High-Level Programming via Generalized Planning and LTL Synthesis

Blai Bonet[1]*, **Giuseppe De Giacomo**[2], **Hector Geffner**[3], **Fabio Patrizi**[2], **Sasha Rubin**[4]
[1]Universidad Simón Bolívar, Venezuela
[2]Sapienza University of Rome, Italy
[3]ICREA & Universitat Pompeu Fabra, Spain
[4]The University of Sydney, Australia
bonet@usb.ve, {degiacomo, patrizi}@dis.uniroma1.it, hector.geffner@upf.edu,
sasha.rubin@sydney.edu.au

Abstract

We look at program synthesis where the aim is to automatically synthesize a controller that operates on data structures and from which a concrete program can be easily derived. We do not aim at a fully-automatic process or tool that produces a program meeting a given specification of the program's behaviour. Rather, we aim at the design of a clear and well-founded approach for supporting programmers at the design and implementation phases.

Concretely, we first show that a program synthesis task can be modeled as a generalized planning problem. This is done at an abstraction level where the involved data structures are seen as black-boxes that can be interfaced with actions and observations, the first corresponding to the operations and the second to the queries provided by the data structure. The abstraction level is high enough to capture intuitive and common assumptions as well as general and simple strategies used by programmers, and yet it contains sufficient structure to support the automated generation of concrete solutions (in the form of controllers). From such controllers and the use of standard data structures, an actual program in a general language like C++ or Python can be easily obtained.

Then, we discuss how the resulting generalized planning problem can be reduced to an LTL synthesis problem, thus making available any LTL synthesis engine for obtaining the controllers. We illustrate the effectiveness of the approach on a series of examples.

1 Introduction

Program synthesis is the problem of turning a specification into an executable. Broadly speaking, there are two types of programs: transformational (that query or transform finite but unbounded data) and reactive (that control and react to an infinite stream of data, e.g., controllers or protocols). Synthesis for transformational programs was pioneered by (Green 1969; Waldinger and Lee 1969), while for reactive programs in (Church 1963; Abadi, Lamport, and Wolper 1989; Pnueli and Rosner 1989). The latter has an elegant and comprehensive theory (Finkbeiner 2016; Ehlers et al. 2017; Gerstacker, Klein, and Finkbeiner 2018), and is deeply related to nondeterministic planning (De Giacomo and Vardi 2015; De Giacomo and Vardi 2016; De Giacomo and Rubin 2018; Camacho et al. 2018; Camacho, Bienvenu, and McIlraith 2019; Aminof et al. 2019).

In this paper we consider the synthesis of *transformational* programs that involve manipulation of data structures, like lists, trees, and graphs. Typical examples of desired programs are finding the minimum in a list, searching for an element in a tree, and traversing a graph. We synthesize transformational programs by devising requirements on their executions, expressing such requirements as an LTL reactive synthesis task, and finally solving the resulting task. We do not propose a fully automated approach where a program is produced that satisfies a given declarative specification of the input/output requirement. Rather, we devise a well-founded approach aimed at supporting program design and implementation. We consider a setting where the designer provides a *declarative pseudocode specification* in terms of *i) execution assumptions*, that formalize abstract knowledge on how the data structures evolve during execution, and *ii)* the *goal*, that formalizes the desired executions of the sought program.

Execution assumptions can be *transitional* or *temporal*. *Transitional assumptions* capture the changes in program state as operations and queries over data structures are executed. These changes, however, are expressed using LTL over a language of action and observable symbols that capture the behaviour of the data structures in terms of high-level features. For example, a list is characterized by an observable that tells whether the cell beneath the cursor has a successor cell, and by the operation that moves the cursor to the next cell and that is executable only when such next cell exists. Since the features do not capture the state of the data structures (e.g., length of the list or cursor's position within the list), the transitional assumptions involve nondeterminism (e.g., when the cursor moves, the next cell may or may not have a next cell).

Temporal assumptions express properties that are achieved after several steps along an execution, such as *moving repeatedly to the next item in a finite list eventually leads to the last item in the list*. These assumptions, that are not expressible with nondeterminism alone, are essential for the correct modelling of executions over concrete data structures. They are typically used implicitly by the programmer without being formalized. Understanding the implicit temporal assumptions constitute a real effort that the program-

*Work performed during sabbatical leave at Universidad Carlos III de Madrid, Spain.

mer confronts when devising an algorithm. In comparison, the transitional assumptions directly follow from the data structures' specification.

Finally, the goal description specifies the set of successful executions of the program sought in terms of LTL formulas over the action and observable symbols. Such a description fulfills the role of a formal input/output description of the programming synthesis task, and is a clear benefit of the proposed approach to synthesis since input/output descriptions are difficult to produce and often involve the use of higher-order logics or operational models. In the case of moving through a list, for example, a successful execution reaches a cell that has no successors, while in the case of tree traversal, a successful execution reaches a configuration with an empty set of unexpanded nodes (provided that generated nodes are automatically added to such a set).

When all the elements are expressed in the right format, the programming task becomes the task of obtaining a controller that generates executions that conform with the goal, yet only the executions that conform with the transitional and temporal assumptions are the ones required to conform with the goal. In other words, since during execution of the program on concrete data structures the induced trajectories adhere to the (transitional and temporal) assumptions, the controller must ensure that the goal is fulfilled on those executions; on the other hand, executions that do not satisfy the assumptions do not need to be considered by the controller as they are only possible in the abstraction but not in the environment where the program executes (since the data structures behave as expected).

We observe that a program can be represented as a finite-state controller with observations (queries or tests) and actions (operations), and that automatically building it requires a form of synthesis or planning in an *infinite-state* setting, as the data present in the structures is unbounded, in general. Effective techniques for program synthesis and planning mainly focus on *finite-state* systems, while results on synthesis for infinite-state systems are scarce (Levesque 1996; Levesque 2005; Lin 2016; Lin 2018; Calvanese et al. 2018). Observe also that execution assumptions and goals are logical requirements on the *trajectories* that the controller considers/produces. The "procedural" flavor of such logical specifications correlates to some proposals in knowledge representation and reasoning (Levesque et al. 1997; De Giacomo, Lespérance, and Levesque 2000; Bacchus and Kabanza 2000; Baier et al. 2008).

In order to reduce the synthesis task in the infinite-state setting to a synthesis task in a finite-state setting, we rely on *generalized planning*, i.e., planning for solving multiple instances at once (Srivastava, Immerman, and Zilberstein 2008; Bonet, Palacios, and Geffner 2009; Hu and De Giacomo 2011; Belle and Levesque 2016; De Giacomo et al. 2016; Bonet et al. 2017). We assume predefined data structures with parameterless operations and tests, but allow for (finitely many) auxiliary registers to implicitly store parameters and results. We then view programming for a given task as a generalized planning problem $\mathcal{Q}$ that consists of an infinite collection of planning problems P that share a common set of applicable actions and a common set of observables; for example, the task of traversing a list corresponds to a problem $\mathcal{Q}$ whose instances P correspond to each possible finite list, and all problems in P share the action to move to the next element, or stay put, and share the observations of whether there is a next element. On the other hand, differently from generalized planning, we do not require that the goal of the programming task is expressible in terms of observables. This is because in our case the goal depends on the actual state of the data structures and cannot always be captured by the observables.

To obtain the finite-state controller that solves the generalized planning problem $\mathcal{Q}$ (and thus the programming task), the infinite set $\mathcal{Q}$ of classical planning problems is *abstracted* into a *single nondeterministic* problem $\mathcal{Q}^A$ which is defined over the common sets of actions and observations, and that captures the transitional assumptions. This abstraction however is often too coarse and without solution because, due to the non-determinism, it admits *spurious* executions that do not correspond to any execution on a concrete problem P in $\mathcal{Q}$. Temporal assumptions are then added, expressed as linear-time temporal (LTL) formulas (Pnueli 1977), in order to prune the spurious executions and render the abstraction solvable. Typical examples of such assumptions are forms of fairness for action effects (Bonet et al. 2017). In this work, we lift such "fairness" assumptions to arbitrary temporal restrictions on the environment (Aminof et al. 2019).

Concerning controller synthesis, we observe that while FOND planners (Srivastava et al. 2011; Bonet and Geffner 2015) would be readily available to solve the abstract problem $\mathcal{Q}^A$ in absence of temporal assumptions, when these are present, such tools cannot be exploited anymore. For this reason, we require the designer to express the generalized planning problem $\mathcal{Q}^A$ and the temporal assumptions in compact form as LTL formulas in order to enable the use of LTL synthesis tools. The choice of the tool is only a technical detail: the problem we are solving is generalized planning in presence of temporal assumptions.

We demonstrate the effectiveness of our approach on some examples, which include (singly and doubly-linked) list, tree, and graph traversal programs, as well as programs for testing membership and finding the minimum in a list. In our examples, we use the state-of-the-art native LTL synthesis engine Strix (Meyer, Sickert, and Luttenberger 2018).

2 Generalized Planning

The framework adopted in this paper extends the framework of Bonet et al. (2017).[1] A *planning instance* is a deterministic classical planning problem extended with an observation function. Namely, each instance P defines a state model $\langle S, s_0, T, Act, A, f, obs, \Omega \rangle$ in compact form, where S is a finite set of *states*, $s_0 \in S$ is the *initial state*, $T \subseteq S$ is the set of *goal states*, Act is the finite set of *actions*, $A : S \rightarrow 2^{Act}$ is the *available-actions* function, $f : Act \times S \rightarrow S$ is the

[1] The crucial difference with Bonet et al. (2017) is that goals in the single instances are different and not expressible directly through the observables, i.e., we *do not assume* that the goals is a Boolean combination of observables.

deterministic *state transition function*, $obs : S \rightarrow \Omega$ is the *observation function*, and Ω is the finite set of *observations*.

A solution for an instance P is an action sequence $a_0, \ldots, a_{n-1}$ that generates a goal-reaching state sequence $s_0, \ldots, s_n$; namely, each action a_i in the sequence must be applicable, i.e. $a_i \in A(s_i)$, the state s_{i+1} must be the state that follows action a_i in state s_i, i.e. $s_{i+1} = f(a_i, s_i)$, and s_n must be a goal state, i.e. $s_n \in T$.

A *generalized problem* $\mathcal{Q}$ is a set of instances P with the *same actions* Act, the *same observations* Ω, and the *same observable action preconditions*.[2] Having the same observable action preconditions means that for every observation $\omega \in \Omega$, there is a set of actions $A_\omega \subseteq Act$ such that in every instance P of $\mathcal{Q}$, $A(s) = A_\omega$ if $obs(s) = \omega$.

A *policy* μ for a generalized problem $\mathcal{Q}$ is a partial function that maps interleaved sequences $\omega_0, a_0, \omega_1, a_1, \ldots, a_{i-1}, \omega_i$ of observations and actions, ending in observations, into actions. A policy μ is said to be memoryless if it only depends on the last observation of the input sequence; i.e., $\mu(\tau) = \mu(\tau')$ if τ and τ' end in the same observation, for every interleaved sequences τ and τ'. A policy μ *induces* a unique sequence of states and actions $s_0, a_0, \ldots, s_n$ in each instance P of $\mathcal{Q}$ where $s_{i+1} = f(a_i, s_i)$, $a_i = \mu(\langle \omega_0, a_0, \ldots, \omega_i \rangle)$ and $\omega_i = obs(s_i)$, $i \geq 0$. For convenience, we make all these sequences of infinite length, assuming that all instances in $\mathcal{Q}$ contain a no-op action *stop* with no preconditions or effects which is applied forever when the execution induced by μ reaches an observation-action prefix where μ is undefined or returns a non-applicable action.

The policy μ *solves an instance* P if it induces a sequence $s_0, a_0, s_1, a_1, \ldots$ over P that is goal-reaching; i.e. $s_n \in T$ for some n. The policy μ *solves the generalized problem* $\mathcal{Q}$ if it solves each problem P in $\mathcal{Q}$.

Example 1. The following generalized planning problem $\mathcal{Q}$ captures the task of traversing a singly-linked list from the beginning, and then stops (forever). Let $\mathcal{Q}$ consist of problems P_ℓ parameterized by the length ℓ of a list. Each problem P_ℓ has action n(ext) that moves the cursor to the next position (if present), and action s(top) that disables action n. The states encode the current position pos of the cursor, and the initial state has $pos = 1$. The goal is the state in which $pos = \ell$. There are two observations, $hasNext$ that holds on states that have $pos < \ell$, and its complement $\neg hasNext$. The (memoryless) policy μ that only looks at the last observation and is defined as $\mu(hasNext) = n$ and $\mu(\neg hasNext) = s$ solves $\mathcal{Q}$. □

Later, we will see an example of traversing a doubly-linked list.

3 Observable Abstractions

For solving generalized planning problems $\mathcal{Q}$ where goals are not observable, we consider an *observation abstraction* $\mathcal{Q}^A$ of $\mathcal{Q}$ that is defined top-down as follows:

Definition 1. *An* observation abstraction $\mathcal{Q}^A$ *of a generalized problem* $\mathcal{Q}$ *is a triplet* $\mathcal{Q}^A = \langle Q^o, \Gamma_F, \Gamma_G \rangle$*, where* Q^o *is the observation projection of* $\mathcal{Q}$*, and* Γ_F *and* Γ_G *are respectively (execution) temporal assumptions and goal constraints on trajectories over* Q^o *that are sound for* $\mathcal{Q}$*.*

We now define the three components in $\mathcal{Q}^A$, namely the observation projection Q^o of $\mathcal{Q}$, the temporal assumptions Γ_F sound for $\mathcal{Q}$ and the goal constraints Γ_G sound for $\mathcal{Q}$.

The *observation projection* captures the possible transitions among observations over the instances P in $\mathcal{Q}$ is defined as:

Definition 2. *For a generalized problem* $\mathcal{Q}$*, the* observation projection $Q^o = \langle S^o, I^o, Act^o, A^o, F^o \rangle$ *is the nondeterministic state model defined by:*

- $S^o = \Omega$,
- $I^o = \{\omega \mid \textit{there is } P \textit{ in } \mathcal{Q} \textit{ such that } obs(s_0) = \omega\}$,
- $Act^o = Act$,
- $A^o(\omega) = A_\omega$ *for every* $\omega \in \Omega$,
- $F^o(a, \omega) = \{\omega' \mid \textit{there are } P \in \mathcal{Q}, a \in A^o(\omega) \textit{ and } s \in S_P$ *such that* $obs(s) = \omega$ *and* $obs(f(a, s)) = \omega'\}$

where Ω*,* Act *and* A_ω *refer to elements of* $\mathcal{Q}$*, and* s_0 *and* S_P *are the initial state and state space of* P *respectively.*

The observation projection is a (fully observable) nondeterministic domain over the actions and observations that are common to the instances in $\mathcal{Q}$. It is similar to the observation projection of Bonet et al. (2017) but with *no information about the goal* because it is no longer assumed to be observable.

The *temporal assumption* Γ_F encodes information in the instances P (typically about fairness of effects) that is lost in the projection Q^o because it is not about individual observation *transitions*, but about entire observation *trajectories* (Bonet et al. 2017). Formally:

Definition 3. *A* sound temporal assumption Γ_F *for* $\mathcal{Q}$ *is a set of observation-action sequences* $\omega_0, a_0, \omega_1, a_1, \ldots$ *in the projection* Q^o *that includes all the* infinite trajectories *that arise in instances* P *in* $\mathcal{Q}$*.*

The observation projection Q^o is indeed nondeterministic, but this non-determinism is a device of the abstraction, and it is neither fair, as assumed in strong cyclic solutions, nor adversarial, as assumed in strong solutions (Cimatti et al. 2003). The temporal assumptions encode constraints on the effects of non-deterministic actions when these actions are applied infinitely often.

Finally, goals are expressed in the abstraction through constraints Γ_G on observation-action trajectories that ensure that the state-action trajectories in the instances are goal reaching:

Definition 4. *A* sound goal constraint Γ_G *for* $\mathcal{Q}$ *is a set of observation-action trajectories* $\tau : \omega_0, a_0, \omega_1, a_1, \ldots$ *of the projection* Q^o *such that every state-action trajectory* $s_0, a_0, s_1, a_1, \ldots$ *of a problem* P *in* $\mathcal{Q}$ *that gives rise to* τ*, i.e. for which* $\omega_i = obs(s_i)$*, reaches the goal in* P*.*

Example 1 (cont). For the generalized problem $\mathcal{Q}$ above, the states in the observation projection Q^o are the common

[2] In fact, observability of action preconditions it is not strictly necessary, e.g., it is not assumed in (Hu and De Giacomo 2011), but it simplifies our treatment.

observations, i.e., $hasNext$ and $\neg hasNext$. The effect of the action s(top) in the abstraction is deterministic, i.e. it does not change the observation, but the effect of n(ext) is *non-deterministic* as it can result in $hasNext$ or $\neg hasNext$. The sound goal-constraint Γ_G consists of all trajectories of Q^o in which $\neg hasNext$ is observed. Finally, the sound temporal assumption Γ_F consists of all trajectories of Q^o except those where n is performed infinitely often and $\neg hasNext$ is never observed. □

We now define the concept of solutions for the abstraction $\mathcal{Q}^A$. Let Γ be an assumption on observation-action trajectories, e.g., an execution assumption or a goal constraint. An observation-action trajectory in Γ is said to *satisfy* Γ.

Definition 5. *A policy μ* solves *the abstraction $\mathcal{Q}^A = \langle Q^o, \Gamma_F, \Gamma_G\rangle$ of $\mathcal{Q}$ if the trajectories induced by μ on the observation projection Q^o that satisfy Γ_F also satisfy Γ_G.*

The key property of a sound observation abstraction is that:

Theorem 1. *If a policy μ solves a sound observation abstraction $\mathcal{Q}^A$ of $\mathcal{Q}$, then the policy μ solves $\mathcal{Q}$.*

Proof. Let τ be an infinite state-action trajectory in an instance P of $\mathcal{Q}$ induced by μ, and let $obs(\tau)$ be the infinite observation-action trajectory associated with τ. By definition of Q^o and Γ_F, $obs(\tau)$ is a trajectory in $\mathcal{Q}^o$ induced by μ that also satisfies Γ_F. Since μ solves $\mathcal{Q}^A$, $obs(\tau)$ satisfies Γ_G. Then, τ is goal-reaching in P since Γ_G is sound for $\mathcal{Q}$. Therefore, since τ is arbitrary, μ solves $\mathcal{Q}$. □

Notice that Theorem 1 is a soundness result, which is useful only when the abstraction $\mathcal{Q}^A$ is solvable.

Example 1 (cont). In the example above, observe that the trajectories in Q^o induced by the policy μ that satisfy the temporal assumption Γ_F also satisfy the goal constraint Γ_G. On the other hand, Γ_F is important since there are μ-trajectories that neither satisfy Γ_F nor Γ_G. Hence, μ would not be a solution if all μ-trajectories were considered. □

4 LTL Observation Abstractions

The observation abstraction $\mathcal{Q}^A$ can often be expressed in compact form, in particular as Linear-time Temporal Logic (LTL) formulas (Pnueli 1977). This is our focus. The syntax of LTL formulas over variables from the set V is

$$\varphi ::= p \mid \varphi \vee \varphi \mid \neg\varphi \mid \circ\varphi \mid \varphi \,\mathsf{U}\, \varphi,$$

where $p \in V$. Intuitively, p means that p is true in the current time, $\circ\,\varphi$ that φ is true in the next time step, and $\varphi \,\mathsf{U}\, \varphi'$ that i) eventually φ' is true, and ii) at every step up until (but not necessarily including) that time, φ is true. We use the following usual abbreviations: $\Diamond\,\varphi$ (read "eventually") for $\top \,\mathsf{U}\, \varphi$, and $\Box\,\varphi$ (read "always") for $\neg(\Diamond\,\neg\varphi)$.

LTL formulas are interpreted over infinite trajectories (or traces), i.e., infinite sequences, $\alpha = \alpha_0\alpha_1 \ldots \in (2^V)^\infty$ of propositional valuations. Given α and an LTL formula φ, we inductively define when α *satisfies φ at step i*, written $\alpha, i \models \varphi$, as follows:

- $\alpha, i \models p$ iff $p \in \alpha_i$ (for $p \in V$);
- $\alpha, i \models \neg\varphi$ iff $\alpha, i \not\models \varphi$;
- $\alpha, i \models \varphi_1 \wedge \varphi_2$ iff $\alpha, i \models \varphi_1$ and $\alpha, i \models \varphi_2$;
- $\alpha, i \models \circ\,\varphi$ iff $\alpha, i+1 \models \varphi$;
- $\alpha, i \models \varphi_1 \,\mathsf{U}\, \varphi_2$ iff there exists $j \geq i$ s.t. $\alpha, j \models \varphi_2$, and for all $k, i \leq k < j$ we have that $\alpha, k \models \varphi_1$.

We say that α *satisfies* φ, written $\alpha \models \varphi$ if $\alpha, 0 \models \varphi$.

LTL synthesis (Pnueli and Rosner 1989) is the problem of producing a controller that achieves a given property no matter how the environment behaves. The idea is that the environment and controller interact in discrete stages. At stage i the environment sets the variables in some fixed set X and the controller then responds by setting the variables in some fixed (disjoint) set Y, producing a valuation $X_i \cup Y_i$ over the variables $X \cup Y$. This interaction repeats producing an infinite sequence $(X_0 \cup Y_0)(X_1 \cup Y_1)\cdots$ of valuations. In planning terminology, X can be viewed as a representation of observations (i.e. $\Omega = 2^X$), and Y as a representation of actions (i.e. $Act = 2^Y$). A sequence $(X_0 \cup Y_0)(X_1 \cup Y_1)\cdots$ is *induced* by the policy μ if $\mu(\langle X_0, Y_0, X_1, Y_1, \cdots, X_i\rangle) = Y_i$ for every i. For an LTL formula φ over the variables $X \cup Y$, a controller policy μ *solves the LTL synthesis problem for the formula φ* if every sequence induced by μ satisfies φ.[3] When such a μ exists we say that φ is *realizable* by the controller. Dually, we say that φ is *realizable by the environment* if there is an environment policy, i.e., a function that maps every finite trajectory (ending in an action) to a state, that induces trajectories that satisfy φ.

LTL observation abstractions $\mathcal{Q}^A = \langle Q^o, \Gamma_F, \Gamma_G\rangle$ are observation abstractions represented in compact form by LTL formulas (D, E, F, G) that are defined over the symbols a and o encoding the actions and observations in $\mathcal{Q}^A$ (i.e., over the variables in X and Y). The formulas are:

- D for capturing the action preconditions of Q^o, the requirement of doing exactly one action at a time, and the technical fact that once the action $stop$ is done it is the only applicable action;
- E for capturing the initial conditions, action effects, and the frame axioms (persistence) of Q^o (notice that both D and E can be automatically computed from the observation projection Q^o of Q, see e.g., (Aminof et al. 2019; Camacho, Bienvenu, and McIlraith 2019));
- F for capturing Γ_F, the execution temporal constraints on the environment over Q^o;
- G for capturing Γ_G, the goal over Q^o.

Intuitively, satisfying D and G is the responsibility of the controller, while satisfying E and F is the responsibility of the environment. In the presence of assumptions about the environment behavior, such as our E and F, the synthesis specification is taken to be an implication:

$$Env \supset Cntrl$$

where Env are the assumptions on the environment and $Cntrl$ is the specification of controller's desired behavior

[3] In these formulas, there is no syntactic distinction between observations and actions, they are all propositional variables.

(under the environment assumptions Env). Not every LTL formula can be used for Env: it is required that the environment must have a strategy to enforce Env in spite of what the controller does. Formally, Env must be realizable by the environment, i.e., there must be a strategy for the environment that solves Env. We refer to (Aminof et al. 2019) for a thorough discussion.

In our case $Env \supset Cntrl$ is detailed as follows:

- Env is $D \supset (E \wedge F)$, i.e., the environment guarantees that when the preconditions are satisfied, the effects and temporal assumptions hold.
- $Cntrl$ is $D \wedge G$, i.e., (when Env is guaranteed) the controller will guarantee that both the preconditions D and the goal G hold.

Importantly, given what D, E and F represent, the requirement that Env be realizable by the environment is always satisfied, as the following theorem shows.

Theorem 2. *Let D, E, F, and G be LTL formulas representing the LTL observation abstraction $\mathcal{Q}^A = \langle Q^o, \Gamma_F, \Gamma_G \rangle$ as above. Then $D \supset (E \wedge F)$ is realizable by the environment.*

Proof sketch. We need to find a strategy for the environment such that every induced sequence that satisfies D also satisfies $E \wedge F$. Intuitively, the environment can behave as follows. It simulates the execution of some fixed concrete instance P (with transition function f and initial state s_0) from $\mathcal{Q}$, and on its turn simply outputs the observation of the current state of P. That is, on the first turn, the environment outputs $obs(s_0)$. We assume the agent responds with a single applicable action $a_0 \in A(s_0)$ (otherwise we are done), and then on its second turn, the environment outputs $obs(f(s_0, a_0))$, and this process repeats. The resulting sequence satisfies D (this is because we are in the case that the agent always plays single applicable actions), and thus, by construction of the environment strategy, the sequence also satisfies E, i.e., it is a trajectory of P. By soundness of Γ_F, the trajectory satisfies Γ_F, and thus, by definition of F, the trajectory also satisfies F. □

The formula $(D \supset (E \wedge F)) \supset (D \wedge G)$ is logically equivalent to the formula $D \wedge ((E \wedge F) \supset G)$ which is the one used below. The latter formula requires the policy μ to prescribe actions that satisfy their preconditions, and in all induced trajectories where E and F are true (i.e., the initial conditions, dynamics, and temporal assumptions), the goal G is true as well. Our task will be the synthesis of a policy for the formula $D \wedge ((E \wedge F) \supset G)$. The correctness of the approach is guaranteed by the following results.

Lemma 6. *Let D, E, F, and G be LTL formulas representing the LTL observation abstraction $\mathcal{Q}^A = \langle Q^o, \Gamma_F, \Gamma_G \rangle$ as above. Then, if the policy μ solves the LTL synthesis problem for the formula $D \wedge ((E \wedge F) \supset G)$, then μ solves $\mathcal{Q}^A$.*

Theorem 3. *Let $\mathcal{Q}$ be a generalized planning problem, and let $\mathcal{Q}^A$ be a sound observation abstraction of $\mathcal{Q}$ with corresponding LTL formulas D, E, F, and G. Then every policy μ that solves the LTL synthesis problem for the formula $D \wedge ((E \wedge F) \supset G)$ solves $\mathcal{Q}$.*

5 Our Framework at Work

We now show how to use the above framework for obtaining controllers for program synthesis tasks. In each case, given a description T (even in natural language) of a programming task, take the following steps:

1. Think of T as a generalized planning problem $\mathcal{Q}_T$, and single out a common set of actions and observations;
2. Provide *sound* temporal assumptions and goal constraints for the observation abstraction $\mathcal{Q}_T^A$ of $\mathcal{Q}_T$;
3. Write LTL formulas D, E, F, G characterizing the observation abstraction $\mathcal{Q}_T^A$;
4. Automatically solve the LTL synthesis problem for the formula $D \wedge ((E \wedge F) \supset G)$, and output a solution/controller μ for Q_T^A if there is one. If there is no solution, revise the formulas Γ_F and Γ_G, trying to make the former stronger to prune further spurious executions and the latter weaker to allow more goal-reaching executions, and go back to step 3.

Remark 1. *Proving that this approach is correct requires formalizing T, e.g., in mathematics, in logic, or in a formal specification language such as Z (Spivey 1989). Such a formalization would* induce *a generalized planning problem $\mathcal{Q}_T$, e.g., by considering all the possible instantiations of the data structures in the formalization. One could then* prove *that the temporal assumption and goal constraint are sound for $\mathcal{Q}_T$, and so deduce, by Theorem 3, that a synthesized controller μ solves $\mathcal{Q}_T$, and hence that μ solves the programming task T.*

To illustrate our framework at work, we will work with singly-linked lists, doubly-linked lists, trees, and graphs, along with their corresponding operations, which operate as expected (we omit details for brevity). We will make use of the following: (finitely many) cursors that point to cells in the data structures (like Java iterators) and (finitely many) registers, if needed; sensors that compare values in the cells pointed at by the cursors and in the registers, resulting in a fixed set of booleans/observations; operations that move the pointers in the structures and allow us to copy the content of one pointed cell into another, resulting in a fixed set of actions.

A number of elements are common among the various models below. We use action $s(top)$ as a final no-op indicating termination (once done, it repeats forever), and restrict to executing exactly one action per step, expressed using the XOR operator "$\oplus$". We also use the abbreviations: PERSISTS(p) to express persistence of p, i.e. $(p \leftrightarrow \circ p)$, and PERSISTS$(p_1, \ldots, p_n)$ to express persistence of $p_1, p_2, \ldots, p_n$.

All controllers are obtained with the *online demo* version of Strix (https://strix.model.in.tum.de/try) and computed in less than 10 seconds.

Traversing a Singly-Linked List. The first task T is list traversal, which involves visiting (with a cursor) each cell of a list exactly once, starting from the first (leftmost) cell. Actions and observations in the generalized problem $\mathcal{Q}_T$ as well as the LTL formulas for the abstraction are as follows:

- Observations: $hasNext$ (cell pointed at by cursor has a successor (next) cell)
- Actions: n(ext), move cursor to next cell, s(top)
- Formula D (preconditions, non-concurrent actions, and stop action):
  - $\Box(n \rightarrow hasNext)$
  - $\Box(n \oplus s)$
  - $\Box(s \rightarrow \circ s)$
- Formula E (initial conditions, effects, and frame):
  - $\Box(s \rightarrow \textsc{persists}(hasNext))$
- Formula F (temporal assumptions):
  - $\Box\Diamond n \rightarrow \Diamond\neg hasNext$
- Formula G (goal): visit all cells once, i.e., reach the end and stay there (we could also use $\Diamond\neg hasNext$ as the cursor can never go back; this will change in the following examples)
  - $\Diamond\Box\neg hasNext$

The obtained controller is shown in Figure 1. The controller is initially in q_0, where it remains as long as $hasNext$ is observed, and moves to q_1 (where it remains forever) as soon as $\neg hasNext$ is observed. At state q_0, if not at the end of list, the controller prescribes n(ext), and s(top) otherwise; at state q_1, s(top) is unconditionally prescribed. It is easy to see that all cells have been visited iff the controller is at q_1. Notice that although this particular controller uses memory (it has two states), this is not required: it could use just one state and prescribe n when $hasNext$ and s otherwise.

The formulas define a sound observation abstraction, and hence every controller solving the LTL synthesis problem for the formula $D \wedge ((E \wedge F) \supset G)$ also solves $\mathcal{Q}_T$.

Let us give an idea of how one can check for soundness (see Remark 1). Let us suppose that we formalize a list as the usual abstract data type[4] that has a single variable pos varying over the positions of the list, and operations n(ext) and s(top). This formalization induces a generalized planning problem $\mathcal{Q}_T$ obtained by instantiating the data type in all possible ways. In particular, the i-th planning problem P_i in $\mathcal{Q}_T$ has observations $\neg hasNext$, actions n and s, states in $\{1, 2, \ldots, i\} \times \{done, \neg done\}$, and the initial state is $(1, \neg done)$; in addition, if we refer to the first component of the state as its *position*, the goal states have position i, the observation function outputs $\neg hasNext$ iff the position is i, action s is always available, action n is available in all states of the form $(j, \neg done)$ where $j < i$, and the transitions are $n(j, \neg done) = (j + 1, \neg done)$, and $s(j, L) = (j, done)$ for $L \in \{done, \neg done\}$.

Now, one can formally show that the execution assumptions and goal formulas given above are sound for $\mathcal{Q}_T$. In this case, it is easy to see: the execution assumption is true in every trajectory of P_i; and every trajectory in P_i that satisfies the goal constraint reaches the goal state.

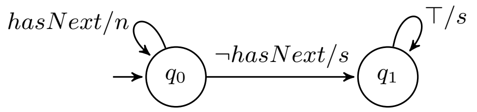

Figure 1: Controller for traversing a singly-linked list.

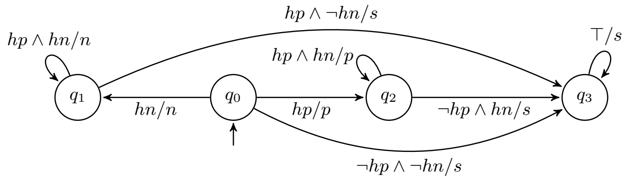

Figure 2: Controller for traversing a doubly-linked list. The controller produced by Strix has non-realizable edges; e.g., there is an edge from the initial state (q_0) on observation $hp \wedge hn$. Likewise, the controller may have states that are only reachable through non-realizable edges. For clarity, such edges and states are not shown in this figure nor the following figures.

Traversing a Doubly-Linked List. This is analogous to list traversal except that the list is doubly-linked, i.e., the cursor can move to the n(ext) or p(revious) cell (if present). The observations hn (has next) and hp (has previous) model the fact that a cell has a successor/previous cell. Actions and observations in the generalized problem $\mathcal{Q}_T$ and the LTL formulas modeling the abstraction are as follows:

- Observations: hn, hp (cell pointed at by cursor has previous or successor)
- Actions: n(ext), p(revious), s(top)
- Formula D (preconditions, non-concurrent actions, and stop action):
  - $\Box(n \rightarrow hn) \wedge \Box(p \rightarrow hp)$
  - $\Box(n \oplus p \oplus s)$
  - $\Box(s \rightarrow \circ s)$
- Formula E (initial conditions, effects, and frame):
  - $\neg hp \vee \neg hn$
  - $\Box(n \rightarrow \circ hp) \wedge \Box(p \rightarrow \circ hn)$
  - $\Box(s \rightarrow \textsc{persists}(hp, hn))$
- Formula F (temporal assumptions):
  - $(\Box\Diamond n \wedge \Diamond\Box\neg p) \rightarrow \Diamond\neg hn$
  - $(\Box\Diamond p \wedge \Diamond\Box\neg n) \rightarrow \Diamond\neg hp$
- Formula G (goals): visit all cells once, i.e., reach other end and never invert direction
  - $(\neg hp \rightarrow \Diamond\Box\neg hn) \wedge (\neg hn \rightarrow \Diamond\Box\neg hp)$
  - $\Box((p \rightarrow \circ\, \neg n) \wedge (n \rightarrow \circ\, \neg p))$

The controller is shown in Figure 2. At state q_0 no previous observations have been collected nor actions have been performed; at q_1 (resp. q_2) the cursor has moved from its initial position and has all its previous (resp. successor) cells visited; q_3 stands for cursor at an extreme position with all cells visited. The prescribed actions are as follow: if the cursor is on the first cell, move to next cell until the end of the list (path $q_0 \rightarrow q_1 \cdots q_1 \rightarrow q_3$); if the cursor starts on the last cell and not on the first, then move to previous cell until

[4] For this example, we can ignore the values in the cells of the list. In later examples, we use such values.

the beginning of the list (path $q_0 \to q_2 \cdots q_2 \to q_3$); if the list has only one cell then stop (path $q_0 \to q_3$); once at q_3, stop forever.

Here, contrarily to the singly-linked case, memory is required. When the cursor is not at either extreme, whether all previous or successor cells have been visited depends on "how" the current position was reached. If it was done starting from the end (resp. beginning) and performing only p(revious) (resp. n(ext)) then all previous (resp. successor) cells have already been visited, thus the cursor must move to previous (resp. next) cell. To distinguish between these two situations, the controller needs states q_1 and q_2, as the sole information "in the middle of list" does not suffice to make the correct decision.

Formulas can be shown to be sound in essentially the same way as for the singly-linked list.

Traversing a Tree. Tree traversal requires visiting all nodes in a finite tree exactly once, starting from the root. We assume a memory where nodes to be visited can be stored and retrieved, initially only containing the root node.

- Observations: em(pty memory), ha(s_children) (current node's children have not been put in memory)
- Actions: e(xtract a node from memory, for visiting), p(ut all children of current node into memory), s(top)
- Formula D (preconditions, non-concurrent actions, and stop action):
  - $\Box(e \to \neg em) \wedge \Box(p \to ha)$
  - $\Box(e \oplus p \oplus s)$
  - $\Box(s \to \circ s)$
- Formula E (initial conditions, effects, and frame):
  - $\neg em$
  - $\Box(p \to \circ(\neg em \wedge \neg ha))$
  - $\Box(s \to \textsc{persists}(em, ha))$
- Formula F (temporal assumption):
  - $\Box\Diamond e \to \Diamond(em \wedge \neg ha)$
- Formula G (goals): visit all nodes once, i.e., end up with empty memory and no children to visit and never extract until all children of current node are put in memory
  - $\Diamond\Box(em \wedge \neg ha)$
  - $\Box(ha \to (\neg e \,\mathsf{U}\, p))$

We discuss formulas F and G. Formula F says that if one keeps extracting nodes from memory, a point will be eventually reached where the current extracted node has no children and the memory is empty. F is sound because action p, which has precondition ha and puts all the current node's children in memory, cannot be executed forever in a finite tree. The last formula in G says that children nodes must always be visited, i.e., that the children of the current node must be put in memory before other nodes are extracted.

The controller is shown in Figure 3. State q_0 models absence of previous observations and actions; in q_1 a node has just been extracted and its children are not in memory; in q_2 all the current node's children are in memory, and memory is nonempty; in q_3 the memory is empty and all the current node's children are in memory, i.e., the node is leaf. The controller prescribes: to extract a node whenever memory is nonempty and all the current node's children are in memory (paths $q_0 \to q_1$, $q_1 \to q_1$, and $q_2 \to q_1$); put all the current node's children in memory whenever they are not ($q_0 \to q_2$, $q_1 \to q_2$); stop on empty memory and current node with no children ($q_1 \to q_3$).

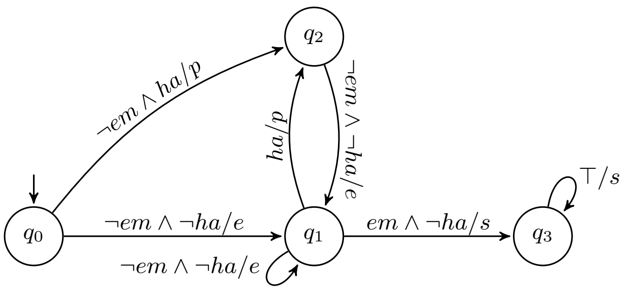
q_2
$\neg em \wedge ha/p$
ha/p
$\neg em \wedge \neg ha/e$
$\top/s$
q_0
$\neg em \wedge \neg ha/e$
q_1
$em \wedge \neg ha/s$
q_3
$\neg em \wedge \neg ha/e$

Figure 3: Controller for tree traversal.

Traversing a Graph. Graph traversal requires all the nodes of a finite graph to be visited exactly once. We assume that the graph is connected. Also in this case, a memory is available to store nodes to visit. We assume that when nodes are put in memory, they are simultaneously marked, which is the only way to mark nodes. The abstraction for graph traversal is essentially the same as for trees, except:

- Observation and action ha(s_children) and p(ut_children) are replaced by ha(s_unmarked) and p(ut_unmarked) respectively. The latter puts the unmarked neighbors in memory while marking them.
- The temporal assumption formula F is changed to:
  - $(\Box\Diamond e \wedge \Diamond\Box\neg p) \to \Diamond\Box\, em$
  - $\Box\Diamond p \to \Diamond\Box\neg ha$
- Formulas D, E, and G are as for tree traversal, except for renaming of ha(s_children) to ha(s_unmarked), and p(ut_children) to p(ut_unmarked).

The first temporal assumption formula says that if one keeps marking (unmarked) nodes, at some point there will be no unmarked nodes left, as the graph is finite. The second temporal assumption formula says instead that the memory will eventually become empty if one keeps extracting nodes from memory and at some point stops putting nodes there.

The obtained controller is essentially identical to the controller for tree traversal depicted in Fig. 3, except that the symbols ha(s_children) and p(ut_children) are replaced by ha(s_unmarked) (neighbors) and p(ut_unmarked) (neighbors) respectively. Soundness can be easily checked.

Minimum of a List. The task is to return in a register the minimum element of a finite, singly-linked list. We assume the cursor starts on the first cell. A register can be updated with the value stored in the current cell, and one can check whether the value in the current cell is less than the value in the register. The abstraction is a slight change from the one used for singly-linked list traversal:

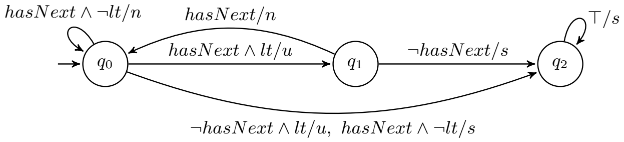
$hasNext \wedge \neg lt/n$
$hasNext/n$
$\top/s$
q_0
$hasNext \wedge lt/u$
q_1
$\neg hasNext/s$
q_2
$\neg hasNext \wedge lt/u,\ hasNext \wedge \neg lt/s$

Figure 4: Controller for finding minimum in a list.

- Observations: $hasNext$ (as in list), lt (value stored in pointed cell is less than value in register)
- Actions: n(ext) and s(top) as in list, plus u(pdate_register with value in pointed cell)
- Formula D (preconditions, non-concurrent actions, and stop action): as in list traversal, but with $\Box(n \oplus s)$ replaced by $\Box(n \oplus u \oplus s)$
- Formula E (initial conditions, effects, and frame):
  - $\Box(u \rightarrow (\circ\neg lt \wedge \text{PERSISTS}(hasNext)))$
  - $\Box(s \rightarrow \text{PERSISTS}(hasNext, lt))$
- Formula F (temporal assumptions): as in singly-linked list
- Formula G (goals): as singly-linked list traversal, plus update register iff current value is smaller than register
  - $\Box(u \leftrightarrow lt)$

The key formula is G: it requires the value in the register to be updated iff it is greater than the value in the current cell. The controller is shown in Figure 4. It prescribes moving to next cell until a cell with a value less than the value in the register is pointed at (paths $q_0 \rightarrow q_0$, $q_1 \rightarrow q_0$) or the end of the list is reached. In the former case, even if the end is reached, an update action is prescribed (paths $q_0 \rightarrow q_1$, $q_0 \rightarrow q_2$), in the latter case, a stop action is performed forever ($q_0 \rightarrow q_2$, $q_2 \rightarrow q_2$).

Membership in a Tree. This task requires checking whether a finite tree contains some node with a value equal to that stored in a register. We assume the same data structure as for tree traversal, extended with a register for storing the value to be found. The goal is to stop the search when the label for the current node matches the value in the register or, if there is no such node, to carry out a full traversal. The abstraction is obtained from small changes in the abstraction for tree traversal:

- Observations: em(pty) and ha(s_children) as in tree, eq(ual) (pointed node has label matching searched value)
- Actions: e(xtract), p(ut_children) and s(top) as in tree traversal
- Formula D (preconditions, non-concurrent actions, and stop action): as in tree traversal
- Formula E (initial conditions, effects, and frame): as in tree traversal except for s
  - $\Box(s \rightarrow \text{PERSISTS}(em, ha, eq))$
- Formula F (temporal assumptions): as in tree traversal
- Formula G (goals): search value, i.e., same as tree traversal, but stop when the value is found
  - $\Diamond\Box eq \vee \Diamond\Box(em \wedge \neg ha)$
  - $\Box((ha \wedge \neg eq) \rightarrow (\neg e \,\mathsf{U}\, p))$
  - $\Box(eq \rightarrow s)$

The first goal formula states that the traversal should proceed until the element sought is found or there are no more nodes. The second says that until the sought element is found, no children should be left unexplored. The third says that the traversal stops when the element is found. The obtained controller is shown in Fig. 5.

Swamp Crossing. This is a classical programming exercise about recursion. A swamp is represented as a matrix containing either *water* or *land* in its cells. The task is to find a path of (horizontally or right-diagonally) adjacent land cells from a leftmost (land) cell to a rightmost cell. In our specification, only cells with *land* are explicitly labelled, and *water* is assumed in all other cells. In this variant, we focus only on checking whether the rightmost column is reachable, without actually constructing (nor returning) the path found. The abstraction we propose is conceptually similar to that used for tree membership, once matrix cells are interpreted as nodes and the neighbor relationship holds (asymmetrically) between a cell and those right-adjacent to it. We also model the presence of water or land in each cell and assume that the memory initially contains all cells (nodes) of the leftmost column:

- Observations: em(pty memory), ha(s_neighbors) (current node has some neighbor not put in memory), r(ightmost) (current node belongs to rightmost column), l(and) (current node is land);
- Actions: e(xtract a node from memory, for visiting), p(ut all neighbors of current node into memory, if not already there), s(top)
- Formula D (preconditions, non-concurrent actions, and stop action):
  - $\Box(e \rightarrow \neg em) \wedge \Box(p \rightarrow ha)$
  - $\Box(e \oplus p \oplus s)$
  - $\Box(s \rightarrow \circ s)$
- Formula E (initial conditions, effects, and frame):
  - $\neg em$
  - $\Box(p \rightarrow \circ(\neg em \wedge \neg ha))$
  - $\Box(s \rightarrow \text{PERSISTS}(em, ha, r, l))$
- Formula F (temporal assumptions): by continuously extracting nodes from memory, it eventually becomes empty
  - $\Box\Diamond e \rightarrow \Diamond(em \wedge \neg ha)$
- Formula G (goals): visit all reachable land nodes once, until a rightmost land node (if any) is achieved:
  - $\Diamond\Box(r \wedge l) \vee \Diamond\Box(em \wedge \neg ha)$
  - $\Box((ha \wedge l \wedge \neg r) \rightarrow (\neg e \,\mathsf{U}\, p))$
  - $\Box((r \wedge l) \rightarrow s)$

Similarly to membership in a tree, the visit stops when a rightmost land cell is achieved (last formula). However, only neighbors of land nodes are put in memory for future visiting (second formula) to guarantee that the cells "touched" when moving are all made of land. Notice also that we do not explicitly prevent visiting cells more than once since the

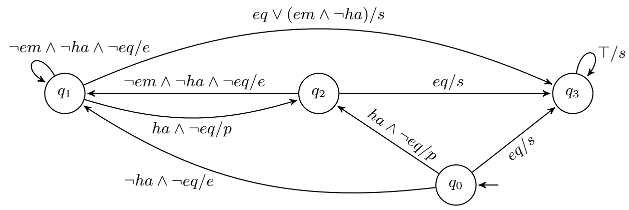
$eq \vee (em \wedge \neg ha)/s$
$\neg em \wedge \neg ha \wedge \neg eq/e$
$\top/s$
q_1
$\neg em \wedge \neg ha \wedge \neg eq/e$
q_2
eq/s
q_3
$ha \wedge \neg eq/p$
$ha \wedge \neg eq/p$
eq/s
$\neg ha \wedge \neg eq/e$
q_0

Figure 5: Controller for finding an element in a tree.

graph induced by the matrix is a acyclic. This however could have been easily prevented by enforcing a marking mechanism similar to the one used for graph traversal which only requires a different concrete instantiation of the actions *extract* and *put*. The obtained controller has a structure similar to the controller for finding an element in a tree, cf. Fig. 5.

6 Conclusions

We addressed the problem of automatically constructing a concrete program in the form of a finite-state controller from a high-level LTL specification that describes the programming environment and the requirements on the controller. The former describes the transitional and temporal assumptions of the data structures, initial conditions, and non-concurrency of operations, while the requirements on the controller identify successful executions, i.e., executions that solve the programming task. Our methods solve two difficult obstacles associated with programming synthesis tasks. First, by considering the observation projection of the generalized planning problem $\mathcal{Q}$ paired with the transitional and temporal assumptions, the synthesis task is formulated over a finite state space rather than over an infinite state space. Second, by using goal formulas, we circumvent the need to formalize the programming task in terms of mappings from inputs to outputs, a formalization that is typically complex and uses higher-order logics or operational models.

The whole approach has been implemented and tested by students as part of the graduate course "Elective in AI: Reasoning Agents" at University of Rome "La Sapienza". The implementation takes the input specification, runs the LTL solver and translates the output controller into a Python program. The implementation has been used successfully to solve several programming problems, including: find min/max of a list, copy even numbers into another list, sum the positive numbers of a list, bubble sort, find an element in a tree, and a robot navigation task inside a matrix.

Other approaches oriented towards automated programming have been proposed previously, where planners have been used to derive general programs and controllers, but from examples (Bonet, Palacios, and Geffner 2009; Aguas, Celorrio, and Jonsson 2019) and thus offer no guarantees on the generality and correctness of the resulting programs. On the other hand, our approach is deductive and thus guarantees that any solution for the observable abstraction $\mathcal{Q}^A$ is indeed a solution for the programming task.

While a number of FOND planners exist that deal with different LTL fragments (Patrizi, Lipovetzky, and Geffner 2013; Camacho et al. 2017; Camacho et al. 2018), none seem to correctly handle the fragment needed here. We have used general LTL tools to cope with the unrestricted temporal assumptions. From the planning perspective, the challenge is to replace the LTL tools, which will not scale in the presence of many observations, by scalable FOND planners.

From a general perspective it is interesting to investigate how far the approach can be pushed. E.g., one could study how different specifications can be combined in a modular way, or whether (some) forms of recursive procedures can be handled.

Acknowledgments

This work was partly supported by the European Research Council under the EU Horizon 2020 Programme through the ERC Advanced Grant WhiteMech (No. 834228) and by the Sapienza Project DRAPE: Data-awaRe Automatic Process Execution. B. Bonet was funded by a Banco Santander – UC3M Chair of Excellence Award. The work of Hector Geffner is partially supported by grant TIN-2015-67959-P from MINECO, Spain. The authors thank Edoardo Cicero, Daniele Giunta, and Luca Pierdicca for implementing and testing the approach in an actual Python-based setting.

References

Abadi, M.; Lamport, L.; and Wolper, P. 1989. Realizable and unrealizable specifications of reactive systems. In *ICALP*, volume 372 of *Lecture Notes in Computer Science*, 1–17. Springer.

Aguas, J. S.; Celorrio, S. J.; and Jonsson, A. 2019. Computing programs for generalized planning using a classical planner. *Artif. Intell.* 272:52–85.

Aminof, B.; De Giacomo, G.; Murano, A.; and Rubin, S. 2019. Planning under LTL environment specifications. In *ICAPS*, 31–39.

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artif. Intell.* 116(1-2):123–191.

Baier, J. A.; Fritz, C.; Bienvenu, M.; and McIlraith, S. A. 2008. Beyond classical planning: Procedural control knowledge and preferences in state-of-the-art planners. In *AAAI*, 1509–1512.

Belle, V., and Levesque, H. J. 2016. Foundations for generalized planning in unbounded stochastic domains. In *KR*, 380–389.

Bonet, B., and Geffner, H. 2015. Policies that generalize: Solving many planning problems with the same policy. In *IJCAI*, 2798–2804.

Bonet, B.; De Giacomo, G.; Geffner, H.; and Rubin, S. 2017. Generalized planning: non-deterministic abstractions and trajectory constraints. In *IJCAI*, 873–879.

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *ICAPS*, 34–41.

Calvanese, D.; De Giacomo, G.; Montali, M.; and Patrizi, F. 2018. First-order μ-calculus over generic transition systems and applications to the situation calculus. *Inf. Comput.* 259(3):328–347.

Camacho, A.; Triantafillou, E.; Muise, C. J.; Baier, J. A.; and McIlraith, S. A. 2017. Non-deterministic planning with temporally extended goals: LTL over finite and infinite traces. In *AAAI*, 3716–3724.

Camacho, A.; Muise, C. J.; Baier, J. A.; and McIlraith, S. A. 2018. LTL realizability via safety and reachability games. In *IJCAI*, 4683–4691.

Camacho, A.; Bienvenu, M.; and McIlraith, S. A. 2019. Towards a unified view of AI planning and reactive synthesis. In *ICAPS*, 58–67.

Church, A. 1963. Logic, arithmetics, and automata. In *Proc. Int. Congress of Mathematicians, 1962*.

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence* 147(1-2):35–84.

De Giacomo, G., and Rubin, S. 2018. Automata-theoretic foundations of FOND planning for LTLf and LDLf goals. In *IJCAI*, 4729–4735.

De Giacomo, G., and Vardi, M. Y. 2015. Synthesis for LTL and LDL on finite traces. In *IJCAI*, 1558–1564.

De Giacomo, G., and Vardi, M. Y. 2016. Ltl_f and ldl_f synthesis under partial observability. In *IJCAI*, 1044–1050.

De Giacomo, G.; Di Stasio, A.; Murano, A.; and Rubin, S. 2016. Imperfect-information games and generalized planning. In *IJCAI*, 1037–1043.

De Giacomo, G.; Lespérance, Y.; and Levesque, H. J. 2000. Congolog, a concurrent programming language based on the situation calculus. *Artif. Intell.* 121(1-2):109–169.

Ehlers, R.; Lafortune, S.; Tripakis, S.; and Vardi, M. Y. 2017. Supervisory control and reactive synthesis: a comparative introduction. *Discrete Event Dynamic Systems* 27(2):209–260.

Finkbeiner, B. 2016. Synthesis of reactive systems. In *Dependable Software Systems Engineering*. IOS Press. 72–98.

Gerstacker, C.; Klein, F.; and Finkbeiner, B. 2018. Bounded synthesis of reactive programs. In *ATVA*.

Green, C. C. 1969. Application of theorem proving to problem solving. In *IJCAI*, 219–240.

Hu, Y., and De Giacomo, G. 2011. Generalized planning: Synthesizing plans that work for multiple environments. In *IJCAI*, 918–923.

Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *J. Log. Program.* 31(1-3):59–83.

Levesque, H. J. 1996. What is planning in the presence of sensing? In *Proc. AAAI*, 1139–1146.

Levesque, H. J. 2005. Planning with loops. In *IJCAI*, 509–515. Professional Book Center.

Lin, F. 2016. A formalization of programs in first-order logic with a discrete linear order. *Artif. Intell.* 235:1–25.

Lin, F. 2018. Machine theorem discovery. *AI Magazine* 39(2):53–59.

Meyer, P. J.; Sickert, S.; and Luttenberger, M. 2018. Strix: Explicit reactive synthesis strikes back! In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, 578–586. Springer.

Patrizi, F.; Lipovetzky, N.; and Geffner, H. 2013. Fair LTL synthesis for non-deterministic systems using strong cyclic planners. In *IJCAI*, 2343–2349. IJCAI/AAAI.

Pnueli, A., and Rosner, R. 1989. On the synthesis of a reactive module. In *Proc. of POPL 1989*, 179–190.

Pnueli, A. 1977. The temporal logic of programs. In *FOCS*, 46–57. IEEE Computer Society.

Spivey, J. M. 1989. An introduction to Z and formal specifications. *Software Engineering Journal* 4(1):40–50.

Srivastava, S.; Zilberstein, S.; Immerman, N.; and Geffner, H. 2011. Qualitative numeric planning. In *AAAI*.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning generalized plans using abstract counting. In *AAAI*, 991–997.

Waldinger, R. J., and Lee, R. C. T. 1969. PROW: A step toward automatic program writing. In *IJCAI*, 241–252.