Reactive Synthesis using Sketching

Garvit Juniwal
UC Berkeley
Email: garvitjuniwal@eecs.berkeley.edu

Abstract—We first formalize the idea of using sketching (completing partial programs using specifications) in reactive synthesis. Then we develop a sound and complete approach to solve the problem. We also present a couple of examples that demonstrate the boundaries/advantages/disadvantages of this approach.

I. INTRODUCTION

The classic techniques of synthesizing from LTL specifications use algorithmic game solving to look for correct implementations. In a recent work [1], Finkbeiner and Schewe fix some bounds on some system parameters like the number of states and use SMT solving to look for implementations within that bound, incrementally relaxing the bounds. But in all these approaches, the implementations generated are typically finite state systems which are too large and unreadable.

Most exciting area to apply reactive synthesis techniques is robotics. We observe that most of the robot controllers that are used in practice are just conditional updates to the control/output variables in a loop. For eg. consider a coffee machine that is supposed to respect the following specification.

\[ G (\text{button} \& \neg \text{brew} \& \neg \text{grind}) \implies X (\neg \text{brew} \& \text{grind}) \& XX (\neg \text{brew} \& \text{grind}) \& XXXX (\neg \text{grind} \& \text{brew}) \& XXXX (\text{brew} \& \text{grind}) \]

which means that upon a button press, an idle coffee machine should grind for the next two time steps and brew for the further next two time steps and then stop.

This motivates the idea of using sketching for reactive synthesis. We can start with partial programs of this general form (formalized later in the paper) and expect to complete them using the specification. The overall idea is to synthesize complete programs which are correct for all traces of length less than a certain bound and then verify the synthesized programs in the unbounded setting. We use SKETCH [2] for the bounded synthesis step and NuSMV [3] for the verification step.

II. THE SYNTHESIS PROBLEM

The synthesis problem is to decide whether or not there exists an implementation that satisfies a given specification. In the following, we formalize this problem by using LTL as our specification language and partial programs as representations of implementations.

A. Specifications

We use LTL as the specification logic.

Syntax An LTL formula \( \phi \) is defined over a set of atomic propositions \( AP \). An LTL formula has the following syntax.

1) \( \psi \in AP \) is an LTL formula
2) If \( \psi \) and \( \phi \) are LTL formulae then so are \( \neg \psi, X \psi, \psi U \phi, \psi \land \phi, \psi \lor \phi \).

The operators are the next time operator \( X \), the until operator \( U \), and its dual is the release operator \( R \).

Each formula defines a set of infinite words over \( 2^{AP} \). Let \( \pi \in (2^{AP})^\omega \) be an infinite word. We denote the suffix of a word by \( \pi^i = \sigma_i \sigma_{i+1} \sigma_{i+2} \cdots \) where \( \sigma_i \in 2^{AP} \), and \( \pi_i \) denotes the prefix \( \pi_i = \sigma_0 \sigma_1 \cdots \sigma_i \). When a formula defines a word \( \pi \) at time \( i \), this is denoted by \( \pi^i \models \psi \). The set of infinite words defined by a formula \( \psi \) is \( \{ \pi \in (2^{AP})^\omega | \pi \models \psi \} \). The relation \( \models \) is inductively defined in the following way.
The first part of the program declares all the variables with their roles and types.

\[ P(X) = \{ b \mid b \text{ is generated by } B\text{Exp}(X) \} \]
\[ P_\alpha = P(I \cup O) \]

\[ π^i \models ψ \iff ψ \in σ_i \text{ for } ψ \in AP. \]
\[ π^i \models ψ \iff π^i \not\models ψ. \]
\[ π^i \models ψ ∧ φ \iff π^i \models ψ \text{ and } π^i \models φ. \]
\[ π^i \models ψ ∨ φ \iff π^i \models ψ \text{ or } π^i \models φ. \]
\[ π^i \models Xψ \iff π^{i+1} \models ψ. \]
\[ π^i \models ψ U φ \iff ∃n \geq i \text{ such that } π^n \models φ \]
\[ \text{and } π^j \models φ \text{ for all } i \leq j < n. \]
\[ π^i \models ψ R φ \iff ∀n \geq i, π^n \models φ \]
\[ \text{or } π^j \models φ \text{ for some } i \leq j < n. \]

If \( π^0 \models ψ \) we simply write \( π \models ψ \). This presentation of the semantics is intentionally redundant. The additional operators allows us to transform any formula to a positive normal form. Formulas in positive normal form have negations only in front of atomic propositions. Using the dualities \( ψ U φ \equiv ¬(¬ψ R ¬φ) \), \( ¬Xψ \equiv X¬ψ \) and DeMorgan’s law any formula can be transformed without blowup to positive normal form. All the formulas considered in this report are assumed to be in positive normal unless otherwise mentioned. We also make use of standard abbreviations \( T \equiv p ∨ ¬p \) for some arbitrary \( p \in AP \), \( ⊥ \equiv ¬T \), \( Fψ \equiv T U ψ \) (‘finally’), and \( Gψ \equiv R U ψ \equiv ¬F¬ψ \).

### B. Implementation

Given a set of input variables \( I \), output variables \( O \) and machine variables \( S \), a reactive system can be imagined to have the following structure

\[
\begin{align*}
I &:= \text{scan}(); \\
O, S &:= \text{initialize}(I); \\
\text{repeat} \\
& \quad \text{I := scan();} \\
& \quad \text{O, S := update(I, O, S);} \\
\end{align*}
\]

The system scans for values of the input from the environment and updates the output variables in each step. More concretely, we consider programs from the following grammar.

\[
\text{Program} ::= \text{Decls} \; \text{Inits} \; \text{Controller}
\]

This defines the sets of typed variables \( I, O \) and \( S \). Let \( V = I ∪ O ∪ S \). Now the rest of the program comes from

\[
\begin{align*}
\text{Inits} & ::= \text{List of } \text{Init}(v) \text{ for all } v \in V \\
\text{Init}(v) & ::= \text{Identifier}(v) ::= \text{Exp}(I) \\
\text{Controller} & ::= \text{Stmt} \\
\text{Stmt} & ::= \text{if}(\text{BExp}(V)) \text{ then } \text{Stmt} \\
& \quad \text{else } \text{Stmt} \mid \text{Update} \\
\text{Update} & ::= \text{List of } \text{Assign}(v) \text{ for } v \in O ∪ S \\
\text{Assign}(v) & ::= \text{Identifier}(v) ::= \text{Exp}(V)
\end{align*}
\]

The expressions come from the following grammar

\[
\begin{align*}
\text{Exp}(V) & ::= \text{BExp}(V) \mid \text{AExp}(V) \\
\text{BExp}(V) & ::= \text{BExp}(V) \: \text{Bop} \: \text{BExp}(V) \\
& \quad \mid \text{Exp}(V) \: \text{Cop} \: \text{Exp}(V) \\
& \quad \mid \text{BAtom}(V) \mid \text{true} \mid \text{false} \\
\text{Bop} & ::= \& \mid ∨ \\
\text{Cop} & ::= < | ≤ | > | ≥ | == | ≠ \\
\text{BAtom}(V) & ::= v, \text{for } v \in V, \text{type}(v) = \text{bit} \\
\text{AExp}(V) & ::= \text{AExp}(V) \: \text{Aop} \: \text{AExp}(V) \\
& \quad \mid \text{AAtom}(V) \mid \text{const}, \text{const } \in \mathbb{N}_c \\
\text{Aop} & ::= + | - \\
\text{AAtom}(V) & ::= v, \text{for } v \in V, \text{type}(v) = \text{int}
\end{align*}
\]

where, \( \mathbb{N}_c \) is integers representable in \( c \) bits. It is straightforward to see how this program fits in the paradigm defined above.

Given such a program \( \alpha \) over \( I, O, S \) as input, output, machine variables, we define the set of propositions \( P_\alpha \) of the system as the set of all possible boolean expressions over the input and output variables of the system.

\[
\begin{align*}
P(X) & = \{ b \mid b \text{ is generated by } B\text{Exp}(X) \} \\
P_\alpha & = P(I \cup O)
\end{align*}
\]
Since we work only on bounded integers, this set is finite. Also, define the notion of state of the program as an assignment to all the variables in the program. Notice that we care about the state of the program only at the points where it has fully executed the controller code. Let $\Sigma_\alpha$ be the state space of the program, and $\sigma_0^\alpha$ be the state immediately before reading the first input. Valuations of all the variables in this state is $\perp$. For $\sigma \in \Sigma_\alpha$ and a valuation $I$ to the input variables $I$, $[[\text{Controller}]]\sigma$ denotes the state after executing the controller once on state $\sigma$(similar interpretation for $[[\text{Inits}]]\sigma$) and $\sigma[I := I]$ denotes the state with values of the input variables replaced with the values from $I$. We can define a transition system for this program in the form of a Kripke structure with $\mathbb{P}_\alpha$ as the set of atomic propositions. $M_\alpha = (2^{\Sigma_\alpha}, S_0^\alpha, T_\alpha, L_\alpha)$.

$$\begin{align*}
\sigma \in S_0^\alpha & \iff \exists I, \text{ a valuation of inputs } I \\
[[\text{Inits}]](\sigma_0^\alpha[I := I]) & = \sigma \\
(\sigma, \sigma') \in T_\alpha & \iff \exists I, \text{ a valuation of inputs } I \\
(([[\text{Controller}]]\sigma)[I := I] & = \sigma' \\
L_\alpha(\sigma) & = \{ b \in \mathbb{P}_\alpha | \sigma \models b \}
\end{align*}$$

Note that the transition relation is total since for every state we can choose any input and have a next state.

Now we can write any LTL specification $\psi$ using $\mathbb{P}_\alpha$ as the set of atomic propositions. $\alpha \models \psi \iff$ all infinite runs of $M_\alpha$ starting in the initial state are models of $\psi$.

### C. Partial Implementations

The idea of using sketching involves starting with partial implementations and complete them using the specifications. The partial implementations we start with are of the following form. All other grammar rules remain the same except

$\begin{align*}
\text{Stmt} & ::= \text{if}(BExp??(V)) \text{ then } \text{Stmt} \\
& \text{ else } \text{Stmt} \mid \text{Update} \\
\text{Update} & ::= \text{List of Assign}(v) \text{ for } v \in O \cup S \\
\text{Assign}(v) & ::= \text{Identifier}(v) := EExp??(V)
\end{align*}$

$\begin{align*}
EExp??(V) & ::= BExp??(V) \mid AExp??(V) \\
BExp??(V) & ::= BExp??(V) \text{ Bop } BExp??(V) \\
& \text{ !BExp??(V)} \\
& \text{ AExp??(V) Cop } AExp??(V) \\
& \text{ BAtom}(V) \text{ true } \mid \text{ false} \\
& \text{ ?}??(V)
\end{align*}$

\[
L((\sigma_0^\alpha I := I)) = \sigma
\]

Also, define the notion of state of the program as an assignment to all the variables in the program.

where, $\mathbb{N}_c$ is integers representable in $c$ bits.

??(V) is hole in the implementation that is filled with an expression on type $t$ and can use only symbolic variables $V$. The set of possible expressions that could replace a hole is written using the SKETCH language which allows for use of recursive generators and regular expressions. These holes can be characterized in three classes shown below by examples.

- **Integer Holes**

$$??(c)$$

where $c$ is a fixed known integer. This hole can take as value any integer that can be represented in $c$ bits. Clearly, this hole is of integer type and does not use any symbolic variables.

- **Regular Expression Holes**

$$\{ ||v_1|v_2|\ldots|| \}$$

where, $v_1, v_2, \ldots$ are symbolic variables or constants or integer holes. This just means that you can pick any of $v_1, v_2, \ldots$ as a value for this hole. Note that the types of $v_1, v_2, \ldots$ must be the same and that is the type of this hole.

- **Generators**

\[
\begin{align*}
\text{generator bit} \\
\text{cond(int x, int y, int bnd)} \{ \\
& \text{assert bnd>0;} \\
& \text{if(??) return} \\
& \quad \{ ||(x|y) (>_<|=|!=|<=|>=) \\
& \quad \quad (x|y|??) || \}; \\
& \text{if(??) return} \\
& \quad \text{cond(x, y, bnd-1) &&}
\end{align*}
\]
This is an example of a generator for a bit hole which takes two integer variables \( x \) and \( y \) and a parameter \( bnd \). This hole can take as its values any boolean expression with a syntax tree of depth \( bnd \), involving \( x \) and \( y \) and constants and only conjunction as the operator.

D. Problem Definition

Given a partial implementation of a program over \( I \), \( O \), \( S \) as the input, output, machine variables. And a LTL specification \( \psi \) over the set of atomic propositions \( P(I \cup O) \), complete the partial implementation such that the completed implementation \( \alpha \models \psi \).

III. Bounded LTL Model Checking

In bounded model checking we consider finite sequences of states in a system, while LTL formulae specify the infinite behavior of the system. The key observation by Biere et al [4] was that a finite sequence can still represent an infinite path if it contains a loop. An infinite path \( \pi = s_0s_1\cdots \) is a \( (k,l) \)-loop if there exists integers \( l \) and \( k \) such that \( s_{l-1} = s_k \) and \( \pi = (s_0s_1\cdots s_{l-1})(s_l\cdots s_k)^\omega \) (we also use the term \( k \)-loop).

See figure 1 When \( k \) is fixed there are \( k+1 \) possibilities for a bounded path. There are \( k \) different \( (k,l) \)-loops and it is of course possible that no loop exists. The basic idea is to write a formula which is valid iff the path is a model of the LTL formula. The complete translation is just a case split on all these possible looping cases.

We define the bounded semantics for LTL to work on these bounded words. Given an infinite path \( \pi \) and a bound \( k \in \mathbb{N} \), a formula \( \psi \) holds in a path \( \pi \) with bound \( k \) iff \( \pi \models^0_k \psi \) where

\[
\begin{align*}
\pi \models^i_k p & \iff p \in s_i \text{ for } p \in AP. \\
\pi \models^i_k \neg p & \iff p \notin s_i \text{ for } p \in AP. \\
\pi \models^i_k \psi \land \phi & \iff \pi \models^i_k \psi \text{ and } \pi \models^i_k \phi. \\
\pi \models^i_k \psi \lor \phi & \iff \pi \models^i_k \psi \text{ or } \pi \models^i_k \phi.
\end{align*}
\]

Note that we have used optimistic semantics for the no loop cases for the \( \text{U} \) and \( \text{R} \) operators. This means that a formula like \( \text{G} \psi \) will be true in the no loop case iff \( \forall i, 0 \leq i < k : \pi \models^i_k \psi \). It might be possible that in the future states, \( \psi \) might become false. Similarly, \( \text{F} \psi \) is always true in the no loop case because we assume optimistically that \( \psi \) will become true in the future states. This means, that if an LTL formula is true on \( \pi \) over bounded semantics then it might be false in the unbounded semantics but if the the formula is false in the bounded semantics, then we have clearly found a counter example for the unbounded semantics as well.

Let \( M \) be the Kripke structure of the system and \( I(s) \) be the symbolic initial states predicate and \( T(s, s') \) be the symbolic transition relation. We consider the unrolling of states \( s_0s_1\cdots s_k \). Each \( s_i \) is a vector of the state variables. The unrolling is obtained by the following formula

\[
[[M]]_k := I(s_0) \land \bigwedge_{i=1}^{k} T(s_{i-1}, s_k)
\]  

(1)

We consider the case where the unrolling is a \( (k,l) \)-loop. The translation in Table I creates a boolean formula which encodes the bounded semantics.

The auxiliary translation \( \langle \cdot \rangle \) which computes the approximations of the fix points is defined in the last two rows. \( p(s_i) \) means that \( p \) holds in state \( s_i \). For the no-loop case using optimistic semantics, the translation is written in Table II
Let $\text{loop}_i, 0 < i \leq k$ denote the boolean formula that encodes when the given unrolling of bound $k$ is a $(k, l)$-loop.

$$\text{loop}_i \Leftrightarrow (s_{i-1} = s_k)$$

The overall assertion that covers all cases is the following.

bit loop_tracker = 0;
if($\text{loop}_1$)$\{\text{assert}([\lceil \psi \rceil]_0); \text{loop}_\text{tracker} = 1;\}$
if($\text{loop}_2$)$\{\text{assert}([\lceil \psi \rceil]_0); \text{loop}_\text{tracker} = 1;\}$
.
if($\text{loop}_i$)$\{\text{assert}([\lceil \psi \rceil]_i); \text{loop}_\text{tracker} = 1;\}$
.
if($\text{loop}_k$)$\{\text{assert}([\lceil \psi \rceil]_k); \text{loop}_\text{tracker} = 1;\}$
if(!loop_tracker)$\{\text{assert}([\lceil \psi \rceil]_0);\}$

If the above assertion is true for all unrollings given by Eq 1, then we can claim that there is no counter example from $M$ representable in length $k$ for the formula $\psi$.

IV. THE BOUNDED SYNTHESIS APPROACH

The synthesis problem: Given a partial implementation $\alpha_p$ of a program over $I, O, S$ as the input, output, machine variables. And a LTL specification $\psi$ over the set of atomic propositions $P(I \cup O)$, we want to come up with a completed implementation $\alpha$, s.t. $\alpha \models \psi$. Our approach involves two steps. It is schematically depicted in Fig 2.

A. Synthesis Step

We start with a small bound $k$. It is okay to always start with $k = 1$. We unroll $\alpha_p$ for $k$ steps and record $\sigma_i, 0 \leq i \leq k$ which denotes the program state after executing the controller code $i$ times and reading the $i^{th}$ input $I^i$ as depicted below.

\[ \alpha_p :: \]

\text{Decs}

\text{Inits}

\text{Controller}

\[ \alpha_p^k :: \alpha_p \text{ unrolled } k \text{ steps} \]

\text{Decs}

$\langle \text{read input } I^0 \rangle$

\text{Inits}

$\langle \text{store program state } \sigma^0 \rangle$

\text{Controller}

$\langle \text{read input } I^1 \rangle$

$\langle \text{store program state } \sigma^1 \rangle$

\ldots

\text{Controller}

$\langle \text{read input } I^i \rangle$

$\langle \text{store program state } \sigma^i \rangle$

\ldots

\text{Controller}

$\langle \text{read input } I^k \rangle$

$\langle \text{store program state } \sigma^k \rangle$
In this way, we encode all runs of $M_{\alpha_p}$ of length $k$ starting from the initial state. Now we can assert the LTL spec $\psi$ over $P_{\alpha_p}$ in the bounded semantics using the assertion described in the previous section using

$$\text{loop}_i \iff (\sigma^{i-1} = \sigma^k)$$
$$b(\sigma^i) \iff \sigma^i \models b$$

Note that, we constrain the corresponding holes in each copy of the controller in the unrolling to hold the same value because we need each copy of the controller to look the same.

Now, this unrolled program with holes and the assertion that validates the bounded semantics of $\psi$ for bound $k$ can be fed to SKETCH, which essentially solves the following constraint to give a complete program from $\alpha_p$.

$$\exists \text{Holes} \forall I_0 \cdots I_k \alpha_p(\text{Holes}) \models_k \psi$$

where $\alpha_p(\text{Holes})$ is a complete program with Holes plugged in $\alpha_p$.

- If SKETCH rejects the partial program, it means that for no values of the allowed holes can the assertion be valid. Now, the user can change the partial program in some way to make it feasible and repeat the synthesis step.
- If SKETCH returns a complete program, lets call it $\alpha$. We can claim that there are no counterexamples to $\psi$ in $\alpha$ of length atmost $k$. But this clearly does not mean that there cannot be larger counterexamples, for which we need the verification step.

B. Verification Step

Given a complete program $\alpha$, we use the symbolic model checker NuSMV to check whether it satisfies $\psi$ in the unbounded semantics. Translating the programs from our grammar into NuSMV state machines is straightforward.

- If NuSMV declare the program to be correct, then we are done and the loop exists. $\alpha$ is the required program
- If NuSMV generates a counterexample (which has to be of length greater than bound $k$ from the previous step), we revise our bound to the length of the counterexample and restart the synthesis step.

C. Guarantees about the approach

Since we have a verifier in the loop, it is very easy to see that

Claim 1. This approach is sound in the sense that the synthesized complete program is correct with respect to $\psi$.

Since the state space of our programs is finite, we can claim the following.

Claim 2. For any partial program $\alpha_p$ and LTL formula $\psi$ over $P_{\alpha}$, there exists a bound $k_{\alpha_p}$ for which we get a correct complete program $\alpha$, if there exists one. We can also say the $k_{\alpha_p} \leq |\Sigma_{\alpha_p}|$, since with $|\Sigma_{\alpha_p}|$ many unrollings, we capture all behaviors of the program. This approach is complete in this sense.

V. EVALUATION

We test this approach on various examples in robot path planning. We present the examples in increasing order of complexity.

A. Robot on a Grid

Consider a robot moving on a square grid with an objective to reach a given final destination starting at any point in the grid without falling off the grid. The robot can only move right, left, up or down in each time step. See figure 3.

Inputs: initial position: $(ix, iy)$
destination position: $(dx, dy)$
System Variables: position: $(x, y)$
high edge of the grid: $h$
low edge of the grid: $l$
Partial Program:

$$x:=ix, \ y:=iy$$
$$\text{repeat until}(x==dx \ \&\ \ y==dy)\{$$
  if(??) $x:=x+1$; //move right
  elsif(??) $x:=x-1$; //move left
  elsif(??) $y:=y+1$; //move up
  else $y:=y-1$; //move down
$$\}$$

We wish to synthesize the conditional expressions. We allow the holes to take values from all possible boolean expressions involving all the system and input variables using the generator as described in section II-C allowing depth 2 expressions in this case. Specification states that the robot should not fall off the grid and should eventually reach the destination.

Spec: $G( l=x=h \ \&\ l=y=h)$ & $F( x=dx \ \&\ y=dy)$
The tool is able to produce a completed implementation.

\[
\text{Partial Program:} \\
\begin{align*}
x &:= \text{ix}, \; y := \text{iy} \\
\text{repeat until} \ (x{=}dx \& \& y{=}dy) \{ \\
\quad \text{if} (x{<}dx) \; x := x+1; \quad \text{//if D is on right} \\
\quad \text{elsif} (x{>}dx) \; x := x-1; \quad \text{//if D is on left} \\
\quad \text{elsif} (y{<}dy) \; y := y+1; \quad \text{//if D is up} \\
\quad \text{else} \; y := y-1; \quad \text{//} \\
\}\end{align*}
\]

Unrolling depth required : 1
Running time: 2788ms

B. Robot on a Grid with Obstacle

Building on to the previous example, we assume that there is at most one fixed obstacle on the grid whose location is not known to the robot. We also assume that the robot can sense whether the obstacle is left, right, up or down relative to its own position.

Inputs: ...

obstacle location \((ox, oy)\)

Note that although location of the obstacle is treated as input, we only use it to model the proximity sensors by only allowing comparisons of \((ox, oy)\) with \((x{\pm}1, y)\) or \((x, y{\pm}1)\) in the generators of conditional expressions.

1) No Destination Requirement: We first consider a simple case where we just expect the robot to keep moving on the grid without falling off and avoiding the obstacle. No requirement to reach the destination.

\[
\text{Partial Program:} \\
\begin{align*}
x &:= \text{ix}, \; y := \text{iy} \\
\text{repeat} \{ \\
\quad \text{if}(??) \; x := x+1; \quad \text{//move right} \\
\quad \text{elsif}(??) \; x := x-1; \quad \text{//move left} \\
\quad \text{elsif}(??) \; y := y+1; \quad \text{//move up} \\
\quad \text{else} \; y := y-1; \quad \text{//move down} \\
\}\end{align*}
\]

Spec: \( G( l{\leq}x{\leq}h \& l{\leq}y{\leq}h) \& G!( x{=}ox \& y{=}oy) \)

Again the holes are allowed to take all possible boolean expressions of depth 2 involving \(x, y, l, h\) and the particular comparisons with \(ox, oy\) as described before to model the proximity sensors. We also have the environment assumption that the initial position of the robot and the position of the obstacle cannot be the same. Completed program looks like

\[
\text{Partial Program:} \\
\begin{align*}
x &:= \text{ix}, \; y := \text{iy} \\
\text{repeat until} \ (x{=}dx \& \& y{=}dy) \{ \\
\quad \text{if}(??) \; x := x+1; \quad \text{//move right} \\
\quad \text{elsif}(??) \; x := x-1; \quad \text{//move left} \\
\quad \text{elsif}(??) \; y := y+1; \quad \text{//move up} \\
\quad \text{else} \; y := y-1; \quad \text{//move down} \\
\}\end{align*}
\]

Unrolling depth required: 1
Running Time: 2949ms

Interesting thing to note about these controllers is that they are symbolic in the grid edge bounds. Although we use concrete values of \(l=1\) and \(h=3\) to synthesize them, but since we allow only particular kind of boolean expressions for the holes, the synthesized implementation is generalizable to any grid sizes.

2) Adding Destination Requirement: Building upon the previous case, we add the requirement in the specification that the robot should eventually reach the destination. The holes are allowed to take all possible boolean expressions as in the previous example but with an increased depth of 4 to allow for more complex conditions.

Partial Program:

\[
\begin{align*}
x &:= \text{ix}, \; y := \text{iy} \\
\text{repeat until} \ (x{=}dx \& \& y{=}dy) \{ \\
\quad \text{if}(??) \; x := x+1; \quad \text{//move right} \\
\quad \text{elsif}(??) \; x := x-1; \quad \text{//move left} \\
\quad \text{elsif}(??) \; y := y+1; \quad \text{//move up} \\
\quad \text{else} \; y := y-1; \quad \text{//move down} \\
\}\end{align*}
\]
Spec: G( l<=x<=h & l<=y<=h) &
G!( x=ox & y=oy) &
F( x=dx & y=dy)

This partial program is rejected by SKETCH for an unrolling depth of 1. This means that no possible expressions of the conditionals from the allowed expressions can result in a correct program.

Running time until rejected: about 300s

3) With Two Modes: Since there is no way to fill the template in the given form, we modify the template to have two modes. (We know of a correct implementation with two modes that works for this specification, we are just trying to see if it can be synthesized).

System Variables: ...
mode of operation: mode

Partial Program:

x:=ix, y:=iy
repeat until(x==dx & y==dy){
  if(mode == 1){
    if(??) x:=x+1; //move right
    elsif(??) x:=x-1; //move left
    elsif(??) y:=y+1; //move up
    elsif(??) y:=y-1; //move down
    else mode=2
  }
  elsif(mode == 2){
    if(??) x:=x+1; //move right
    elsif(??) x:=x-1; //move left
    elsif(??) y:=y+1; //move up
    elsif(??) y:=y-1; //move down
    else mode=1
  }
}

We allow boolean expressions of depth 3, since we know that there exists a controller of this form which uses only depth 3 conditionals. We are not able to synthesize the correct implementation from this partial program within a reasonable amount of time. The best we could do is

Unrolling Depth : 3
Running time to synthesize : 6495s
NuSMV generates a counter example of length : 9

Please note that we are still working on the concrete 3x3 grid in the synthesis step but we are looking for generalizable controllers. The state space size with a 3x3 grid is 18 (9 locations, 2 modes). So an unrolling depth of 18 would be enough to synthesize the correct controller but running it for depth 18 is infeasible. Also note that we can easily add more hints in the partial implementation to drive the tool towards the correct implementation but that assumes that we know that correct implementation in advance, hence making the experiment very artificial.

VI. CS219C

This project applied the idea of bounded model checking that was covered extensively in class to the synthesis domain.

VII. CONCLUSION AND FUTURE DIRECTIONS

We conclude that using sketching in reactive synthesis wins over the existing LTL synthesis techniques in the following ways

- It allows us to synthesize implementations which are much more readable
- It allows a lot of flexibility in specifying the template of the implementation and the allowed values of holes, so users can encode their intuition about the problem in the partial program, like we did in examples demonstrated. This also leads to simpler specifications. For eg. since we encoded the moves of the robot on the grid in our template, we no longer have to specify the legality of robot moves in the specification which you would have to do in cases where no templates are allowed.
- It allows the user to synthesize much more generalizable implementations by treating the variables symbolically and restricting the form of allowed hole values as demonstrated in the examples. Any existing approach for LTL synthesis would generate implementations very specific to the grid size.

Inspite of these advantages, the approach clearly is not smarter than the programmer since it has problems of scaling. It is also very sensitive to the form of templates used and the allowed values of holes which makes it less usable. It might be interesting to think about whether these templates can in turn be meta-synthesized from some input output examples making the job of the user much easier.
ACKNOWLEDGMENT

The author would like to thank Prof. Sanjit Seshia, Prof. Ras Bodik and Ruediger Ehlers for all the valuable discussions and directions.

REFERENCES


