High-level Programming via Generalized Planning and LTL Synthesis

Blai Bonet\textsuperscript{1}, Giuseppe De Giacomo\textsuperscript{2}, Hector Geffner\textsuperscript{3}, Fabio Patrizi\textsuperscript{2}

Sasha Rubin\textsuperscript{4}
\textsuperscript{1}Universidad Simon Bolivar, Venezuela
\textsuperscript{2}Sapienza University of Rome, Italy
\textsuperscript{3}ICREA & Universitat Pompeu Fabra, Spain
\textsuperscript{4}The University of Sydney, Australia
bonetblai@gmail.com, \{degiacomo, patrizi\}@dis.uniroma1.it, hector.geffner@upf.edu, sasha.rubin@sydney.edu.au

Abstract
We look at a form of program synthesis where the aim is to automatically synthesize a controller that operates on data structures, from which a concrete program can easily be derived. We do not aim at fully automatizing the process, i.e., we do not propose an approach that starts from a declarative specification of the input/output requirements and produces an executable program. Rather, we aim at: 1. devising a well-founded approach to support a programmer at design time; and 2. automatizing the implementation phase.

To this end, 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 offering actions and observations i.e., operations and tests provided by the data structure. The abstraction level is high enough to capture the assumptions and the goals a programmer has typically in mind when producing a program, but concrete enough to support automated generation of a concrete solution (in the form of a controller).

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

1 Introduction
Program synthesis consists in 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 planning in nondeterministic domains (De Giacomo and Vardi 2015; De Giacomo and Vardi 2016; De Giacomo and Rubin 2018; Camacho et al. 2018; Camacho, Bienvenu, and McIraith 2019; Aminof et al. 2019).

In this paper, we consider synthesizing transformational programs that involve manipulating data structures, such as lists, trees, and graphs, so as to automatize, to a certain extent, programming tasks such as finding the minimum in a list, searching an element in a tree, graph traversal, etc., but we synthesize such transformational programs by devising requirements on executions and solving reactive synthesis. We do not propose a fully automated approach where a declarative specification of the input/output requirements is taken as input and the program satisfying it is produced. Rather, we devise a well-founded approach aimed at supporting program design and at automatizing the implementation phase. We consider a setting where the designer formalizes the execution assumptions and the goal based on which he would write the program, as in a sort of declarative pseudocode specification.

Execution assumptions can be transitional or temporal. Transitional assumptions model the (high-level) program’s successor state relation that stems from the observation of the data structures against characteristic properties (e.g., emptiness), as well as from effects of the operations provided by the data structures (e.g., inserting an item in a list makes the list non-empty); these are provided at an abstraction level higher than the concrete one, involving nondeterminism in both actions and observations, to capture ignorance about the actual contents of the data structures.

Temporal assumptions express (multiple-step) properties of executions, such as moving repeatedly to the next item in a finite list will eventually lead to the last item. Such assumptions, not expressible with nondeterminism alone, are essential to correctly model executions on concrete data structures. Typically, they are used at programming time in an implicit way (in the designer mind) without being formalized. Temporal assumptions constitute the real effort a designer affords when devising an algorithm, in that transitional assumptions can easily be derived from standard data structures’ documentation (as long as unambiguous).

By exploiting the assumptions, we want to fulfill the goal. Since the specification does not express the input/output relation but the program execution from a high-level perspective, goals capture program states where the programming task is considered completed. For instance, in a tree traversal, one stops when the set of unexpanded nodes is empty: this is the program’s state which guarantees, under the as-
We tackle synthesis in the infinite-state setting by relying on generalized planning, i.e., planning for solving multiple instances (corresponding to the instantiated data structures) at once (Srivastava, Immerman, and Zilberstein 2008; Bonet, Palacios, and Geffner 2009; Hu and De Giacomo 2011; Belle and Levesque 2016; Bonet et al. 2017). To this end, we assume predefined data structures with parameterless operations and tests, but we allow for (finitely many) auxiliary registers to implicitly store parameters, as well as partial and final results, etc. We then view programming for a given task (e.g., finding the minimum in a list) as a generalized planning problem $Q$, consisting of a collection of planning problems $P$ (e.g., one for each possible list) that share these operations and observations as actions and tests, respectively. We do not require the goal in each instance in $Q$ to be expressible in terms of observables (as often assumed in generalized planning). This is because the actual goal depends on the data stored in the data structure and cannot always be captured by the outcome of the tests.

To obtain the finite-state controller, a.k.a. generalized policy in generalized planning, the potentially infinite set $Q$ of classical problems is abstracted into a single nondeterministic problem $Q^A$, which is defined over the common sets of actions and observations, and captures the transitional assumptions and the goal. However, such an abstract problem is too coarse for expressing non-trivial program synthesis tasks, so temporal assumptions are added, which are expressed as linear-time temporal (LTL) formulas (Pnueli 1977). Temporal assumptions capture necessary conditions for a trace of observations to arise from some concrete problem instance; they represent a way to rule out spurious executions, i.e., those, due to non-determinism, introduced during the abstraction step. Typical examples of such assumptions are forms of fairness of action effects (Bonet et al. 2017); here, we lift such “fairness” assumptions to arbitrary temporal restrictions on the environment (Aminof et al. 2019). Note that temporal assumptions specify properties of trace outcomes instead of data configurations (since data have been abstracted away).

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 $Q^A$ 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 $Q^A$, and in particular the temporal assumptions, in compact form as LTL formulas, so as to enable solution of the generalized problems $Q$ using 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 minimum in a list. In our examples, we use the state-of-the-art native LTL synthesis engine Strix (Meyer, Sickert, and Luttenberger 2018). Since coming up with execution assumptions is often easy but requires ingenuity, our approach is semi-automated.

## 2 Generalized Planning

The framework adopted in this paper follows the one by Bonet et al. (2017). However a crucial difference with that work is that goals in the single instances are different and not expressible directly through the observables, i.e., we do not assume that goals are a Boolean combination of observations. More precisely, a planning instance is a deterministic classical planning problem extended with an observation function. Namely, each instance $P$ defines a state model $(S, s_0, T, A, f, obs, \Omega)$ 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, $A$ is the finite set of actions, $f : A \times S \rightarrow 2^A$ is the available-actions function, $f : \times S \rightarrow S$ is the deterministic state transition function, $obs : S \rightarrow \Omega$ is the observation function, and $\Omega$ is the finite set of observations.

A solution to an instance $P$ is an action sequence $a_0, \ldots, a_{n-1}$ that generates a goal-reaching state sequence $s_0, \ldots, s_n$. For this, each action $a_i$ in the sequence must be applicable in the state $s_i$, i.e. $a_i \in A(s_i)$, the state $s_{i+1}$ must be the state that follows action $a_i$ in the 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 $Q$ is a set of instances $P$ with the same actions $A$, same observations $\Omega$, and same observable action preconditions. The latter means that for every observation $\omega \in \Omega$, there are sets of actions $A_\omega \subseteq A$ s.t. in every instance $P$ of $Q$, $A(s) = A_\omega$ if $obs(s) = \omega$.

A policy $\mu$ for a generalized problem $Q$ is a partial function that maps interleaved sequences $\tau : \omega_0, a_0, \omega_1, a_1, \ldots, a_{i-1}, \omega_i$ of observations and actions, ending in observations, into actions. A policy $\mu$ induces a unique sequence of states and actions $s_0, a_0, \ldots, s_n$ in each instance $P$ of $Q$ where

---

1 In fact, observability of action preconditions is it not strictly necessary, e.g., it is not assumed in (Hu and De Giacomo 2011), but it simplifies our treatment.
goals are expressed in the abstraction through constraints \( \Gamma_G \) on observation-action trajectories that ensure that the state-action trajectories in the instances are goal reaching:

**Definition 3.** A sound goal constraint \( \Gamma_G \) for \( Q \) is a set of observation-action trajectories \( \tau : \omega_0, a_0, \omega_1, a_1, \ldots \) of the projection \( Q^o \) s.t. every state-action trajectory \( s_0, a_0, s_1, a_1, \ldots \) of a problem \( P \) in \( Q \) that gives rise to \( \tau \), i.e., for which \( \omega_i = \text{obs}(s_i) \), reaches the goal in \( P \).

Finally, the temporal assumption \( \Gamma_F \) 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 observation trajectories (Bonet et al. 2017).

**Definition 4.** A sound temporal assumption \( \Gamma_F \) for \( Q \) is a set of observation-action sequences \( \tau : \omega_0, a_0, \omega_1, a_1, \ldots \) in the projection \( Q^o \) that includes all the infinite trajectories that arise in instances \( P \) in \( Q \).

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

**Example 1 (cont.).** For the generalized problem \( Q \) above, the states in the observation projection \( Q^o \) are the common observations, i.e., \( \text{hasNext} \) and \( \neg \text{hasNext} \). The effect of action \( \text{stop} \) in the abstraction is deterministic, i.e., it does not change the observation. However, the effect of \( \text{next} \) is non-deterministic as it can result in \( \text{hasNext} \) or \( \neg \text{hasNext} \). The sound goal-constraint \( \Gamma_G \) consists of (a representation of) all trajectories of \( Q^o \) in which \( \neg \text{hasNext} \) is observed. Finally, the sound temporal assumption \( \Gamma_F \) consists of all trajectories of \( Q^o \) in which if \( \text{next} \) is performed infinitely often then eventually \( \neg \text{hasNext} \) is observed.

We now define what it means to solve an abstraction \( Q^A \). Let \( \Gamma \) be an assumption on observation-action trajectories, e.g., an execution assumption or a goal constraint. An observation-action trajectory in \( \Gamma \) is said to satisfy \( \Gamma \).

**Definition 5.** A policy \( \mu \) solves the abstraction \( Q^A = \langle S^o, I^o, A^o, F^o \rangle \) of \( Q \) if the trajectories induced by \( \mu \) on the observation projection \( Q^o \) that satisfy \( \Gamma_F \) also satisfy \( \Gamma_G \).

The key property of a sound observation abstraction is that:

**Theorem 1.** If a policy \( \mu \) solves a sound observation abstraction \( Q^A \) of \( Q \), then \( \mu \) solves \( Q \).

**Proof.** Let \( \tau \) be an infinite state-action trajectory in an instance \( P \) of \( Q \) induced by \( \mu \). Let \( \text{obs}(\tau) \) be the infinite observation-action trajectory given rise to by \( \tau \). Then \( \text{obs}(\tau) \) is a trajectory in \( Q^o \), and it is also induced by \( \mu \). Now, \( \text{obs}(\tau) \) satisfies \( \Gamma_F \) (since \( \Gamma_F \) is sound for \( Q \)); so \( \text{obs}(\tau) \) satisfies \( \Gamma_G \) (since \( \mu \) solves \( Q^A \)) and so \( \tau \) is goal-reaching in \( P \) (since \( \Gamma_G \) is sound for \( Q \)). So, \( \mu \) solves \( Q \).

Notice that Theorem 1 is a soundness result, which is useful only when the abstraction \( Q^A \) is solvable.
Example 1 (cont.). In the example above, observe that the trajectories in $Q^o$ induced by the policy $\mu$ that satisfy the temporal assumption $\Gamma_F$ also satisfy the goal constraint $\Gamma_G$. On the other hand, $\Gamma_F$ is important since there are $\mu$-trajectories that neither satisfy $\Gamma_F$ nor $\Gamma_G$. Hence, $\mu$ would not be a solution if all $\mu$-trajectories were considered.

4 LTL Observation Abstractions

The observation abstraction $Q^A$ 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 $V$ is $\varphi ::= p | \varphi \lor \varphi | \neg \varphi | \varphi \land \varphi | \varphi U \varphi$, where $p \in V$. The semantics is given in terms of infinite sequences $\alpha \in (2^V)^\omega$ of valuations of the variables. In particular, $p$ means that $p$ is true in the current time, $\neg \varphi$ that $\varphi$ is true in the next time step, and $\varphi \land \varphi'$ that i) eventually $\varphi'$ is true, and ii) at every step up until (but not necessarily including) that time, $\varphi$ is true. We use the usual abbreviations; i.e., $\diamond \varphi$ (read “eventually”) is defined as $\top \land \varphi$, and $\Box \varphi$ (read “always”) is defined as $\neg \diamond \neg \varphi$.

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 sets the variables in some set $X$, and the controller then responds by setting the variables in some (disjoint) set $Y$, and this interaction repeats. In planning terminology, $X$ can be viewed as a representation of observations (i.e. $\Omega = 2^A$), and $Y$ as a representation of actions (i.e. $\text{Act} = 2^V$). For an LTL formula $\varphi$ over the variables $X \cup Y$, a controller policy $\mu$ solves the LTL synthesis problem for the formula $\varphi$ if every sequence of observations and actions induced by $\mu$ satisfies $\varphi$. When such a $\mu$ exists we say that $\varphi$ is realizable by the controller. Dually, we say that $\varphi$ is realizable by the environment if there is an environment policy, i.e., is a function from finite traces ending in actions to states, that induces traces that satisfy $\varphi$.

LTL observation abstractions $Q^A = (Q^o, \Gamma_F, \Gamma_G)$ 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 $Q^A$ (i.e., over the variables in $X$ and $Y$). The formulas are:

- $D$ for capturing the action preconditions of $Q^o$, the requirement on doing exactly one action at a time, and the technical fact that once the action $\text{stop}$ 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 the execution temporal constraints on the environment $\Gamma_F$ over $Q^o$;
- $G$ for capturing the goal $\Gamma_G$ 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 presence of assumptions about the environment behavior, such as our $E$ and $F$, the synthesis specification is taken to be an implication:

$$\text{Env} \supset Cntrl$$

where $\text{Env}$ are the assumptions on the environment and $Cntrl$ is the specification of controller desired behavior (under the environment assumptions $\text{Env}$). Not every LTL formula can be used for $\text{Env}$: it is required that the environment must have a strategy to win $\text{Env}$ in spite of whatever the controller does. That is the environment must be able to react, resolving its own nondeterminism, without getting stuck, to every controller action. Formally, $\text{Env}$ must be realizable by the environment, i.e., there must be an environment’s strategy that solves $\text{Env}$. We refer to (Aminof et al. 2019) for a thorough discussion.

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

- $\text{Env}$ is $(D \supset (E \land F))$, i.e., the environment guarantees that when the preconditions are satisfied, the effects and temporal assumptions hold.
- $Cntrl$ is $(D \land G)$, i.e., (when the $E$ 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 $\text{Env}$ 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 $Q^A = (Q^o, \Gamma_F, \Gamma_G)$ as above. Then $(D \supset (E \land F))$ is realizable by the environment.

Proof sketch. We need to find a strategy for the environment s.t. every induced sequence that satisfies $D$ also satisfies $E \land F$. 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 $Q$, and on its turn simply outputs the observation of the current state of $P$. That is, on the first turn, the environment outputs $\text{obs}(s_0)$. 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 $\text{obs}(f(s_0, a_0))$, 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 trace of $P$. By soundness of $\Gamma_F$, the trace satisfies $\Gamma_F$, and thus, by definition of $F$, the trace also satisfies $F$. □

Now, it is easy to see that the formula $(D \supset (E \land F)) \supset (D \land G)$ is logically equivalent to $D \land ((E \lor F) \land G)$, which is the formula used below. The latter requires the policy $\mu$ to be s.t. actions satisfy preconditions (including structural constraints) and in all induced trajectories where $E$ and $F$ are true (i.e., 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 $D \land ((E \land F) \land G)$.

---

3In these formulas, there is no syntactic distinction between observations and actions, they are all propositional variables.
The correctness of the approach is guaranteed by the results below.

**Lemma 6.** Let $D$, $E$, $F$, and $G$ be LTL formulas representing the LTL observation abstraction $Q^A = (Q^A, \Gamma_F, \Gamma_G)$ as above. Then, if the policy $\mu$ solves the LTL synthesis problem for the formula $D \land (\neg \neg (E \land F) \supset G)$, then $\mu$ solves $Q^A$.

Then, it follows from Theorem 1 that:

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

## 5 Our Framework at Work

We now show how to use the framework above 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 $Q_T$, and single out the common set of actions and observations;
2. Come up with sound temporal assumption and goal constraint for the observation abstraction $Q^A_T$ of $Q_T$;
3. Write LTL formulas $D$, $E$, $F$, $G$ characterizing the observation abstraction $Q^A_T$;
4. Automatically solve the LTL synthesis problem for the formula $D \land (\neg \neg (E \land F) \supset G)$, and output a solution/controller $\mu$ for $Q^A_T$ if there is one.

*Nota Bene.* We can prove the correctness of the approach if we have formalized $T$ (e.g., in mathematics, in logic, or in a formal specification language such as Z (Spivey 1989)). This formalization induces a generalized planning problem $Q_T$, e.g., by considering all the possible instantiations of the data structures in the formalization. We then prove that the temporal assumption and goal constraint are sound for $Q_T$, and so, deduce (by Theorem 3) that the synthesized controller $\mu$ solves $Q_T$, and hence that $\mu$ solves the programming task $T$.

We assume to have standard data structures such as singly-linked lists, doubly-linked lists, trees, graphs, along with their corresponding operations, which operate as expected (we omit details for brevity). We also assume suitable (finitely many) cursors that point to cells in the data structures (like Java iterators) and (finitely many) registers, if needed. A fixed set of sensors compare values in the cells pointed by the cursors and in the registers, resulting in a fixed set of booleans/observations. In addition, a fixed set of actions move the pointers in the structures and allow us to copy the content of one pointed cell into another. A number of elements are common among the various models, i.e.: we solve the LTL synthesis problem for the formula $D \land (\neg \neg (E \land F) \supset G)$; we use action $s(\text{top})$ as a final no-op indicating termination (once done, it repeats forever); we restrict to executing exactly one action per step, which is expressed using the XOR operator “⊕”. We also use the abbreviations: PERSISTS$(p)$, to express persistence of $p$, i.e., $(p \leftrightarrow \bigcirc p)$ and PERSIST$(p_1, \ldots, p_n)$, to express persistence of all $p_i$’s.

All controllers are obtained with the online demo version of Strix (https://strix.model.in.tum.de/try) and computed in less than 3 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) one. Actions and observations in the generalized problem $Q_T$, as well as the LTL formulas for the abstraction, are as follows:

- **Observations**: hasNext (cell pointed by cursor has a successor (next) cell)
- **Actions**: $n$ (move cursor to next cell), $s$ (top)
- **Formula $D$** (preconditions, non-concurrent actions, and stop action):
  - $\square(n \rightarrow \text{hasNext})$
  - $\diamond(n \oplus s) \land \square(s \rightarrow s)$
- **Formula $E$** (initial conditions, effects, and frame):
  - $\square(s \rightarrow \text{PERSIST(\text{hasNext})})$
- **Formula $F$** (Temporal assumptions):
  - $\langle\square \diamond n\rangle \rightarrow \langle\neg \neg \text{hasNext} \rangle$
- **Formula $G$** (Goal): visit all cells once, i.e., reach the end and stay there (we could also use $\langle \diamond \neg \neg \text{hasNext} \rangle$, as the cursor can never go back; this will change in the next examples)

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 $\neg \neg \text{hasNext}$ 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. As we will see, this is not always possible.

The formulas define a sound observation abstraction, and hence every controller solving the LTL synthesis problem for the formula $D \land (\neg \neg (E \land F) \supset G)$ also solves $Q_T$.

*Proving soundness.* To give an idea of how one can check for soundness, suppose we formalize a list as the usual abstract data type\(^3\) 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 $Q_T$ obtained by instantiating the data type in all possible ways. In particular, the $i$-th planning problem $P_i$ in $Q_T$ has observations $\neg \neg \text{hasNext}$, actions $n$ and $s$, states in $\{1, 2, \ldots, i\} \times \{\text{done}, \neg \text{done}\}$, and the initial state is $(1, \neg \text{done})$; in addition, if we refer to the first component of the state as its position, the goal states have position $0$.

\(^3\)For this example, we can ignore the values in the cells of the list. In later examples, we use such values.
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 \(n(\text{ext})\) or \(p(\text{previous})\) cell (if present), \(hn\) (has next) and \(hp\) (has previous) model the fact that a cell has a successor/previous cell. Actions and observations in the generalized problem \(Q_T\), as well as the LTL formulas modeling the abstraction, are as follows:

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

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

Here, contrarily to the singly-linked case, memory is required. E.g., 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 \(p(\text{previous})\) (resp. \(n(\text{ext})\)) then all previous (resp. successor) cells have already been visited, thus the cursor must move to previous (resp. next) cell. To distinguish 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 to visit 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 containing the root only.

- Observations: \(em(\text{pty})\) (memory is empty), \(ha(s, \text{children})\) (current node’s children have not been put in memory)
- Actions: \(e(\text{xtact a node from memory, for visiting}), p(\text{ut all children of current node into memory}), s(\text{top})\)
- Formula \(D\) (preconditions, non-concurrent actions, and stop action):\n  - \(\Box (e \rightarrow \neg em) \land \Box (p \rightarrow ha)\)
  - \(\Box (e \oplus p \oplus s) \land \Box (s \rightarrow o s)\)
- Formula \(E\) (initial conditions, effects, and frame):
  - \(\neg em\)
  - \(\Box (p \rightarrow \Diamond (\neg em \land \neg ha))\)
  - \(\Box (s \rightarrow (\text{PERSISTS}(em) \land \text{PERSISTS}(ha)))\)
- Formula \(F\) (Temporal assumption):
  - \(\Box (e \rightarrow \Diamond (em \land \neg ha))\)
- 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
  - \(\Box (em \land \neg ha)\)
  - \(\Box (ha \rightarrow (\neg e \cup p))\)

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 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.

![Controller for traversing a singly linked list.](image)

![Controller for traversing a doubly linked list.](image)
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 current node’s children are in memory, and memory is nonempty; in $q_3$ the memory is empty and all current node’s children are in memory, i.e., node is leaf. The controller prescribes: to extract a node whenever memory is nonempty and all current node’s children are in memory (paths $q_0 \rightarrow q_1$, $q_1 \rightarrow q_1$, and $q_2 \rightarrow q_1$); put all current node’s children in memory whenever they are not ($q_0 \rightarrow q_2$, $q_1 \rightarrow q_2$); stop on empty memory and current node with all children in memory, i.e., node is a leaf ($q_2 \rightarrow q_3$, $q_1 \rightarrow q_3$).

Traversing a Graph. Graph traversal requires to visit all nodes of a finite graph 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:
  - $((\Box eq) \land (\Box\neg p)) \rightarrow \Box\neg(em)$
  - $(\Box p) \rightarrow \Box\neg(ha)$
- 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 while at some point stops putting nodes there. The obtained controller is essentially identical to that for tree traversal (see Fig. 3) except that symbols $ha(s_{children})$ and $p(ut_{children})$ replaced by $ha(s_{unmarked})$ (neighbors) and replaced by $p(ut_{unmarked})$ (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 currently pointed cell, and one can check whether the value in the pointed cell is less than the value in the register. The abstraction is a slight change from the one used for singly-linked list traversal:

- Observations: $hasNext$ (as in list, $lt$ (value stored in pointed cell is less than that 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 $\Box(n \oplus u \oplus s)$ replaced by $\Box(n \oplus u \oplus s)$
- Formula $E$ (initial conditions, effects, and frame):
  - $\Box((u \rightarrow (\Box(\neg lt) \land persist(hasNext))))$
  - $\Box(s \rightarrow (persist(hasNext,lt)))$
- 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 to update the value in the register iff it is greater than that in the current cell. The controller is shown in Figure 4. It prescribes to move to next cell until a cell with a value less than that in the register is pointed (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 to check 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 sought. The goal is to stop the search when the current node’s label matches the searched value 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: $c(xtract)$, $p(ut_{children})$ 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 $s$
  - $\Box(s \rightarrow persist(em, ha, eq))$
- Formula $F$ (Temporal assumptions): as in tree traversal
- Formula $G$ (goals): search value, i.e., same as tree traversal, but stop when value is found
\[
- (\Diamond \Box eq) \lor (\Diamond \Box (em \land \neg ha))
- \Box ((ha \land \neg eq) \rightarrow ((e) \lor p))
- \Box (eq \rightarrow s)
\]

The goal formulas are the interesting ones. The first 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 given to students when learning 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 cells from some leftmost cells to some rightmost cell. In our specification, only cells with land are explicitly labelled, and water assumed in all other. In this variant, we focus only on checking whether the rightmost column is reached, 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 that. 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 is empty), ha(s(neighbors)) (current node has some neighbor not put in memory), r(rightmost) (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):
  - \(\Box (e \rightarrow \neg em) \land \Box (p \rightarrow ha)\)
  - \(\Box (e \lor p \lor s) \land \Box (s \rightarrow \circ s)\)
- **Formula E** (initial conditions, effects, and frame):
  - \(\neg em\)
  - \(\Box (p \rightarrow O (\neg em \land \neg ha))\)
  - \(\Box (s \rightarrow \text{PERSISTS}(em, ha, r, l))\)
- **Formula F** (Temporal assumptions): By continuously extracting nodes from memory, it will eventually become empty:
  - \(\Box (\Diamond e) \rightarrow (e) \land \neg ha\)
- **Formula G** (goals): visit all reachable land nodes once, until a rightmost land node (if any) is achieved:
  - \(\Box (r \land l) \lor \Box (em \land \neg ha)\)
  - \(\Box ((ha \land l \land \neg r) \rightarrow (e) \lor p)\)
  - \(\Box ((r \land l) \rightarrow s)\)

Similarly to membership in a tree, the visit stops when a rightmost land cell is achieved (last formula). Here, however, only neighbors of land nodes are put in memory for future visiting (second formula), to guarantee that the nodes touched when moving toward the rightmost column are all land. Notice that here we do not prevent visiting nodes more than once. However, since the graph induced by the matrix is a DAG (it is immediate to see that there cannot be cycles), the visit cannot get stuck inside a cycle. This, however, could have easily been prevented by enforcing a marking mechanism similar to that adopted for graph traversal (which, in fact, requires only a different concrete instantiation of actions extract and put). The obtained controller has a structure similar to that of Fig. 5.

### 6 Conclusions

We have addressed the problem of automatically constructing a concrete program (in the form of a finite-state controller), starting from a high-level specification of the assumptions and the goal a designer would take into account when devising the program manually. The approach capitalizes the effort made by the designer when coming up with the specification, a task he would perform anyway, although informally, in his mind: by formalizing the specification in LTL, the implementation step can be carried out automatically. We mention that the whole approach has been implemented and experimented by students as part of a graduated course. The implementation takes the input specification, runs the LTL solver and translates the output controller into an actual Python program. The implementation has been used to successfully solve several programming problems, including: find minimum/maximum of a list, copy even numbers in another list, sum of positive numbers of a list, bubble sort, find an element in tree, and robot navigation to position in a matrix.

Other approaches oriented towards automated programming have been previously proposed, where planners have been used to derive general programs and controllers, but from examples (Bonet, Palacios, and Geffner 2009; Aguas, Celorrio, and Jonsson 2019). Here, instead, the abstractions are explicitly formalized, in LTL. 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 solved the problem with general LTL tools, to cope with the increased modeling power obtained by adding unrestricted temporal assumptions. From the planning perspective, the challenge is to replace the LTL synthesis tool, 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 recursion 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. The authors thank Edoardo Cicero, Daniele Giunta, and Luca Pierdicca for implementing and testing the approach in an actual Python-based setting.

References


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


