Program Synthesis

OPLSS 2023

Anshuman Mohan, Patrick LaFontaine, Cole Kurashige, Wenjia Ye

July 2023

1 Introduction

1.1 What is synthesis?

To verify a program, one writes a program and a specification for the program, and then produces a proof that the program obeys its spec.

The key argument of *program synthesis* is that this is:

1. Too much work
2. Likely to lead to trouble

Why not have the user write *just the specification*, and then automatically generate a verified implementation from the specification? The first benefit is clear to see: we got an implementation for free. The second is more subtle and powerful: the synthesized program is *correct by construction*.

From a developmental point of view, synthesis can help a user arrive at a strong, tight specification: it can engage in a sort of Socratic dialogue with the user, pointing out the subtle ways that the spec is unrealizable. Think of it as a corner-case sniffer-outer. The user can then decide if that corner-case was spurious (in which case they can ignore it by tweaking their spec) or not (in which case they may decide your spec is indeed unrealizable for some deep reason).

1.2 The plan

§2 Reactive Synthesis

§3 Deductive Synthesis

§4 Constraint Based Synthesis

§5 Synthesis for Modular Verification

§6 New Applications

2 Reactive Synthesis

2.1 Introduction

2.1.1 The big picture

In reactive synthesis, we take a logical specification ϕ and produce automatically a circuit that realizes this specification.

Why synthesize? It is hard to create correct circuits.

Why a logical specification? It is a compact way to represent the myriad requirements of real circuits.

What use does it have? Circuits synthesized using reactive synthesis are used in embedded processors! For example, the ARM AMBA.

2.1.2 Simple circuit synthesis

To better motivate the *reactive* part of Reactive Synthesis, let's first talk about synthesizing a circuit with no temporal constraints.

What does this mean? Let us for demonstration purposes define a simple circuit synthesis problem. If you are not familiar with circuits, you can think of a circuit as a logical formula such as $p \vee \neg q$ where p and q are inputs to the circuit.

Definition (Simple Circuit Synthesis Problem)**.** *Given a input vector of booleans $\bar{i}$ and an output o, synthesize a circuit C such that when given the input $\bar{i}$, C produces the output o (denoted $C(\bar{i}) = o$).*

If you are familiar with circuits, this should seem simple (perhaps even trivial) to write a synthesizer for. Just check if the input is $\bar{i}$ and return the right output o.

If you are not familiar with circuits, you can think of this as just hardcoding the answer - it is doable and simple.

Let us consider a more complicated example.

Definition (Less Simple Circuit Synthesis Problem)**.** *Given a set of input-output pairs $\{(\bar{i}, o)\}$, synthesize a circuit C such that for each pair $(\bar{i}, o)$, $C(\bar{i}) = o$.*

We basically have generalized our problem to one where there is more than one input and output that must be satisfied. Again, there is a simple naive solution to this problem, which, if you do not know, is essentially hardcoding.

If it interests you, there are also more complex solutions which result in smaller circuits, such as Karnaugh maps, but the takeaway here is that even in this case we can easily synthesize *a* circuit, no matter how "bad."

2.1.3 Reactive synthesis

The above synthesis problems are instantiations of the Church Synthesis Problem, first proposed by Alonzo Church in 1962.

Remark (Church Synthesis Problem)**.** *Given a requirement ϕ on the input/output behavior of a Boolean circuit, compute the circuit c that satisfies ϕ.*

In the simple examples discussed in Section 2.1.2 we assumed circuits were essentially functions from boolean vector inputs to boolean outputs[1].

But not all circuits that we create are just functions from inputs to outputs. What if our circuit's outputs evolve over time?

To see how time complicates things, let's consider something that isn't a circuit. Let's say you have a graphical application and you want to add a button that changes color from red to blue when clicked. A time-independent specification might say "If the state is `red`, the button's color value is `0xFF0000` (the hex color for red)." But if we only used this for synthesis, there's nothing in the specification that says the program can't rapidly switch state between red and blue even when the button is not clicked. A time-sensitive specification to resolve this might say "If the state is `red`, it will stay `red` in the future until the button is clicked."

For the above example we implicitly assumed that the graphical application loops infinitely, changing its state in response to inputs. This is what is meant by *reactive*.

In this context, we need a stronger specification. Referring to the Church Synthesis Problem's framing, the ϕ we provide needs to be expressive enough to encode temporal constraints. This in turn makes synthesis more complex.

Reactive Synthesis has two main components.

1. A specification in Linear Temporal Logic (LTL) (Section 2.2)
2. A synthesis algorithm that converts the specification into a circuit (Section 2.5)

2.2 Linear Temporal Logic (LTL)

Linear Temporal Logic extends predicate logic to allow for reasoning with respect to time.

LTL	Colloquially
p	p
$X\ \phi$	next ϕ
$G\ \phi$	always ϕ
$F\ \phi$	eventually ϕ
$\phi\ U\ \psi$	ϕ until ψ

References: Lecture 1 slide 12 formalizes the syntax of LTL. Linear Temporal Logic on Wikipedia.

[1] This generalizes to boolean vector outputs too.

2.2.1 Examples

The lecture 1 slides have good visual examples on slide 11. Here are some other examples (including ones from the lecture).

Some desirable properties expressible in LTL:

1. Safety : (G $\neg\psi$) : ψ never holds
2. Liveness : (F ψ) : Eventually ψ will hold
3. G(request $\Rightarrow$ F acknowledge) : All requests get acknowledged

Here are some more examples relating to the button example in Section 2.1.3. In that example, we wanted to express the property that a button changes from red to blue when clicked.

The examples come with some simple exercises to help you build familiarity with LTL. If an exercise doesn't have an answer, you should be able to use subsequent examples to help answer it.

Disclaimer: these examples are student-generated and as a result may be factually incorrect. Exercise: verify the examples.

- (`red` U `clicked`)
  - The proposition `red` indicates that the button is colored red and the proposition `clicked` is true when the button is clicked.
  - This expresses that the button starts red and must stay that way until it is clicked.
  - Exercise: We can simplify our requirement to where the button only stops being red when it is clicked (instead of changing to blue specifically). Does this LTL proposition express this requirement?
- (`red` U `clicked`) $\wedge$ G (`clicked` $\Rightarrow \neg$`red`)
  - This adds a conjunction to the previous proposition with a new proposition which expresses that the button can never be red when it is clicked.
  - Exercise: Why must we use G in the new proposition?
  - Answer: Because `clicked` $\Rightarrow$ `red` only is valid for the state at time 0.
  - Exercise: An implementation might make `clicked` only true for the duration that the button is clicked. We might want to express that the button starts red and never becomes red again after it is clicked (again, simplifying from the requirement that it be blue thereafter). Does this property express this requirement?
  - Exercise: One way of trying to satisfy the previous exercise would be to change the proposition to add a G as in `clicked` $\Rightarrow$ (G $\neg$`red`). Does this express the desired property? (Hint: what do satsifying assignments look like?)
- Exercise: Can you think of a way to define our desired property in LTL?

2.3 Satisfiability versus realizability

Before we can talk about the reactive synthesis algorithm, we should first discuss the difference between satisfiability and realizability.

An LTL formula is *satisfiable* if there is a trace that satisfies it. A trace in this context is just an assignment of variables for all time steps. Lecture 1 slide 12 gives a formal definition of satisfiability and LTL (as do sites like Wikipedia).

It might be tempting to say that we can synthesize a circuit if the LTL formula is satisfiable, but that doesn't quite make sense. This is because asking for a satisfying assignment assumes that the circuit has control over all of the variables (i.e. that it can change all of the variables values such that the LTL formula is satisfied). We usually do *not* want this to be the case!

In the button example from Section 2.1.3, `red` is controlled by the application but `clicked` should be a user input. If we let the application control both, it could simply set `clicked` to be always false and `red` to be always true, which technically satisfies our spec that the button stays red until clicked (if it is never clicked, it must always stay red).

Realizability captures this notion of input versus output variables. Output variables are controlled by the system and therefore it need only find a satisfying assignment for the LTL formula with respect to them. Input variables; however, are not under the system's control and they must be universally satisfied.

Informally, we say that "the system is good if it fulfills the spec for all possible inputs."

Lecture 1 slides 17 and 18 provide an alternate formulation. When the system can control all variables, we can view this as verifying that a desired property holds for the system. When we designate some variables as inputs, it becomes a game. The system must be able to respond with a satisfying assignment of output variables for any assignments that an adversary can give to the input variables.

2.4 Synthesis as a game

Lecture 1 slides 22 through 30 depict a general approach to solving the Church Synthesis Problem and follow an example using an LTL spec.

The approach is as follows:

1. Provide a specification that your desired solution must conform to.
2. Construct a game where the adversary (called environment in the slides) controls the inputs and your system controls the outputs.
3. Find a winning strategy for the system. (No winning strategy means that the solution is unrealizable).
4. Construct a system using this winning strategy.

The slides depict an instantiation of this approach to an LTL formula as follows

1. The specification is an LTL formula.

2. The game involves finding an accepting path through a Büchi automaton (explained in brief in Section 2.4.1). The adversary and system control separate nodes and are allowed to pick any paths out of the nodes they control.

3. A winning strategy is one where the system passes through an accepting node infinitely many times no matter how the adversary plays.

4. This winning strategy can be directly translated to a system that produces the desired outputs (the choice of path corresponds to the values that the system needs to output given certain inputs).

The rest of this section will walk through the specifics of this example game - we will see how it generalizes to Reactive Synthesis in Section 2.5. Note that Reactive Synthesis itself is but an instantiation of this "game-based" approach to program synthesis, which could also be used to apply to specifications in, for example, propositional logic.

2.4.1 Büchi automata

Hopefully you know what a deterministic finite automaton is - if not, this explanation may not be very helpful.

To follow the synthesis approach, we need something we can translate an LTL formula to so that we can play a game on it. However, in LTL we deal with infinite sequences. And a finite automaton only recognizes finite words like *aaaa* or *aaabbb*. Consider the LTL formula $G\ a$, which states that a is always true. Only the infinite sequence $aaaa\ldots$ would satisfy $G\ a$, so we need a different kind of automaton.

A Büchi automaton is similar to a finite automaton, except it accepts infinite words instead of finite words. In this case we will consider deterministic Büchi automata but nondeterministic Büchi automata are defined analogously to nondeterministic finite automata.

There is one primary difference between a DFA and a deterministic Büchi automaton. A Büchi automaton can only accept if any of its accepting states are visited infinitely many times. This is what makes it accept infinite words - which is what we want because satisfying assignments in LTL are infinite.

There is a second difference between the Büchi automata we will consider and DFAs as you've seen before. This is that to satisfy an LTL formula we need to specify the state of all variables at each time step. For example, $G\ (a \Rightarrow b)$ could have a satisfying assignment that begins

Time	t_0	t_1	t_2	$\ldots$
a	0	0	1	$\ldots$
b	1	0	1	$\ldots$

However usually DFAs have transitions that look like a or b, which cannot appropriately represent these assignments. To solve this we will let our alphabet be the powerset of all variables in our formula. So the alphabet for the Büchi automaton representing the formula $G\ (a \Rightarrow b)$ would be $\{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. If a variable is included in the set, it is considered true, else false. So we could represent the beginning of this satisfying assignment as being the word $\{b\}, \emptyset, \{a, b\}, \ldots$ (commas added for clarity).

This means that there are exponentially many transitions in our automaton with respect to the number of variables but usually a lot of them are to the same state. A convention is to let a logical formula represent all elements in the alphabet for whose assignments the logical formula is true. For example, the formula $a \vee b$ would represent three transitions: one for each of $\{a\}, \{b\}$, and $\{a, b\}$. The formula $\neg a$ would represent two transitions: $\emptyset$ and $\{b\}$ (observe that the formula need not use all variables).

You can see an example Büchi automaton on lecture 1 slide 25.

Resources: Büchi automaton on Wikipedia.

2.4.2 The Büchi automaton game

Some (but as you will see later, not all) LTL formulas can be translated to deterministic Büchi automata as shown in lecture 1 slide 25.

We will make a game out of a Büchi automaton accepting and use this to synthesize our system. First we need to expand our states. The structure of our game requires that our inputs are chosen by the adversary first and then our system gets to respond with its outputs. We will encapsulate this by having states which are "adversary-controlled" whose outward transitions all belong only to settings of input variables and states which are "system-controlled" whose outwards transitions all belong to settings of output variables. On lecture 1 slide 27 the adversary (called environment on the slide) states are boxes and the system states are circles.

We always are able to modify our automaton in this fashion. The exercises below should help convince you of this.

Exercise: Look at the automaton on slide 27 and convince yourself that it is equivalent to the one on slide 25 (if we assume that the system controls all states in the former).

Exercise: can you explain informally how we can construct the former automaton from the latter? Hint: will you ever have more than one adversary state per state in the original Büchi automaton? Why?

Exercise: Why is the starting state the state labeled q_0 if the q_{00} and q_{0r} states also represent the q_0 state from the automaton on slide 25?

Onto the game itself.

The game we define is simple: we start in the starting state. The system and adversary take turns picking transitions (this is because by design, taking a transition from a system state will go to an adversary state and vice versa). If for any strategy the adversary employs, the system can pass through an accepting state infinitely many times, the system wins. Otherwise, the adversary wins.

Slides 29 and 30 in lecture 1 depict a winning strategy and the code generated from that winning strategy. Observe that the adversary inputs can be thought of as conditions in addition to the current state. When the state and adversary inputs are given, we can first find the state in our modified Büchi automaton and then take the transition representing the adversary input. The winning strategy will tell us which outputs to give.

2.4.3 Games formally

Lecture 2 slide 3 gives a formal definition of games and instantiations of them. The game that we defined before is the Büchi game defined on Lecture 2 slide 4.

The reachability game defined on the same slide is a relaxation of the rules of the Büchi game. In this game, the system only needs to eventually reach a state. A possible situation like this is one where the system controls a robot that needs to navigate to a goal and the inputs are its surroundings and sensor readings.

The way to solve this game is to find the "attractor" as noted in lecture 2 slide 5. The idea is to start with a set S and try to find all of the states from which the system can force a path to S. This grows recursively until it reaches a fixpoint.

If a circle state points to any state in S, we can add it to S because the system controls the circle states (it can just choose that path into S). If a square state only has transitions into S, we can add it to S because no matter what transition the adversary picks, it will end up in S.

In the example starting on slide 6 of lecture 2, the attractor is computed for state v_4 and v_5. If we imagine that these were our goal state and v_0 was the starting state, then we simply find the attractor of the set $\{v_4, v_5\}$ and check if v_0 is in it. In this case, it is not, so the adversary has a winning strategy (which is keeping the system trapped on the top in the v_1 state).

2.5 Reactive synthesis algorithm

We're finally ready to talk about the proper reactive synthesis algorithm.

The Büchi game described in Section 2.4.2 was a valid way to do reactive synthesis, but it required a special case: that we could construct a deterministic Büchi automaton from an LTL formula. This is not generally true: lecture 2 slide 16 gives the counterexample formula $F\ G\ p$ that cannot be expressed by a deterministic automaton (unlike with finite automata, deterministic and nondeterministic Büchi automata are *not* equivalent!).

However the issue with nondeterministic automata is that our game becomes harder to play. Our game requires that we always have a winning strategy, but in nondeterministic automata it's ok to have some paths that lead to rejection if there is at least one path that is accepting. So without determinism we can't know whether we have a proper winning strategy. Lecture 2 slide 16 has a counterexample if this explanation is too hand-wavey.

So we have to adapt our synthesis algorithm to account for the fact that we cannot always get a deterministic Büchi automaton.

The general reactive synthesis algorithm from LTL formulas is described on lecture 2 slide 15 and summarized below.

You start with an LTL formula and each item can be reduced to the next.

1. LTL formula
2. Nondeterministic Büchi automaton
3. Deterministic parity automaton
4. Parity game

Finally, if (and only if) we win the parity game, we can synthesize a circuit.

2.5.1 Time complexity

Refer to lecture 2 slide 17: LTL synthesis in this manner is 2EXPTIME-complete (i.e. $O(2^{(2^{p}(n))})$ where p is a polynomial). This means it is pretty expensive.

2.5.2 Issues with Safra's construction

Safra's construction is what is used to convert a nondeterministic Büchi automaton into a deterministic parity automaton. However, it contributes to the doubly-exponential runtime and is difficult to implement. Several works (referred to on slides 20 and 21 of lecture 2) seek to skip directly from an LTL formula to a deterministic parity automaton which is known colloquially as being "Safraless."

References: Safra construction paper

2.6 Additional references

1. Recent challenges and ideas in temporal synthesis. A nice reference on the problems mentioned/overview with Reactive Synthesis.
2. The Strix paper referred to in lecture.

3 Deductive Synthesis

3.1 What is deductive synthesis

A deductive approach to program synthesis is presented for the construction of recursive programs. This approach regards program synthesis as a theorem-proving task and relies on a theorem-proving method that combines the features of transformation rules, unification, and mathematical induction within a single framework [7].

3.2 What is an SMT solver?

SMT solvers are tools for proving formulas.

- Reference
- A core engine in:
  - Program analysis
  - Software engineering
  - Program model checking
  - Hardware verification
- Combine propositional satisfiability search techniques with specialized theory solvers
  - Linear arithmetic
  - Bit vectors
  - Uninterpreted functions with equality
- Examples of SMT solver: z3, CVC4, OpenSMT, Barcelogic

For more information about these kinds of solvers, see the MiniSat paper.

3.3 Complete functional synthesis

Definition (Synthesis Procedure)**.** *A synthesis procedure takes as input formula* $F(x, a)$ *and output (Note:* $pre(a)$ *is the "best" possible):*

1. *a precondition formula* $pre(a)$
2. *list of terms* ψ

such that the following holds:
$\exists x.F(x, a) \Leftrightarrow pre(a) \Leftrightarrow F[x := \psi]$

3.4 Deductive synthesis overview

- process every equality: take an equality E_i, compute a parametric description of the solution set and insert those values in the rest of formula.
- at the end there are only inequalities

4 Constraint Based Synthesis

4.1 Program Sketching

Given a program sketch with a hole ??, find a way to fill the hole such that the specifications are satisfied. The intuition for this kind of program synthesis is that the programmer is able to provide the high-level intuition about program structure and can leave the small details for the synthesizer to fill in.

Definition (Sketch-based Synthesis)**.** *Find c such that* $\forall x.(x * c) = x + x$ *which is equivalent to* $\exists c, \forall x.(x * c) = x + x$*.*

Generalized: $\exists c_1, c_2, \ldots, \forall x_1, x_2, \phi(c_1, c_2, \ldots, x_1, x_2, \ldots)$*.*

This form of synthesis has also led to work on hole-based programming which leverages similar techniques to have to programmer work in tandem with the compiler/editor to construct programs in an interactive fashion. To learn more, a great start is Armando Solar Lezama's thesis[10].

4.2 Constraint-based Synthesis

The goal of Constraint-based Synthesis is to take a given synthesis problem and encode it into the specification language of an automated solver. General purpose SAT/SMT solvers with years of optimizations and decision procedures can then solve the problem more efficiently than a standalone synthesizer and the solution can be extracted out from the model returned by said synthesizers.(Here a model is the mapping of variables in the provided formula to their satisfying values if a solution exists)

1. Convert synthesis problem P to formula ϕ.
2. Solve ϕ with an smt solver.
3. Map the model back to the solution space as a program that solves P.

4.3 Syntax Guided Program synthesis(Sygus)

Sygus speaks to a broad domain of programs where in addition to being given a specification constraint the synthesizer is also provided an input grammar which defines the language with which the program should be synthesized.

4.3.1 Programming By Example(PBE)

One area of Sygus is Programming By Example where the specification language describes a series of program inputs and their corresponding outputs. While the classic example is bit-vector transformations, this kind of synthesis is applicable to synthesizing general programs, spread sheet and database operations, and operations for editing videos and audio.

Broadly speaking, PDE is any problem with the following kind of specification: $[i_1 \rightarrow o_1, \ldots]$.

The flagship success of PBE is FlashFill[4] which is the program synthesis tool behind Excel spreadsheets which allow for automatic completion of spreadsheet cells by creating excel functions(regular expressions) from user provided examples.

Another example of this is Scythe[13] which applies similar techniques for generating SQL queries from example database manipulations.

4.4 Specifications from Programs

Specification do not just need to come in the form of logical constraints. The goal of program optimization is to take a given program a find the most optimal version of it for a given cost functions. This can be directly solved as a synthesis task.

A straight forward application of this is with super-optimizing compilers(synthesizer) which take in a possibly naive implementation of a function and treat it like an oracle for the expected behavior. The synthesizer can then proceed to do a more expansive search than the more local edits a traditional compiler might do to find a lower cost program that is semantically equivalent.

A similar usage of this is in Automated feedback generation for introductory programming assignments[9] which given a reference solution is able to compare student solutions and find the minimum number of edits. This can then be used to automatically provide feedback to hundreds of students in an efficient manner.

Finally, programs as specifications can be used to help generate logical invariants on the programs themselves. This often takes the form of coming up with Hoare Logic specifications for each individual step of the program to be able to semantically map the program to its behavior. In general, the problem of finding correct invariants on any looping code is undecidable in practice and difficult for the programmer to provide directly. Here we can borrow ideas from sketching and have the programmer provide a template of the invariant that should be synthesized at each looping point. The synthesizer can then go through and deduce valid specification for each of those points given the full context of the program and any post-condition that the program should satisfy.

4.5 High-level Idea

Program Synthesis can be used to solve a variety of interesting problems.

classical program synthesis(see slides for a nice diagram):

- Specification “what”
- Synthesizer
- Implementation “how”

4.6 Solving Sygus

While it may appear that these Sygus problems can just be encoded into smt-lib and shipped off to your preferred synthesizer to solve, in practice, the program

space is too big and unconstrained for this to work for most interesting cases because the constraints get too complicated/outside of the decidable logic of the solver. In these cases, the goal of the synthesizer is to break the synthesis problem down into bite size pieces that are easier process and iteratively arrive at the solution.

One method of doing this is active learning(See lecture 3 slide 29 for a helpful diagram). In active learning, a search algorithm/learner proposes candidate expressions and a learning oracle/teacher either verifies the result or gives the learner a counter examples. This method described is embodied in program synthesis as Counter-example guided inductive synthesis(Cegis[11]). This has turned out to be a very powerful technique and has led to much research in both a variety of search/learning strategies and leveraging this technique to solve synthesis tasks.

As shown in the slides. Initially, the learner knows almost nothing about the problem space and what a good solution looks like. Thus initially, they propose an obviously bad solution. This gets passed off to the teacher who identifies why the program can satisfy the target specification and provides a counter example. With each counter example, the learn better understands the constraints of the problem, can rule out similarly bad programs early, and with each iteration better search the space towards the solution.

The most basic form of this is known as enumerative search where the learner exhaustively tries all of the programs in the space that satisfy the counterexamples it's been given until the solution is found. From this there are many techniques to help prune the space to rule out whole sets or classes of programs from being tried. This is necessary for most tasks because the search space is almost always exponential proportional to the number of components.

4.7 The success of Sygus

Despite the inherent NP-Completeness of the problem, SMT-solvers have become widely successful and have seen significant industry adoption. With this rise has also propelled Sygus as a well-known problem/benchmark for the field even spinning off into their own competitions.

5 Synthesis for Modular Verification

One of the problems that deductive synthesis runs into is that writing good specification is hard. It is very easy to write specifications that are too broad to the point of being incorrect, or two narrow such it misses key behavior needed for verification. Specifying these specifications as contracts between the human and the computer is hard for general logic but can be alleviated with a common tool we've learned a lot about in OPLSS... type systems!

The work of liquid types[12] looks at a subset of dependent types which are in a decidable logic of SMT solvers and leverages them to describe interesting properties of programs. Here, the type signature describes the pre and

post conditions of the function by annotating each of the types with additional predicates. Each predicate describes a proposition about it's argument or result using any other variables in scope(constants or arguments to the left of it in the signature). These specifications are then translated along with the program body into a series of verification conditions which are solvable with SMT solvers.

While this tells you whether the program can be verified with respect to the user provided type signatures, it is still a challenge to come up with the right predicates. This is where G2[5] comes in. During a failed verification attempt, G2 can come up with counter examples for why a specification on one of the helper libraries is not sufficient to verify the program. This counter example can then be fed into a synthesizer which augments the user specification to satisfy both that specification and the counter example. This loop can then continue in the CEGIS style until all of the specification are strong enough such that the verifier can verify the program with respect to the type signatures of the program. Note here that the only specification that must remain unchanged is the top-level signature as this describes the intent of the user. It is the specifications of the helper/library functions which are augmented by the synthesizer in each iteration. It is also possible that synthesis ends with a concrete counterexample to the users code such that it is impossible to verify the program(indicating a bug in the users program).

For further reading, another take on this space is Synquid[8] which leverages refinement types to fully synthesize a Haskell program in a deductive synthesis style.

6 New Applications

Autonomous driving is a real world example of Reactive Synthesis. It features:

1. A hierarchical system which issues commands to the vehicle
2. Satisfies specification(speed limits, stays on road, knows when it is safe to reverse, etc.)

Boolean circuits are not up to this task: a car is controlled using 20+ sensors, and this far exceeds what traditional reactive synthesis can support.

A neat answer came from the lecturer's group: temporal stream logic (TSL) [3]. Direct link: Temporal Stream Logic: Synthesis Beyond the Bools.

It contains:

1. All the operators we know from LTL: global, next, until, eventually.
2. The use of a new abstraction, *signals*. These are "streams" of user-defined atomic data. The key observation is that not all domains of interest can be modeled efficiently using booleans as a primitive. These new user-defined signals are a little more sophisticated than booleans.

Our temporal operators are consequently more sophisticated as well: we have new "update" terms that allow the user to apply the result of a function as an output signal. We also have predicates applied on function terms.

The basic game is going to be to express TSL queries as LTL queries. If the LTL version is realizable, so is the TSL. The LTL formula will have two things:

1. A syntactic translation of the TSL formula
2. Additional formulae that specify the uniqueness of updates.

The latter in particular may be tricky to arrive at in one go; it is likely that the user will need to go back and forth with the LTL translation to see when it is unrealizable and what, if anything, can be done to aid it.

A neat example is a music player app.

The pro is obvious: that we can model this at all! Given a spec, we're going to synthesize an Android app that will run on the Android OS and deal with user input. It clearly needs more than booleans. It needs to do some nifty things, e.g., how to respond to weird combinations of play/pause/reboot, how to use external APIs, etc. We are able to make small changes in the spec and gain (potentially) large changes in the generated code.

The con is that this method is, in general, undecidable. This is via a reduction from the Post correspondence problem. The Post correspondence problem, also known as the domino problem, is a decision problem that is known to be undecidable. Wikipedia article.

Zooming out to look at functional reactive programming (FRP) more generally, we know there is a link between FRP and LTL via the Curry-Howard correspondence [6]. Some of this comes together in another of the lecturer's papers [2]. Direct link: Synthesizing Functional Reactive Programs

Another of the lecturer's projects explores the possible union of reactive synthesis and syntax-guided synthesis [1]. Direct link: Can reactive synthesis and Sygus be friends?

One of the contributions is to endow TSL with *theories*. This allows you to get over a bump that plain TSL would have gotten stuck on. Functions need not be treated opaquely; we can look into them. Constants need not be black boxes; we can investigate them.

In general, syntax-guided synthesis was designed to tackle data transformation problems, while reactive synthesis is good for control-flow problems. More real-world programs have elements of both data and control, which is why a union of these two approaches is of such interest. The paper finds that TSL, modulo theories, is a great language to specify this union of strategies!

References

[1] Wonhyuk Choi. Can reactive synthesis and syntax-guided synthesis be friends? In *Companion Proceedings of the 2021 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications:*

Software for Humanity, SPLASH Companion 2021, page 3–5, New York, NY, USA, 2021. Association for Computing Machinery.

[2] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. Synthesizing functional reactive programs. In Richard A. Eisenberg, editor, *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, pages 162–175. ACM, 2019.

[3] Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. Temporal stream logic: Synthesis beyond the bools. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 609–629, Cham, 2019. Springer International Publishing.

[4] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.*, 46(1):317–330, jan 2011.

[5] William T. Hallahan, Anton Xue, Maxwell Troy Bland, Ranjit Jhala, and Ruzica Piskac. Lazy counterfactual symbolic execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 411–424, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Alan Jeffrey. Ltl types frp: Linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification*, PLPV '12, page 49–60, New York, NY, USA, 2012. Association for Computing Machinery.

[7] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, jan 1980.

[8] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 522–538, New York, NY, USA, 2016. Association for Computing Machinery.

[9] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26, 2013.

[10] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[11] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support*

for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006, pages 404–415, 2006.

[12] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. *SIGPLAN Not.*, 49(9):269–282, aug 2014.

[13] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. *SIGPLAN Not.*, 52(6):452–466, jun 2017.