for the transitions we need to take into account that the resulting state of the domain must allow for the execution of the plans corresponding to the next transition, and so on, possibly forever.

Under this framework, the problem of concern is the following: *can an agent planning program be realized in the dynamic domain?* That is, *can each of the possible options in the program that the agent may autonomously choose be always guaranteed?*

Technically, agent planning problem cannot be realized by resorting to planning techniques developed within the planning community. Solving, that is, realizing, such planning programs requires temporally extended plans that loop and possibly do not even terminate, analogously to [6, 15]. To synthesize such plans, we resort to synthesis techniques developed for Linear-time Temporal Logics (LTL) [12, 11].

When it comes to building agents, the advantages of using agent planning programs are twofold. First, agents can be thought and designed in terms of declarative goals, without the need to specify the detailed procedural information needed on how to bring about the agent's goals. As already accepted in the literature, declarative goals provide several 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 [19]. Second, the declarative goals can be combined to encode the dynamics and relations among such goals that the agent designer/programmer may have available. More concretely, the target program restricts the options that will be available next once the (current) goal is brought about. This provides indeed a way of specifying procedural knowledge of the domain, but at a higher-level of abstraction than what typical agent programming languages do.

The rest of the paper is structured as follows. In Section 2 and 3, we introduce agent planning programs, the corresponding realization problem and a full fledged example. In Section 4, we briefly review some results on LTL synthesis via model checking of game structures. In Section 5, we develop a sound, complete, and optimal (from the computational complexity point of view) technique for realizing agent planning programs exploiting such results. In Section 6, we consider the case in which agents may act on the domain only through available actuators/devices. We show that this sophisticated extension can be reduced to the basic setting. We conclude the paper in Section 7 with a brief discussion.

2. THE FRAMEWORK

Our framework consists of two main ingredients: *(i)* a (possibly nondeterministic) *dynamic domain*, formalizing the environment that the agent acts in; and *(ii)* an *agent planning program*, providing a high-level representation of the desired domain evolutions.

Definition 1 (Dynamic Domain). A *dynamic domain*, or *environment*, is a tuple $\mathcal{D} = \langle P, A, S_0, \rho \rangle$, where:

- $P = \{p_1, \ldots, p_n\}$ is a finite set of *domain propositions*. A *state* is a subset of 2^P;
- $A = \{a_1, \ldots, a_r\}$ is the finite set of *domain actions*;
- $S_0 \in 2^P$ is the *initial state*;
- $\rho \subseteq 2^P \times A \times 2^P$ is the *transition relation*. We freely interchange notations $\langle S, a, S' \rangle \in \rho$ and $S \xrightarrow{a} S'$ in $\mathcal{D}$. ■

A $\mathcal{D}$-*history* is a finite sequence of the form $\tau = S^0 \xrightarrow{a^1} S^1 \cdots S^{\ell-1} \xrightarrow{a^\ell} S^\ell$ such that *(i)* $S^i \in 2^P$ for $i \in \{0, \ldots, \ell\}$; and *(ii)* $S^i \xrightarrow{a^{i+1}} S^{i+1}$ in $\mathcal{D}$, for each $i \in \{0, \ldots, \ell - 1\}$. Informally, $\mathcal{D}$-histories stand for the possible evolutions of $\mathcal{D}$ starting from a state S^0. The set of all possible $\mathcal{D}$-histories is denoted by $\mathcal{H}$.

Given a dynamic domain $\mathcal{D}$, a general plan is a (possibly partial) function $\pi : \mathcal{H} \mapsto A$ that outputs an action given a $\mathcal{D}$-history. For finite sequences $\tau = S^0 \xrightarrow{a^1} S^1 \cdots S^{\ell-1} \xrightarrow{a^\ell} S^\ell$, we define $|\tau| \doteq \ell + 1$, and for infinite ones, $|\tau| \doteq \infty$. Given a (finite or infinite) sequence $\tau = S^0 \xrightarrow{a^1} S^1 \xrightarrow{a^2} \cdots$, we denote, for $0 < k < |\tau| + 1$, its k-length finite prefix as $\tau|_k = S^0 \xrightarrow{a^1} \cdots \xrightarrow{a^{k-1}} S^{k-1}$. An execution of a general plan π from a state $S \in 2^P$ is a, possibly infinite, sequence $\tau = S^0 \xrightarrow{a^1} S^1 \xrightarrow{a^2} \cdots$ such that *(i)* $S^0 = S$; *(ii)* $\tau|_k$ is a $\mathcal{D}$-history, for all $0 < k < |\tau| + 1$; and *(iii)* $a^k = \pi(\tau|_k)$, for all $0 < k < |\tau|$.

Observe that an execution τ of a general plan can be infinite, i.e., $|\tau| = \infty$. When all possible executions of a general plan are finite, the plan is called a conditional plan. The set of all conditional plans over $\mathcal{D}$ is referred to as Π. Note that, being finite, executions of conditional plans are $\mathcal{D}$-histories. A finite execution τ such that $\pi(\tau)$ is undefined is a complete execution—the execution cannot be extended further. In the following, we shall only consider conditional plans and refer to them simply as "plans."

Next, we introduce the second component in our framework, namely, *agent planning programs*, which are meant to be high-level specifications of *desired* agent behaviors in terms of *declarative goals*.

Definition 2 (Agent Planning Program). An *agent planning program*, or simply a *planning program*, for a dynamic domain $\mathcal{D}$ is a tuple $\mathcal{T} = \langle T, \mathcal{G}, t_0, \delta \rangle$, where:

- $T = \{t_0, \ldots, t_q\}$ is the finite set of *program states*;
- $\mathcal{G}$ is a finite set of goals of the form *achieve ϕ while maintaining ψ*, denoted by pairs $g = \langle \psi, \phi \rangle$, where ψ and ϕ are propositional formulae over P;
- $t_0 \in T$ is the *program initial state*;
- $\delta \subseteq T \times \mathcal{G} \times T$ is the *transition relation*. We freely interchange notations $\langle t, g, t' \rangle \in \delta$ and $t \xrightarrow{g} t'$ in $\mathcal{T}$. ■

When an agent planning program is realized, a typical session is as follows: at any point in time, the planning program is in a state t and the environment in a state $S \in 2^P$ (initially, states t_0 and S_0, respectively); the agent requests a transition $t \xrightarrow{\langle \psi, \phi \rangle} t'$ in $\mathcal{T}$; then, a plan π is executed from S which eventually leads the environment to a state that satisfies achievement goal ϕ, while only traversing states satisfying maintenance goal ψ; upon plan completion, the agent planning program moves to t' and requests a new transition $t' \xrightarrow{\langle \psi', \phi' \rangle} t''$ in $\mathcal{T}$, and so on. Notice that, at any point in time, all possible choices available in the agent planning program must be guaranteed by the system—every legal request needs to be satisfied.

Next, we formalize agent planning program's semantics. Concretely, we shall define when a planning program is *realizable* in a domain, that is, when a agent planning program can always be executed by continuously fulfilling the agent's requests in the domain. To do so, we first need to introduce some technical notions.

We say that a $\mathcal{D}$-history $\tau = S^0 \xrightarrow{a_1} S^1 \cdots S^{\ell-1} \xrightarrow{a_\ell} S^\ell$ *achieves* goal ϕ if $S^\ell \models \phi$. Similarly, τ *maintains* goal ψ if $S^i \models \psi$, for every $i \in \{0, \ldots, \ell - 1\}$. Such notions can be extended to conditional plans in a straightforward manner. A conditional plan π *achieves* goal ϕ from state S if all of its complete executions from S do so; and π *maintains* goal ψ from S if all of its (complete or not) executions from S do.

We now have all the technical machinery to define the notion of *PLAN*-simulation relations.

Definition 3 (Plan-based Simulation Relation). Let $\mathcal{D}$ be a dynamic domain and $\mathcal{T}$ an agent planning program. A (contingent) *plan-based simulation relation*, or *PLAN-simulation relation*, is a relation $R \subseteq T \times 2^P$ such that $\langle t, S \rangle \in R$ implies that for each transition $t \xrightarrow{\langle \psi, \phi \rangle} t'$ in $\mathcal{T}$, there exists a plan π such that:

1. π *achieves* ϕ and *maintains* ψ from state S; and
2. for all π's possible complete executions from S of the form $\mathcal{S}^0 \xrightarrow{\pi(\tau|_1)} \cdots \xrightarrow{\pi(\tau|_\ell)} S^\ell$, it is the case that $\langle t', S^\ell \rangle \in R$. ■

We say that a plan π *preserves* relation R from $\langle t, S \rangle$ for a given transition $t \xrightarrow{\langle \psi, \phi \rangle} t'$ in $\mathcal{T}$ if requirement 2 above holds.

Observe the strong similarity of the above definition with the formal notion of simulation relation [10]: *PLAN*-simulation relations can be seen as kinds of high-level simulation relations, where transitions are realized by plans, rather than single action executions.

So, a planning program state $t \in T$ is plan-simulated by a $\mathcal{D}$-state $S \in 2^P$, denoted $t \preceq_{PLAN} S$, if there exists a *PLAN*-simulation relation R such that $\langle t, S \rangle \in R$. Clearly, $\preceq_{PLAN}$ is a *PLAN*-simulation relation itself and, in particular, the *largest* one.

Finally, a planning program $\mathcal{T}$ is <u>*realizable*</u> in a dynamic domain $\mathcal{D}$ if $t_0 \preceq_{PLAN} S_0$.

Intuitively, that an agent planning program is realizable means that all potential requests can be fulfilled by a conditional plan. Of course, such requests cannot be arbitrary, they need to respect the planning program's structure. Observe that an *adequate* plan (i.e., one witnessing the existence of a plan-based simulation relation), might in fact not be the shortest one. Indeed, the shortest plan to reach the achievement goal may actually prevent to fulfill future goals in the program. When the agent planning program is indeed realizable in a domain, one can build a function that, if at any point in time the environment reaches state S and the program requests a transition $t \xrightarrow{\langle\psi,\phi\rangle} t'$ in $\mathcal{T}$, outputs a conditional plan π that *(i)* achieves ϕ while maintaining ψ when executed from S; and *(ii)* guarantees that, for all possible states the environment can reach after π's execution, all program transitions outgoing from t' (according to δ) can still be realized by a conditional plan (possibly returned by the function itself). Roughly speaking, the plan returned is a witness of a given agent planning program's realizability or, equivalently, of the fact that $t \preceq_{PLAN} S$. The function in question is referred to as agent planning program *realization* and can be formally defined as follows.

Definition 4 (Agent Planning Program Realization). Let $\mathcal{T}$ be an agent planning program and $\mathcal{D}$ a dynamic domain such that $\mathcal{T}$ is realizable in $\mathcal{D}$. A <u>*realization*</u> of program $\mathcal{T}$ in domain $\mathcal{D}$ is a partial function $\Omega : 2^P \times \delta \mapsto \Pi$ such that, for all pairs $\langle t, S\rangle \in \preceq_{PLAN}$ and all transitions $t \xrightarrow{\langle\psi,\phi\rangle} t'$ in $\mathcal{T}$, the conditional plan $\Omega(S, \langle t, \psi, \phi, t'\rangle)$ achieves goal ϕ and maintains ψ from S, and preserves $\preceq_{PLAN}$ from $\langle t, S\rangle$ for transition $t \xrightarrow{\langle\psi,\phi\rangle} t'$. ■

Informally, for each requested transition of the planning program, its realization outputs a *correct* conditional plan that will not only satisfy the current goals' request, but will also guarantee that all possible requests issued in the future will be fulfilled. The problem we are concerned with is then: *how can such a function be built?*

3. AN EXAMPLE

Consider a researcher's everyday-life domain. The researcher moves among four locations, namely *home*, the department's *parking lot*, the *department* building and the *pub*, by either driving her car, taking a bus, or just walking. Due to highways, traffic restrictions, and distances, not all alternatives are available from every location. For instance, walking from home to the department building or to the department's parking place is not feasible due to (long) distance. Similarly, the researcher may not drive her car directly into department building, as campus circulation is restricted to buses only. In Figure 1(a), all allowed movements are depicted.

Besides the location of the researcher agent, there are other features in the domain (not shown in the figure though). For example, each time the car changes location, it consumes some amount of fuel depending on roads' (unpredictable) traffic conditions. With each trip, the car's tank level (*full*, *low*, or *empty*) may stay the same or go from *full* to *low* and from *low* to *empty*. The car tank can be unconditionally brought to its full level by going to the gas station. For simplicity, though, such activity is not explicitly modeled and we simply assume an action `fill`, executable when the researcher and the car are co-located, that fills the fuel tank instantaneously.

Let us formalize this dynamic domain $\mathcal{D} = \langle P, 2^P, A, S_0, \rho\rangle$. We do so by resorting to PDDL 3.0 enriched with construct `oneof`, so as to capture nondeterministic effects.[1] Figure 2 shows a fragment of the PDDL specification for our domain. The first part, lines 1-11, initializes the domain problem, called `researcherWorld`, by stating types (e.g., `loc`), constants (e.g., `home`), and predicates. In particular, three parametric unary predicates are used, namely, `myLoc` (i.e., the current location of the agent), `carLoc` (i.e., the current location of the car), and `fuel` (i.e., the current car's fuel level). An extra predicate `drove` is used to state that the agent has just drove her car. When all these predicates are fully grounded with constants, the actual propositions in P are obtained (e.g., `myLoc(pub)`).

Next, lines 12-32 define the action `goByCar`, which takes the destination as an input parameter `?d` of type `loc`. Its precondition (lines 14-20) requires *(a)* a non-empty fuel tank; *(b)* a source and destination other than the *department building* (recall it is not allowed to drive on campus); and *(c)* the car and researcher to be at the same place. As for the action's effects (lines 21-31), we have: *(a)* the researcher and the car's locations change to the destination location (line 23-26); *(b)* the researcher has just drove her car (line 23);[2] and *(c)* the fuel level becomes either full or low, if previously full (line 27-28) or either low or empty, if previously low (line 29-30). Observe the use of construct `when` to model conditional effects and the use of construct `oneof` to represent the nondeterministic dynamics of predicate `fuel`—exactly one of the effects listed inside `oneof` must apply. As with predicates, actions are parametric; when fully "grounded," they yield the environment set of actions A. Observe that when an action is grounded, all propositions in its body are grounded as well. The domain transition relation ρ is obtained by executing all possible ground actions in all environment's state. Finally, we consider the initial state S_0 as part of the domain description (line 34): both the researcher and the car are at home, with the car's tank being full.

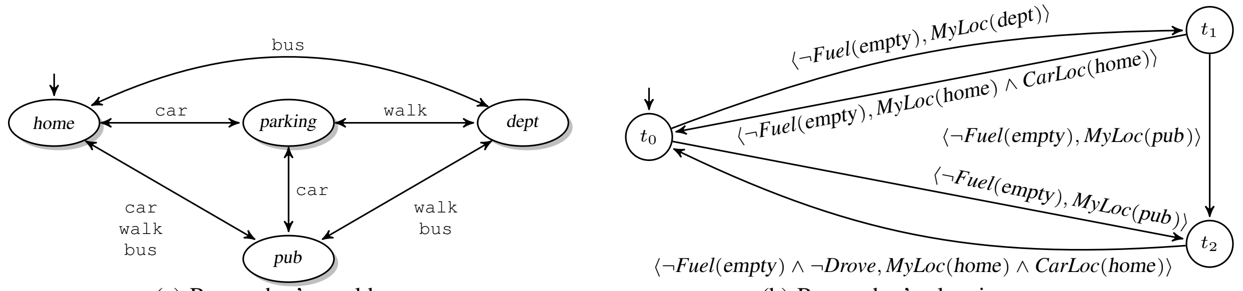
Next, imagine that the researcher agent wants to set up a plan that allows her to go to work and, after work, maybe drop by the pub before heading back to home. Of course, she needs to make sure the car never runs out of fuel and is always at home at the end of the day. Sometimes she may want to go to the pub directly from home (e.g., on weekends). More interesting, for safety reasons, the agent should *not* be driving after being in the pub. Such an agent planning program is depicted in Figure 1(b); each transition is labeled with a tuple $\langle\psi, \phi\rangle$ encoding the maintenance and achievement goals ψ and ϕ, respectively. Observe that, to represent the above constraint of avoiding driving after drinking at the pub, a maintenance goal $\neg$*Drove* is included in the transition from state t_2 to state t_0.

Thus, the question we face is: *can the researcher agent carry out such a program, and if so, how?* Of course, the agent wants the program to work *no matter* how nondeterministic actions turns out to be as the domain evolves. A trivial solution would be to always travel by bus. But what if buses become unexpectedly unavailable, e.g., due to driver strikes?[3] In such a case, the car would be needed in order for the researcher to reach the department. Then, should she wish to go for a beer after work, she could not go there directly by driving her car, as driving back home from the pub is disallowed by the program (see $\neg$*Drove* maintenance goal in the corresponding transition) and she may not leave the car behind at the pub by walking back to home. Consequently, a successful plan would require the agent to pass by home and leave the car there, before moving to the pub by either walking or taking the bus.

[1]Construct `oneof` was proposed for the Fifth International Planning Competition to model actions with nondeterministic effects.

[2]Each time a traveling action other than `goByCar` is performed (e.g., `goByBus`), predicate `Drove` becomes false.

[3]This is currently not modeled in our example, but can easily be accounted for in the domain as non-deterministic events.


bus
home
car
parking
walk
dept
car
walk
bus
car
walk
bus
pub
⟨¬Fuel(empty), MyLoc(dept)⟩
t_1
⟨¬Fuel(empty), MyLoc(home) ∧ CarLoc(home)⟩
t_0
⟨¬Fuel(empty), MyLoc(pub)⟩
⟨¬Fuel(empty), MyLoc(pub)⟩
t_2
⟨¬Fuel(empty) ∧ ¬Drove, MyLoc(home) ∧ CarLoc(home)⟩
(a) Researcher's world map.

(b) Researcher's planning program.

Figure 1: Planning program for a researcher's everyday-life routine.

```
(define (domain researcherWorld)
 (:requirements :typing :equality ... )
 (:types loc fuel_level)

 (:constants
   home parking dept pub - loc
   full low empty - level)

 (:predicates
   (myLoc ?l - loc)
   (carLoc ?l - loc)
   (fuel ?l - level)
   (drove))

 (:action goByCar
   :parameters (?d - loc)
   :precondition
     (and
       (not (fuel empty))
       (not (carLoc dept))
       (not (= ?d dept))
       (exists (?l - loc) (and (myLoc ?l) (carLoc ?l)))
     )
   :effect
      (and
        (and (myLoc ?d) (carLoc ?d) (drove))
        (forall (?l - loc)
          (when (not (= ?l ?d))
                 (and (not (myLoc ?l)) (not (carLoc ?l)))))
        (when (fuel full)
              (oneof (fuel full)(fuel low)))
        (when (fuel low)
              (oneof (fuel low)(fuel empty)))
      )
    )
  ; ... (other actions)
  (:init (myLoc home) (carLoc home) (fuel full))
)
```

Figure 2: Researcher's world domain PDDL specification.

Finally, the agent needs to guarantee that the car never runs out of fuel. To do so, every successful plan is such that, when the car's tank level becomes *low*, the tank is filled out before the car is used again (otherwise the agent might risk violating the program requirement).

4. REACTIVE SYNTHESIS IN LTL

Linear Temporal Logic (LTL) is a well-known logic used to specify dynamic or temporal properties of programs, see e.g., [20]. *Formulas* of LTL are built from a set $\mathcal{P}$ of atomic propositions and are closed under the boolean operators, the unary temporal operators $\bigcirc$ (*next*), $\Diamond$ (*eventually*), and $\Box$ (*always*, from now on), and the binary temporal operator *until* (which in fact can be used to express both $\bigcirc$ and $\Box$, though it will not be used here). LTL formulas are interpreted over infinite sequences σ of propositional interpretations for $\mathcal{P}$, i.e., $\sigma \in (2^{\mathcal{P}})^\omega$. The set of (true) propositions at position i is denoted by $\sigma(i)$, that is, $\sigma = \sigma(0), \sigma(1), \ldots$. If σ is an interpretation, i a natural number, and ϕ is an LTL formula, we denote by $\sigma, i \models \phi$ the fact that ϕ holds in model σ at position i, which is inductively defined as follows (here, $p \in P$ is any proposition and ϕ, ψ any LTL formulas; we omit *until* for brevity):

$$\begin{array}{lll}
\sigma, i \models p & \text{iff} & p \in \sigma(i); \\
\sigma, i \models \phi \vee \psi & \text{iff} & \sigma, i \models \phi;\ \text{or}\ \sigma, i \models \phi \vee \psi; \\
\sigma, i \models \neg\phi & \text{iff} & \sigma, i \not\models \phi; \\
\sigma, i \models \bigcirc\phi & \text{iff} & \sigma, i{+}1 \models \phi; \\
\sigma, i \models \Diamond\phi & \text{iff} & \text{for some } j \geq i, \text{ we have that } \sigma, j \models \phi; \\
\sigma, i \models \Box\phi & \text{iff} & \text{for all } j \geq i, \text{ we have that } \sigma, j \models \phi.
\end{array}$$

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, e.g., a formula ϕ is *satisfiable* if there exists an interpretation that satisfies it. Checking satisfiability or validity for LTL is PSPACE-complete.

Here we are interested in a different kind of logical task, which is called *realizability*, or *Church problem*, or simply *synthesis* [20, 12]. Namely, we partition $\mathcal{P}$ into two disjoint sets $\mathcal{X}$ and $\mathcal{Y}$. We assume to have *no control* on the truth value of the propositions in $\mathcal{X}$, while we can control those in $\mathcal{Y}$. The problem then is: *can we control the values of $\mathcal{Y}$ such that for all possible values of $\mathcal{X}$ a certain LTL formula ϕ remains true?* More precisely, interpretations now assume the form $\sigma = (X_0, Y_0)(X_1, Y_1)(X_2, Y_2)\cdots$, where (X_i, Y_i) is the propositional interpretation at the i-th position in σ, now partitioned in the propositional interpretation X_i for $\mathcal{X}$ and Y_i for $\mathcal{Y}$. Let us denote by $\sigma_{\mathcal{X}}|_i$ the interpretation σ projected only on $\mathcal{X}$ and truncated at the i-th element (included), i.e., $\sigma_{\mathcal{X}}|_i = X_0 X_1 \cdots X_i$. The *realizability problem* checks the existence of a function $f : (2^{\mathcal{X}})^* \rightarrow 2^{\mathcal{Y}}$ such that for all σ with $Y_i = f(\sigma_{\mathcal{X}}|_i)$ we have that σ satisfies the formula ϕ. The *synthesis problem* consists in actually *computing* such a function. Observe that in realizability/synthesis we have no way of constraining the values assumed by the propositions in $\mathcal{X}$: the function we are looking for only acts on propositions in $\mathcal{Y}$. This means that the most interesting formulas for the synthesis have the form $\varphi_a \rightarrow \varphi_r$, where φ_a captures the "relevant" assignments of the propositions in $\mathcal{X}$ (and $\mathcal{Y}$) and φ_r specifies the property we want to assure for such relevant assignments. The realizability (and actual synthesis) are 2EXPTIME-complete for arbitrary LTL formulas [12]. However, recently, several well-behaved patterns of LTL formulas have been identified, for which efficient procedures based on model checking technologies applied to game structures can be devised. Here, we shall focus on one of the most general well-behaved patterns, called "*Generalized Reactivity (1)*" or *GR(1)* [11]. Such formulas have the form $\varphi_a \rightarrow \varphi_r$, with φ_a and ψ_r of the following shape

$$\varphi_a\text{: } \Phi[\mathcal{X},\mathcal{Y}] \wedge \bigwedge_j \Box\Phi_j[\mathcal{X},\mathcal{Y},\bigcirc\Phi[\mathcal{X}]] \wedge \bigwedge_k \Box\Diamond\Phi_k[\mathcal{X},\mathcal{Y}],$$
$$\varphi_r\text{: } \Phi[\mathcal{X},\mathcal{Y}] \wedge \bigwedge_j \Box\Phi_j[\mathcal{X},\mathcal{Y},\bigcirc\Phi[\mathcal{X},\mathcal{Y}]] \wedge \bigwedge_k \Box\Diamond\Phi_k[\mathcal{X},\mathcal{Y}],$$

where $\Phi[\mathcal{Z}]$ (possibly with subscript) stands for any boolean combination of symbols from $\mathcal{Z}$. Notice that: *(i)* with the first conjunct, we can express initial conditions; *(ii)* with the second (big) conjunct, we can express transitions —and we have the further constraint that in doing so within φ_a we cannot talk about the next value of the propositions in $\mathcal{Y}$; and *(iii)* with the third (big) conjunct, we can express *fairness* conditions of the form "it is always true that eventually something holds." For such formulas we have the following result.

Theorem 1 ([11]). *Realizability (and synthesis) of GR(1) LTL formulas $\varphi_a \rightarrow \varphi_r$ can be determined in time $O((p*q*w)^3)$, where p and q are the number of conjuncts of the form $\Box\Diamond\Phi$ in φ_a and φ_r, respectively,*[4] *and w is the number of possible value assignments of $\mathcal{X}$ and $\mathcal{Y}$ under the conditions of φ_a's first two conjuncts.*

5. SOLVING PLANNING PROGRAMS

We now show how to compute a realization of an agent planning program by reducing it to synthesis of a GR(1) LTL formula Υ. The reader should keep in mind that, although the reduction can be informally understood as a set of constraints on the strategy to get the solution, its formal justification is simply Theorem 2, stating its soundness and completeness.

The intuition behind the reduction is as follows. At some point in time, the agent planning program $\mathcal{T}$ and environment $\mathcal{D}$ are in one of their states, say t and S, respectively. $\mathcal{T}$ requests a transition $t \xrightarrow{\psi/\phi} t'$ to be realized. The program realization then builds a plan η executable in S that satisfies two constraints. First, when the plan is executed in S, maintenance goal ψ may not be violated. Second, upon execution completion of plan η, program $\mathcal{T}$ moves to state t' and $\mathcal{D}$ must be in a state S' such that: *(i)* ϕ holds and *(ii)* for all transitions outgoing from t' (i.e., all possible $\mathcal{T}$ next requests), a new plan exists which satisfies the above two constraints.

We start building the GR(1) LTL formula $\Upsilon = \varphi_a \rightarrow \varphi_r$ by specifying the sets of uncontrolled and controlled propositions $\mathcal{X}$ and $\mathcal{Y}$, respectively, and then build assumption formula φ_a and requirement formula φ_r.

Uncontrolled and controlled propositions The set of *uncontrolled* propositions $\mathcal{X}$ is the union of the following sets:

- $\mathcal{X}_P = P$, that is, the propositions p in domain $\mathcal{D}$;
- $\mathcal{X}_T = T$, that is, one proposition for each program state t denoting the current state of $\mathcal{T}$;
- $\mathcal{X}_r = \{req^{\phi}_{\psi} \mid \langle t,\psi,\phi,t'\rangle \in \delta\}$, that is, one proposition for each program transition, where req^{ϕ}_{ψ} states that the agent, according to program $\mathcal{T}$, is (currently) asking for the achievement of goal ϕ while maintaining goal ψ.

The set of *controlled* propositions $\mathcal{Y}$ contains set $\mathcal{Y}_A = A$, that is, one proposition a for each action in the domain $\mathcal{D}$ stating that action a is to be executed next, plus a special proposition $last$ stating that the last action of the current plan is to be executed next.

[4]We assume that both φ_a and φ_r contain at least one conjunct of such a form, if not, we vacuously add the trivial one $\Box\Diamond\top$.

Assumption formula Next, we build a formula of the form $\varphi_a = \varphi^a_{init} \wedge \varphi^a_{trans}$ capturing the *assumptions* on the overall framework the program realization is acting on. For legibility, we define some syntactic shortcuts:

- for each $\mathcal{D}$ state $S \in 2^P$ we define a propositional formula $\gamma_S = \bigwedge_{i=1}^n l_i$, where $l_i = p_i$ if $p_i \in S$; and $l_i = \neg p_i$ otherwise;
- for each program state $t \in T$, we define a propositional formula $req_t = \bigvee_{\langle t,\psi,\phi,t'\rangle\in\delta} req^{\phi}_{\psi}$, representing the fact that the agent is requesting at least one transition available in program state t.

The assumption formula is meant to encode how the overall system is *expected* to behave; technically, it encodes the synchronous execution of dynamic domain $\mathcal{D}$ and program $\mathcal{T}$.

Propositional formula $\varphi^a_{init} = \varphi_{S_0} \wedge t_0$ characterizes the (legal) initial state of the overall system, by requiring $\mathcal{D}$ and $\mathcal{T}$ to start in their respective initial states. Note no constraint on proposition $last$ nor on any proposition in $\mathcal{X}_r$ are imposed here.

LTL formula $\varphi^a_{trans} = \Box trans_{\mathcal{D}} \wedge \Box trans_{\mathcal{T}}$ characterizes the assumptions on the overall system evolution. Specifically, $trans_{\mathcal{D}}$ defines the "rules" for the domain and $trans_{\mathcal{T}}$ defines those for the program. The former is defined as follows:

$$trans_{\mathcal{D}} = \bigwedge_{S\in 2^P, a\in\mathcal{Y}_A} [\gamma_S \wedge a \rightarrow \bigcirc \bigvee_{\{S' \mid \langle S,a,S'\rangle\in\rho\}} \gamma_{S'}].$$

Here, each conjunct states that if the world is in state S and action a is to be executed, then one of the possible states w.r.t. domain transition relation ρ is indeed the *next* state of the world. (We assume that an empty set of disjoins is equal to false.)

Formula $trans_{\mathcal{T}}$, in turn, is built as the conjunction of the following formulae:

- $\bigvee_{t\in\mathcal{X}_T}[t \wedge \bigwedge_{t'\in\mathcal{X}_T\setminus\{t\}} \neg t']$, that is, the program is in exactly one of its states;
- $\bigwedge_{t\in\mathcal{X}_T}[t \rightarrow req_t]$, that is, in each state, the agent executing the program ought to be requesting some of the possible transition available in current state;
- $\bigwedge_{req^{\phi}_{\psi}, req^{\phi'}_{\psi'}\in\mathcal{X}_r, req^{\phi}_{\psi}\neq req^{\phi'}_{\psi'}}[req^{\phi}_{\psi} \rightarrow \neg req^{\phi'}_{\psi'}]$, that is, at most one program transition can be requested at a time;
- $\bigwedge_{\langle t,\psi,\phi,t'\rangle\in\delta}[t \wedge req^{\phi}_{\psi} \wedge last \rightarrow \bigcirc t']$, that is, if transition $t \xrightarrow{\langle\psi,\phi\rangle} t'$ is currently being requested and the last action of current plan is to be executed next, then the program shall move next to its successor state t';
- $\bigwedge_{t\in\mathcal{X}_T}[(t \wedge \neg last) \rightarrow \bigcirc t]$, that is, the program remains still if the current plan is still not completed;
- $\bigwedge_{t\in\mathcal{X}_T,\langle t,\psi,\phi,t\rangle\in\delta}[(t \wedge req^{\phi}_{\psi} \wedge \neg last) \rightarrow \bigcirc req^{\phi}_{\psi}]$, that is, the agent remains requesting the same transition if the current plan is still not completed.

Requirement Formula Let us now build formula $\varphi_r = \varphi^r_{trans} \wedge \varphi^r_{goal}$, which captures the *requirements* for the module to be synthesized, i.e., the program realization: an automaton which, at each step, selects an action for execution.

LTL formula $\varphi^r_{trans} = \Box(\varphi^{act}_{trans} \wedge \varphi^{last}_{trans} \wedge \varphi^{maint}_{trans})$ encodes constraints on action executions and how agent planning programs are "fulfilled." Namely:

- $\varphi^{act}_{trans} = \bigvee_{a \in \mathcal{Y}_A}[a \wedge \bigwedge_{a' \in \mathcal{Y}_A, a' \neq a} \neg a']$, that is, one and only one domain action is expected to be executed at each step;
- $\varphi^{last}_{trans} = \bigwedge_{req^{\phi}_{\psi} \in \mathcal{X}_r}[req^{\phi}_{\psi} \wedge last \rightarrow \bigcirc\phi]$, that is, upon plan completion, achievement goal ϕ, in the requested transition, is indeed achieved;
- $\varphi^{maint}_{trans} = \bigwedge_{req^{\phi}_{\psi} \in \mathcal{X}_r}[req^{\phi}_{\psi} \rightarrow \psi]$, that is, maintenance goal ψ, in the requested transition, is respected along plans' executions.

Finally, by using simply one *fairness* conjunct, we are able to encode the synthesis objective, that is, the realization of the achievement goals and preservation of maintenance ones. Formally, we have:

$$\varphi^{r}_{goal} = \Box \diamond last.$$

That is, we require that each plan is *always eventually completed*. This implies, in turn, that all requested achievement goals are (always eventually) fulfilled.

It is not hard to check that the LTL formula Υ obtained is indeed in GR(1) format. The results from [11] are therefore directly available and we then able to prove our main result:

Theorem 2 (Soundness & Completeness). *There exists a solution to the agent planning program $\mathcal{T}$ in the dynamic domain $\mathcal{D}$ iff the LTL formula Υ, constructed as above, is realizable.*

That is, checking the realizability of Υ is a sound and complete technique for solving the agent planning program in the domain of concern. We stress that by solving realizability with the techniques in [11] we do get an actual solution for the realization of the planning program, not merely verify its existence.

Analyzing the structure of Υ, we get that: *(i)* φ_a contains no subformulas of the form $\Box\Diamond\mu$; *(ii)* φ_r contains just one such subformulas; *(iii)* the number of possible value assignments of $\mathcal{X}$ and $\mathcal{Y}$ under the conditions of $\varphi_a \rightarrow \varphi_r$ is $O(|2^P| * |\delta|)$ (observe that variables that represent the transitions in an agent planning program are pairwise disjoint). Consequently, from Theorem 1, we get that checking the existence of a solution for the agent planning program $\mathcal{T}$ in a dynamic domain $\mathcal{D}$ can be done in $O((|2^P| * |\delta|)^3)$. In fact, such a bound can be refined by replacing 2^P with the number of environment's states that are actually reachable from the initial state. Moreover, since we have no fairness formula in assumption formula φ_a, we just need to check a fairness constraint and not a strong fairness one, hence by inspecting the proof of Theorem 1, we actually get a tighter upperbound, namely $O(|2^P| * |\delta|)$. Now, considering that checking the existence of a conditional plan for an achievement goal in a nondeterministic dynamic domain with full observability is EXPTIME-hard [14], we get a tight computation complexity characterization for solving agent planning problems.

Theorem 3 (Complexity). *Checking the existence of a solution for a agent planning program in a dynamic domain is EXPTIME-complete.*

It is interesting to notice that, in spite of the sophistication of the problem considered, the complexity of solving a agent planning program is essentially the same as that of conditional planning with full observability, a variant of which is solved when realizing each transition of a planning program. In other words, at least from the computational point of view, the additional sophistication introduced by agent planning programs essentially does not require any additional computational effort.

6. BEHAVIOR-BASED PROGRAMMING

So far, we have assumed that, while fulfilling the incoming goal requests from the agent, executable actions (that is, action transitions compatible with the environment) are always available. However, it is usually the case that agents act in the environment only through certain actuators, such as a gripper, a motor, or a web-browser. We will generically model such components as *behaviors* (see below). Besides acting as action executors, thus exposing environment actions to the agent, behaviors may have their own internal logic and *local* actions (e.g., a camera needs to be turned on before a picture can be taken). So, in this section, we consider the problem of realizing an agent planning program in a setting where actions are made available only via a set of available nondeterministic *behaviors*. Nondeterminism captures the fact that these behaviors are *abstractions of actual components* (typically, physical devices or software modules), and hence some of their internal details may not be captured.

In such an extended framework, besides the dynamic domain $\mathcal{D}$ as before, we assume a set of available *behaviors* modeling the components that the agent has at its disposal. Formally, a *behavior* over a dynamic domain $\mathcal{D}$ is a tuple $\mathcal{B} = \langle B, O, b_0, \varrho\rangle$, where:

- B is the finite set of behavior's states;
- O is the finite set of behavior's actions s.t. $O \cap A \neq \emptyset$;
- $b_0 \in B$ is the behavior's initial state;
- $\varrho \subseteq B \times O \times B$ is the behavior's transition relation. We freely interchange notations $\langle b, a, b'\rangle \in \varrho$ and $b \xrightarrow{a} b'$ in $\mathcal{B}$.

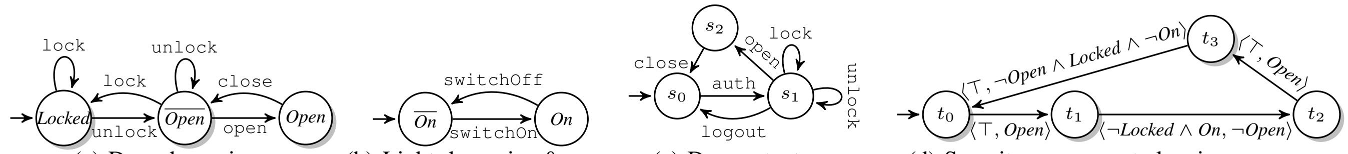
The idea is that, at each state, a behavior offers the agent a set of possible actions from its set O. The agent can interact with the environment $\mathcal{D}$ *only* by means of its available behaviors. When a behavior is instructed to perform an action in the environment, the action is executed and both the behavior in question and the environment evolve, *synchronously* and possibly *nondeterministically*, to their successor states, according to their respective transition relations. So, for an action to be carried out, it needs to be both compatible with the environment and (currently) available in some behavior. As for actions that are *local* to a behavior, that is, actions in $O \setminus A$ (e.g., turning on the camera), they yield no change in the external environment and no executability requirement is enforced on it (though the actions still need to be enabled on the behavior).

The new problem is a straightforward extension of the original one: *can an agent planning program $\mathcal{T}$ be realized in a dynamic domain $\mathcal{D}$ by means of a set of available behaviors $\mathcal{B}_1, \ldots, \mathcal{B}_n$?*

Security Door Example

In a bank agency, customers' security boxes are placed in a security room. The room's door can be *locked/unlocked* and *open/closed.* The door can be opened only if unlocked and locked only if closed. (Un)locking the door when it is (un)locked is allowed, but has no effect. The security room also includes a light, that can be in state *on* or *off.* Our environment $\mathcal{D}$ is indeed the room, i.e., the system composed of the door and the light seen as a whole. From an abstract viewpoint, it consists of the following parts: [5] *(i)* a set of propositions $P = \{\mathit{Locked}, \mathit{Open}, \mathit{On}\}$; *(ii)* a set of actions $A = \{$`lock`, `unlock`, `open`, `close`, `switchOn`, `switchOff`$\}$ with self-explanatory meaning; *(iii)* an initial state $S_0 = \{\mathit{Locked}\}$; and *(iv)* a transition relation ρ describing the joint dynamics of the

[5]Of course, a PDDL specification could be also provided.



(a) Door dynamics. (b) Light dynamics & actuator. (c) Door actuator. (d) Security room agent planning program.

Figure 3: Graphical representation of the security room example.

door and the light, depicted in Figures 3(a) and 3(b), respectively, which evolve independently.

In this scenario, actions are made accessible only through two actuators, model as behaviors, that are able to interact directly with the door and the light. Figure 3(c) shows the door behavior. Note that actions `lock`, `unlock`, `logout`, and `open` become enabled in the behavior, only after a successful authentication in the behavior itself occurs via local action `auth` (only registered customers have access to the security room). To avoid customers from closing the door without logging out, the customer is automatically logged out when the behavior executes a `close` action. Hence, if the customer is inside the room and closes the door, she needs to authenticate again before opening the door and getting out of the room. As for the light switch, for simplicity, we just assume a behavior whose evolution matches the light's dynamics (see Figure 3(b)).

Observe that while an action may be enabled in a behavior, it may not be enabled in the environment, and hence, it cannot be carried out. For instance, if the door behavior is in state s_1 and the environment is in (initial) state $S_0 = \{Locked\}$, action `open` still cannot be performed by the behavior.

Next, as in the previous setting, an agent planning program can be specified to capture the desired ultimate behavior of an autonomous agent. In our case, customer agents are meant to protect their privacy by closing the door when accessing their security boxes. Also, due to bank security policy, they are required *never* to lock the door while inside the room, but to lock it when they are done. So, in order to protect their privacy and respect the bank's policies, customers should *(i)* open the door (and enter the room); *(ii)* switch the light on and close the door without locking it (and access their security boxes); *(iii)* re-open the door (in order to get out); and finally once outside *(iv)* close and lock the door, besides leaving the light off. The planning program corresponding to such a protocol is shown in Figure 3(d).

Compiling behaviors away

The problem now is how to provide the agent system with a (set of) plan(s) to realize a planning program by acting in the domain *only through the available behaviors*.

It turns out that realizing agent planning programs in the extended setting can be easily reduced to the original component-free problem from Section 2. To do so, we *(a)* suitably embed the behavior descriptions into the environment; and then *(b)* generalize the notion of conditional plans, requiring them to return both the action to be executed next and on which behavior.

Let $\mathcal{B}_1, \ldots, \mathcal{B}_n$, with $\mathcal{B}_i = \langle B_i, O_i, b_{i0}, \varrho_i \rangle$, be the set of available behaviors over the dynamic domain $\mathcal{D} = \langle P, 2^P, A, S_0, \rho \rangle$. To encode each $\mathcal{B}_i$ into $\mathcal{D}$, we build a new (extended) dynamic domain $\mathcal{D}' = \langle P', 2^{P'}, A', S'_0, \rho' \rangle$ such that:

1. $P' = P \cup \bigcup_{i=1}^{n} P_i$, where $P_i = \{b \mid b \in B_i\}$ is a set of new propositions representing the different states of available behavior $\mathcal{B}_i$.[6] (wlog we assume sets P and B_i are all disjoint.)
2. $S'_0 = S_0 \cup \{b_{10}, \ldots, b_{n0}\}$, that is, the initial state is extended to include the initial states of all behaviors.
3. $A' = A \cup \bigcup_{i=1}^{n} O_i$, that is, we extend the domain actions to include all behaviors' local actions.
4. $\rho' \subseteq 2^{P'} \times A' \times I \times 2^{P'}$, where $I = \{1, \ldots, n\}$, such that for each $S, S' \in 2^{P'}$, $\langle S, a, i, S' \rangle \in \rho'$ iff:
   - for all $i \in I$, both $S \cap P_i$ and $S' \cap P_i$ are singletons, that is, S and S' represent the fact that each behavior is in exactly one of its states;
   - $\langle S \cap P, a, S' \cap P \rangle \in \rho$, that is, $\mathcal{D}$ state $S \cap P$—the projection of S on P—enables a's execution with possible successor state $S' \cap P$;
   - $\langle b, a, b' \rangle \in \varrho_i$, for $S \cap P_i = \{b\}$ and $S' \cap P_i = \{b'\}$, that is, $\mathcal{B}_i$ enables a's execution in state b, with b' as a possible successor state.

   That is, ρ' essentially represents the synchronous product of $\mathcal{D}$ with the asynchronous product of all behaviors $\mathcal{B}_i$'s.

Observe that extending the transition relation as above has an impact on the definition of domain's histories. Indeed, state transitions are no longer of the form $S^j \xrightarrow{a} S^{j+1}$, but rather of the form $S^j \xrightarrow{a,i} S^{j+1}$. However, generalizing such notions to the extended domain is straightforward and we skip that here.

Next, the notion of plans also needs to be generalized, as these are not only required to output the next action to be executed, but *how* it is going to be executed, that is, the specific behavior that will actually carry it out in the environment. By referring to the set of all $\mathcal{D}'$'s (generalized) histories as $\mathcal{H}'$, we define a *behavior-based conditional plan* as a function $\pi : \mathcal{H}' \mapsto A' \times I$. Again, the notions of *plan execution*, *finite plan*, and *complete execution* extend naturally to this behavior-based setting.

At this point, it can be easily seen that, by introducing a minor modification to the LTL encoding so as to integrate index i, the selected behavior to carry on the next chosen action, one can essentially adopt the same resolution strategy used in the basic scenario to realize agent planning programs in the behavior-based setting. Note that the original framework with no actuators is trivially captured by the extended one, by assuming a single stateless behavior that always enables all domain actions.

Finally, as for complexity, observe that, when n behaviors are present, assuming a logarithmic encoding of the states of the behaviors, we get that $|P'| = O(|P| + n * \log(\max_{i=1}^{n} |B_i|))$. Therefore, recalling that the complexity of realizing an agent planning program is exponential in the number of domain propositions (see Section 4 and 5), we get that also this variant of the problem can be solved in EXPTIME.

[6]This can be done compactly via a logarithmic encoding of behaviors' states. For legibility, though, we use the naïve encoding.

7. CONCLUSION

This work combines automated planning and high-level agent-oriented programming into the novel problem of synthesizing what we call an *agent planning program* for execution on a planning domain. We provided a solution by resorting to *synthesis* for a well-behaved class of LTL formulas for which synthesis can be reduced to model-checking of game structures. This allows us to leverage on existing results and tools (including TLV[7], Anzu[8], and Ratsy[9]).

Interestingly, agent planning programs are particularly suited to represent "routines" in contexts, such as smart-homes,[10] where one wants to continuously support a predefined set of sequences of activities that cyclically repeat over and over.

More generally, our work shares the very same motivations as that of HTN planning [3, 4] and other approaches to mix planning with (agent) programming (e.g., [17, 5]), and in fact, we took such works as an inspiration for our research. Roughly speaking, in such works, an agent is "programmed" in a high-level manner and a final working course of action is synthesized by performing lookahead. Our approach is different both in the kind of problem being solved as well as in the technique used to actually solve it. Rather than taking a strong procedural "goal-to-do" view of agent's goals, we have taken instead a *purely declarative view on goals*. We described agents by means of both achievement and maintainance "goal-to-be," rather than by tasks or processes the agent ought to carry on. Moreover, we relied on a powerful synthesis technique that leaves us room for interesting extensions, such as dealing with nondeterministic environments under explicit fairness assumptions. Indeed, it is worth remarking that we did not use the full power of GR(1) specifications: the assumption formula of our formalization contains no subformulae of the form $\Box\Diamond\phi$. We could consider such kind of formulas to specify, e.g., unbounded but eventualy terminating cyclic sub-behaviors in the dynamic domain. Also, one might consider solutions that work for all possible outcomes of nondeterministic actions, provided plan executions are *fair*, in the sense that they do not loop indefinitely over a same state sequence [1].

Finally, we note that in this work, we have assumed *full observability* on both the underlying domain and behavior states. When the observability assumption is no longer valid, i.e., when planning under *partial observability* or for *conformant plans*, one could still reduce the problem to LTL synthesis [12, 7]. However, there is no guarantee that a GR(1) formulas would be obtained, and hence one would lose the effectiveness of the techniques for synthesis via model checking. Such effectiveness can possibly be recovered by some ad-hoc construction, such as an adaptation of the classical *belief construction* used for planning with partial observability [14, 2], see however [16] for limitations on this approach. Looking into this issue is an interesting direction for future research.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. Fabio Patrizi is partially funded by the IST Programme of the EU Commission – Project SM4All (FP7-224332). Sebastian Sardina acknowledges the support of Agent Oriented Software and the Australian Research Council (under grant LP0882234).

8. REFERENCES

[1] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

[2] G. De Giacomo, R. De Masellis, and F. Patrizi. Composition of partially observable services exporting their behaviour. In *Proc. of ICAPS'09*, 2009.

[3] K. Erol, J. A. Hendler, and D. S. Nau. Complexity Results for HTN Planning. *Annals of Mathematics and Artificial Intelligence*, 18(1):69–93, 1996.

[4] M. Ghallab, D. S. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[5] K. V. Hindriks and T. Roberti. GOAL as a planning formalism. In *Proc. of MATES*, volume 5774 of *LNCS*, pages 29–40. Springer, 2009.

[6] S. Kerjean, F. Kabanza, R. St.-Denis, and S. Thiébaux. Analyzing LTL model checking techniques for plan synthesis and controller synthesis (work in progress). *Electronic Notes Theoretical Comput. Science*, 149(2):91–104, 2006.

[7] O. Kupferman and M. Y. Vardi. Synthesis with incomplete information. In D. G. Howard Barringer, Michael Fisher and G. Gough, editors, *Advances in Temporal Logic*, pages 109–127. Kluwer Academic Publishers, Jan. 2000.

[8] Y. Lespérance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. Foundations of a Logical Approach to Agent Programming. In *Proc. of Int. Workshop ATAL*. 1995.

[9] H. J. Levesque and R. Reiter. High-level Robotic Control: Beyond Planning. A Position Paper. In *AIII 1998 Spring Symposium: Integrating Robotics Research: Taking the Next Big Leap*, Mar. 1998.

[10] R. Milner. An algebraic definition of simulation between programs. In *Proc. of IJCAI*, pages 481–489, 1971.

[11] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) Designs. In *VMCAI*, pages 364–380, 2006.

[12] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Proc. of POPL*, pages 179–190, 1989.

[13] A. S. Rao. Agentspeak(L): BDI agents speak out in a logical computable language. In *Proc. of MAAMAW*, volume 1038 of *LNCS*, pages 42–55. Springer, 1996.

[14] J. Rintanen. Complexity of planning with partial observability. In *Proc. of ICAPS*, pages 345–354, 2004.

[15] S. Sardina and G. De Giacomo. Realizing multiple autonomous agents through scheduling of shared devices. In *Proc. of ICAPS*, pages 304–312, 2008.

[16] S. Sardina, G. De Giacomo, Y. Lespérance, and H. J. Levesque. On the Limits of Planning over Belief States. In *Proc. of KR*, pages 463–471, 2006.

[17] S. Sardina, L. P. de Silva, and L. Padgham. Hierarchical planning in BDI agent programming languages: A formal approach. In *Proc. of AAMAS*, pages 1001–1008, 2006.

[18] Y. Shoham. Agent-oriented programming. *Artificial Intelligence Journal*, 60:51–92, 1993.

[19] B. van Riemsdijk, M. Dastani, and J.-J. Meyer. Semantics of declarative goals in agent programming. In *Proc. of AAMAS*, pages 133–140, 2005.

[20] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, volume 1043 of *LNCS*, pages 238–266, 1996.

[7] `www.cs.nyu.edu/acsys/tlv/`

[8] `www.ist.tugraz.at/staff/jobstmann/anzu/`

[9] `rat.fbk.eu/ratsy/`

[10] See, e.g., `www.sm4all-project.eu`