PROGRAM SYNTHESIS
REACTIVE SYNTHESIS – “HOLY GRAIL” (WELL, ONE OF THEM)

- Autonomous driving
  - Reactive traffic planner decides whether vehicle should stay in the travel lane or perform a passing maneuver, whether it should go or stop, whether it is allowed to reverse, etc.
  - Hierarchical control: reactive traffic planner interacts with mission control (above) and path planner (below).
- Specification consists of
  - Traffic rules (for example “no collision”, “obey speed limits”, ...)
  - Goals (for example “eventually the checkpoint should be reached”)
- More in the following survey paper [Murray et al, 2012]
REACTIVE SYNTHESIS

Is a Boolean circuit the right representation for these systems?

Idea: Temporal Stream Logic
Abstract away from Boolean circuits
Applications are still limited to small examples

- Synthesis from LTL specifications is 2EXPTIME hard
- Synthesis of distributed systems (where the processes have incomplete information) is in general undecidable

We tried to synthesize a simple autonomous driving controller [SCAV2017] with current state of the art tools

The controller only needs to switch between a small number of behaviors, like steering during a bend, or shifting gears on high rpm

To detect those situations, the controller needs to process 20+ sensors of the car

This accumulation of sensors values exceeded the capabilities of the tools
New logic: TSL (temporal stream logic), defined over streams of data, with user defined/API predicates and function calls

New synthesis “procedures” – extending the existing work on reactive synthesis to this new logic, outputting executable FRP programs

New applications: among others we synthesized a controller for a simulator for autonomous vehicles, a music player, …
□(\text{pressedEvent click} \leftrightarrow \text{[count} \leftrightarrow \text{increment count]]) \land \text{[screen} \leftrightarrow \text{display count]})
TSL EXAMPLE

yampaButton =
  proc click -> do
    rec
      count' <- hold 0 <- count
      pic <- arr display <- count'
      count <- if pressedEvent click
        then arr increment <- count'
        else arr id <- count'
    returnA <- pic

pressedEvent = ...
increment = ...
display = ...
TSL EXAMPLE

```
yampaButton =
  proc click -> do
    rec
      count' <- hold 0 <- count
      pic <- arr display <- count'
      count <- if pressedEvent click
      then arr increment <- count'
      else arr id <- count'
    returnA <- pic

  pressedEvent = ...
  increment = ...
  display = ...
```

\( (\text{pressedEvent click} \leftrightarrow [\text{count} \leftrightarrow \text{increment count}]) \)

\& \( [\text{screen} \leftrightarrow \text{display count}] \)
TEMPORAL STREAM LOGIC (TSL)

- All temporal operators are the same as in LTL
- Input variables are not Booleans but *signals*
- Temporal operators are defined on atoms which can either be an update atom, or a predicate applied on function terms
SYNTHESIZING A MUSIC PLAYER APP

- Android Lifecycle

```javascript
Sys.leaveApp() {
    if (MP.musicPlaying())
        Ctrl.pause();
}
Sys.resumeApp() {
    pos = MP.trackPos();
    Ctrl.play(Tr, pos);
}
```

Finding resume and restart errors in android applications Shan, Z., Azim, T., Neamtiu, I OOPSLA 2016

Available online: GitHub, Google Store
SYNTHESIZING A MUSIC PLAYER APP

- Android Lifecycle

```javascript
Sys.leaveApp() {
    if (MP.musicPlaying())
        Ctrl.pause();
}

Sys.resumeApp() {
    pos = MP.trackPos();
    Ctrl.play(Tr,pos);
}
```

Input “variables” for specification:

- The Android system (Sys)
- The Android music player library (MP)
- Its control interface (Ctrl)
- The currently selected track (Tr)
- API functions and routines
SYNTHEIZING A MUSIC PLAYER APP

• Android Lifecycle

Sys.leaveApp() {
    if (MP.musicPlaying())
        Ctrl.pause();
}

Sys.resumeApp() {
    pos = MP.trackPos();
    Ctrl.play(Tr, pos);
}

ALWAYS (leaveApp(Sys) ∧ musicPlaying(MP)
         ⇒ [Ctrl ↩ pause()])

ALWAYS (resumeApp(Sys)
         ⇒ [Ctrl ← play(Tr, trackPos(MP))])
SYNTHESIZING A MUSIC PLAYER APP

- Android Lifecycle

```javascript
Sys.leaveApp() {
  if (MP.musicPlaying())
    Ctrl.pause();
}
Sys.resumeApp() {
  pos = MP.trackPos();
  Ctrl.play(Tr, pos);
}
```

**New task:**

On resume app, only play music if the music was already playing when paused.

```
ALWAYS (leaveApp(Sys) ∧ musicPlaying(MP)
   ⇒ [Ctrl ↔ pause()])
```
SYNTHESIZING A MUSIC PLAYER APP

• Android Lifecycle
  
  ```
  bool wasPlaying = false;
  
  Sys.leaveApp() {
    if (MP.musicPlaying()){
      wasPlaying = true;
      Ctrl.pause();
    } else {
      wasPlaying = false;
    }
  }
  
  Sys.resumeApp() {
    if (wasPlaying) {
      pos = MP.trackPos();
      Ctrl.play(Tr,pos);
    }
  }
  ```

  ALWAYS (leaveApp(Sys) ∧ musicPlaying(MP)
  ⇒ [Ctrl ← pause()])

  ALWAYS (leaveApp(Sys) ∧ musicPlaying(MP)
  ⇒ [Ctrl ← play(Tr,trackPos(MP)]
  AS_SOON_AS resumeApp(Sys))

  AS_SOON_AS:
  \( \varphi \land \psi \equiv \neg \psi \ W(\psi \land \varphi) \)
FUNCTION ABSTRACTION

always (leaveApp(Sys) ∧ musicPlaying(MP) ⇒ [Ctrl ↯ pause()])

always (leaveApp(Sys) ∧ musicPlaying(MP) ⇒ [Ctrl ↯ play(Tr, trackPos(MP)]
  as_soon_as resumeApp(Sys)])

leaveApp(Sys) { ... }
musicPlaying(MP) { ... }
play(Tr, trackPos(MP)) { ... }
resumeApp(Sys) { ... }
SYNTHESIS FROM TSL SPECIFICATIONS
LTL SYNTHESIS

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

Player 0 wins if the LTL formula is realizable.
Player 1 wins if the LTL formula is unrealizable.
OVERVIEW OF THE SYNTHESIS PROCEDURE
TSL SYNTHESIS PROCEDURE

- **Theorem 1**: TSL synthesis problem is undecidable (reducing the Post correspondence problem to a TSL synthesis problem)
TSL SYNTHESIS PROCEDURE

- **Theorem 1**: TSL synthesis problem is undecidable (a proof by reducing the Post correspondence problem to a TSL synthesis problem)
- **Theorem 2**: If the abstracted TSL formula is realizable (in LTL), then is the original formula also realizable
- An LTL synthesis tool constructs a control flow, which means that this flow holds for any given implementation of predicates and functions
TSL SYNTHESIS PROCEDURE – EXAMPLE 0.1

TSL specification

\[ F \ p(x) \Rightarrow FG \ p(y) \]

\( x \) – input, \( y \) – output signals

LTL specification

\[ F \ p_x \Rightarrow FG \ p_y \]

\( p_x, p_y \) – inputs

This LTL specification is unrealizable: the system simply set \( p_x \) to be always true, and \( p_y \) – to be always false.
TSL SYNTHESIS PROCEDURE – EXAMPLE 1.1

TSL
\[ F p(x) \Rightarrow FG p(y) \land F [y \leftarrow y] \]
x – input, y – output

LTL specification
\[ G \left( (y_y \land \neg y_x) \lor (\neg y_y \land y_x) \right) \land \]
\[ F p_x \Rightarrow FG p_y \land F y_y \]
p_x, p_y – inputs

The top line specifies that y can be updated with only one value.
TSL TO LTL ABSTRACTION

- Given a TSL formula, the abstracted LTL formula will be a conjunction of
  - Syntactic conversion from the TSL formula
  - Globally quantified formulas describing the uniqueness of the updates

- This abstraction might need infinitely many terms, if there are functions in the specification
- There are specifications demonstrating that observation
- In practice: lazy instantiation and CEGAR loop
FROM STRATEGIES TO SPECIFICATION REFINEMENT

TSL specification

\[ F \ p(x) \Rightarrow FG \ p(y) \]

\(x\) – input, \(y\) – output signals

TSL specification refinement

\[ F \ p(x) \land \]
\[ G([y \leftarrow x] \land p(x) \Rightarrow Xp(y)) \land \]
\[ G([y \leftarrow x] \land \neg p(x) \Rightarrow X\neg p(y)) \land \]
\[ G([y \leftarrow y] \land p(y) \Rightarrow Xp(y)) \land \]
\[ G([y \leftarrow y] \land \neg p(y) \Rightarrow X\neg p(y)) \]
\[ \Rightarrow FG \ p(y) \]

\(x\) – input, \(y\) – output signals

This new specification is strong enough to be realizable in LTL, when abstracted
MUSIC PLAYER SYNTHESIS

Diagram showing the flow of inputs and outputs for a music player, including buttons like play, pause, resume, and leave, along with symbols for configuration (Cfg) and music playback (MP).
MUSIC PLAYER SYNTHESIS

- **Refinement**
- **Counter Strategy**
- **spurious**
- **non-spurious**
- **unrealizable**

- **TSL**
- **CFM**

- **Synthesis**
  - **LTL**
  - **Circuit**

- **FRP Translator**

- **Project Context**

- **Compiler**

- **EXE**

- **LTL Synthesis Tool**

- **Design Pattern:**
  - Arrow
  - Applicative

- **FRP Library**

- **Function & Predicate Implementations**
Abstracting from data transformations allows synthesis to scale to new application domains.

We trade theoretical complexity for practical scalability.

Temporal Stream Logic - Synthesis Beyond the Bools. CAV 2019. Finkbeiner, Klein, Piskac, Santolucito
Synthesizing Functional Reactive Programs. Haskell 2019. Finkbeiner, Klein, Piskac, Santolucito
REACTIVE SYSTEMS

Synthesized a self-driving car controller in < 4 seconds

Temporal Stream Logic - Synthesis Beyond the Bools. CAV 2019. Finkbeiner, Klein, Piskac, Santolucito
Synthesizing Functional Reactive Programs. Haskell 2019. Finkbeiner, Klein, Piskac, Santolucito
REACTIVE SYSTEMS

Synthesized a self-driving car controller in < 4 seconds

Cyber-physical

Embedded Devices

Mobile Applications

Graphical User Interfaces

Temporal Stream Logic - Synthesis Beyond the Bools. CAV 2019. Finkbeiner, Klein, Piskac, Santolucito
Synthesizing Functional Reactive Programs. Haskell 2019. Finkbeiner, Klein, Piskac, Santolucito
REACTIVE SYSTEMS

Synthesized a self-driving car controller in < 4 seconds

Cyber-physical

Embedded
Devices

Mobile
Applications

Graphical User
Interfaces

Temporal Stream Logic - Synthesis Beyond the Bools. CAV 2019. Finkbeiner, Klein, Piskac, Santolucito

Synthesizing Functional Reactive Programs. Haskell 2019. Finkbeiner, Klein, Piskac, Santolucito
Synthesized a self-driving car controller in < 4 seconds
Synthroids: Synthesizing a Game for FPGAs using Temporal Logic Specifications. Geier, Heim, Klein, Finkbeiner: FMCAD 2019

https://github.com/reactive-systems/Synthroids
LIVE DEMOS

(Dylan Iskandar, Raven Rothkopf, Leyi Cui)

https://monkeyarya.github.io/moveCube/
(Arya Sinha)

https://barnard-pl-labs.github.io/tsl-api/
(Rhea Kothari, Danielle Cai, Nupur Dave)

https://stately.ai/viz/5fadaf7f-90ff-48cd-b36a-9a45dd5246a8
(Shmuel Berman)
always assume {

(! (room.heating.off <-> room.heating.on)) ;
([ room.heating.ctrl <- turnOn() ]
 -> F ([ room.heating.ctrl <- turnOff() ] R room.heating.on)) ;
([ room.heating.ctrl <- turnOff() ]
 -> F ([ room.heating.ctrl <- turnOn() ] R room.heating.off));
([ room.heating.ctrl <- turnOff() ]
 -> F (! (gt outside.temperature room.temperature)))
}
always guarantee {

gt outside.temperature room.temperature
 -> F room.heating.off
BEYOND UNINTERPRETED FUNCTIONS

\( \square (\lbrack y \leftarrow y \rbrack \lor \lbrack y \leftarrow x \rbrack) \land \lozenge p \ x \rightarrow \lozenge p \ y \)

TSL spec

\( \square (x_{\text{to} \ y} \rightarrow (p_{\ x} \leftrightarrow \lozenge p_{\ y})) \)

\( \square \neg(y_{\text{to} \ y} \land x_{\text{to} \ y}) \land \square (y_{\text{to} \ y} \lor x_{\text{to} \ y}) \land \lozenge p_{\ x} \rightarrow \lozenge p_{\ y} \)

Refined Approximation

Can reactive synthesis and syntax-guided synthesis be friends?
Choi, Finkbeiner, Piskac, Santolucito: PLDI 2022
The refinement is a partial encoding of the semantics of uninterpreted functions.

Can we use the same strategy for other theories?

Can reactive synthesis and syntax-guided synthesis be friends? Choi, Finkbeiner, Piskac, Santolucito: PLDI 2022
SYNTAX-GUIDED SYNTHESIS (SYGUS)

Semantic Constraint

<table>
<thead>
<tr>
<th>Input $v_1$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dr. Eran Yahav</td>
<td>Yahav, E.</td>
</tr>
<tr>
<td>Prof. Kathleen S. Fisher</td>
<td>Fisher, K.</td>
</tr>
<tr>
<td>Bill Gates, Sr.</td>
<td>Gates, B.</td>
</tr>
<tr>
<td>George Ciprian Necula</td>
<td>Necula, G.</td>
</tr>
<tr>
<td>Ken McMillan, II</td>
<td>McMillan, K.</td>
</tr>
</tbody>
</table>

Syntactic Constraint

- String expr $P$ := $\text{Switch}(b_1, c_1, \cdots, (b_n, c_n))$
- Bool b := $d_1 \lor \cdots \lor d_n$
- Conjunction d := $\pi_1 \land \cdots \land \pi_n$
- Predicate $\pi$ := $\text{Match}(v_i, r, k) \lor \neg \text{Match}(v_i, r, k)$
- Trace expr e := $\text{Concatenate}(f_1, \cdots, f_n)$
- Atomic expr f := $\text{SubStr}(v_i, p_1, p_2)$
  | ConstStr(s)
  | Loop($A \mid c$)
- Position p := $\text{CPos}(k) \mid \text{Pos}(r_1, r_2, c)$
- Integer expr c := $k \mid k_1 + k_2$
- Regular Expression r := $\text{TokenSeq}(T_1, \cdots, T_m)$
- Token T := $C+ \mid \neg C+ \mid \text{SpecialToken}$

Syntax-Guided Synthesis

Great for data transformation problems!
REACTIVE SYNTHESIS

Temporal Logic Specification

**Guarantee 3.** When a length-four locked burst starts, no other accesses to HREADY is high, so the current burst ends at the fourth occurrence of HREADY.

\[
\square((\text{HMASTLOCK} \land \text{HBURST} = \text{BURST4} \land \text{START} \land \text{HREADY}) \rightarrow \\
\circ(\neg\text{START} \lor \lbrack 3 \rbrack (\neg\text{START} \land \text{HREADY}))).
\]

\[
\square((\text{HMASTLOCK} \land \text{HBURST} = \text{BURST4} \land \text{START} \land \neg\text{HREADY}) \rightarrow \\
\circ(\neg\text{START} \lor \lbrack 4 \rbrack (\neg\text{START} \land \text{HREADY}))).
\]

**Guarantee 6.** If we do not start an access in the next time step, the bus will notgrant.

For each master \( i \),

\[
\square(\neg\text{START}) \rightarrow ((\text{HMASTER} = i \leftrightarrow \circ(\text{HMASTER} = i)) \land \\
(\text{HMASTLOCK} \leftrightarrow \circ(\text{HMASTLOCK}))).
\]

**Assumption 4.** We assume that all input signals are low initially.

\[
\bigwedge_{i} \left((\neg\text{HBUSREQ}[i] \land \neg\text{HLOCK}[i]) \land \neg\text{HREADY}\right).
\]

Synthesized Model

Great for control-flow problems!
SYNTAX-GUIDED SYNTHESIS (SYGUS)

Good for data transformation problems

REACTIVE SYNTHESIS

Good for control-flow problems

But there’s a catch…
SYNTAX-GUIDED SYNTHESIS (SYGUS)

Good for data transformation problems

Not designed for control flow

REACTIVE SYNTHESIS

Good for control-flow problems

Not designed for data transformations

But even trivial programs have both data and control.
WHAT DOES IT MEAN TO HAVE BOTH **DATA** AND **CONTROL**?

**Linux Completely Fair Scheduler**

- Runs the task “that has run for the least amount of time”

- “Time” is weighted
  - 1 μs of prioritized task → 0.25 μs
  - 1 μs of a low-priority task → 5 μs

- **Control:**
  Enqueuing and dequeuing tasks

- **Data:**
  Calculate how long each process has run
**EVEN VAST SIMPLIFICATIONS CAN STILL HAVE DATA AND CONTROL**

- Scheduler with two tasks
- Task 1 must run at least twice

- **States:**
  - Run task 1
  - Run task 2

- **Data transformations:**
  - Count number of executions

- Can we synthesize this?
WE HAVE A LANGUAGE TO SPECIFY IT...

- Scheduler with two tasks
- Task 1 must run at least twice

States:
- Run task 1
- Run task 2

Data transformations:
- Count number of executions

Temporal Stream Logic Modulo Theories (TSL-MT)

\[
\Box (\text{task} \leftrightarrow \text{task}1) \lor \text{[task} \leftrightarrow \text{task}2) \\
\land \text{[task} \leftrightarrow \text{task}1) \leftrightarrow \text{[taskTime}1 \leftrightarrow \text{add taskTime}1 \ 1] \\
\land \text{[task} \leftrightarrow \text{task}2) \leftrightarrow \text{[taskTime}2 \leftrightarrow \text{add taskTime}2 \ 1] \\
\land \text{eq taskTime}1 \ 0 \rightarrow \Diamond \text{(eq taskTime}1 \ 2)
\]
WE HAVE A LANGUAGE TO SPECIFY IT...

- Scheduler with two tasks
- Task 1 must run at least twice

Temporal Stream Logic Modulo Theories (TSL-MT)

\[ □([\text{task} ← \text{task1}] \lor [\text{task} ← \text{task2}]) \]
\[ \land [\text{task} ← \text{task1}] \leftrightarrow [\text{taskTime1} ← \text{add taskTime1} 1] \]
\[ \land [\text{task} ← \text{task2}] \leftrightarrow [\text{taskTime2} ← \text{add taskTime2} 1] \]
\[ \land \text{eq taskTime1} 0 \rightarrow ◇(\text{eq taskTime1} 2) \]

- Control

- Data transformations:
  - Count number of executions

FoSSaCS ’22 Finkbeiner et. al
FIRST, ADD THEORIES TO TSL TO GET TSL-MT...

- Scheduler with two tasks
- Task 1 must run at least twice

- States:
  - Run task 1
  - Run task 2

- Data transformations:
  - Count number of executions

Temporal Stream Logic Modulo Theories (TSL-MT)

\[
\begin{align*}
\Box & (\text{task} \leftarrow \text{task1}) \lor \Box (\text{task} \leftarrow \text{task2}) \\
\land & (\text{task} \leftarrow \text{task1}) \leftrightarrow (\text{taskTime1} \leftarrow \text{add taskTime1} 1) \\
\land & (\text{task} \leftarrow \text{task2}) \leftrightarrow (\text{taskTime2} \leftarrow \text{add taskTime2} 1) \\
\land & \text{eq taskTime1} 0 \rightarrow \Diamond (\text{eq taskTime1} 2)
\end{align*}
\]
THEN, WE NEED A SYNTHESIS PROCEDURE FOR TSL-MT...

- Reactive (TSL) Synthesis can synthesize “control”
- All functions are uninterpreted!
- SyGuS can synthesize “data”
- But it can’t generate state machines!

**Temporal Stream Logic Modulo Theories (TSL-MT)**

{\[
\square (\text{task} \leftarrow \text{task1}) \lor \square (\text{task} \leftarrow \text{task2}) \\
\land (\text{task} \leftarrow \text{task1}) \leftrightarrow (\text{taskTime1} \leftarrow \text{add taskTime1} 1) \\
\land (\text{task} \leftarrow \text{task2}) \leftrightarrow (\text{taskTime2} \leftarrow \text{add taskTime2} 1) \\
\land \text{eq taskTime1} 0 \rightarrow \Diamond (\text{eq taskTime1} 2)
\]}

**Control**

\[\square (\text{task} \leftarrow \text{task1}) \lor \square (\text{task} \leftarrow \text{task2}) \land (\text{task} \leftarrow \text{task1}) \leftrightarrow (\text{taskTime1} \leftarrow \text{add taskTime1} 1) \land (\text{task} \leftarrow \text{task2}) \leftrightarrow (\text{taskTime2} \leftarrow \text{add taskTime2} 1) \land \text{eq taskTime1} 0 \rightarrow \Diamond (\text{eq taskTime1} 2) \]

**Data**

\[\square (\text{task} \leftarrow \text{task1}) \lor \square (\text{task} \leftarrow \text{task2}) \land (\text{task} \leftarrow \text{task1}) \leftrightarrow (\text{taskTime1} \leftarrow \text{add taskTime1} 1) \land (\text{task} \leftarrow \text{task2}) \leftrightarrow (\text{taskTime2} \leftarrow \text{add taskTime2} 1) \land \text{eq taskTime1} 0 \rightarrow \Diamond (\text{eq taskTime1} 2) \]
Temporal Stream Logic Modulo Theories (TSL-MT)

\[
\begin{align*}
\square (\text{task} \rightarrow \text{task1}) & \lor \square (\text{task} \rightarrow \text{task2}) \\
\land (\text{task} \leftrightarrow \text{task1}) & \leftrightarrow (\text{taskTime1} \leftrightarrow \text{add taskTime1 1}) \\
\land (\text{task} \leftrightarrow \text{task2}) & \leftrightarrow (\text{taskTime2} \leftrightarrow \text{add taskTime2 1}) \\
\land \text{eq taskTime1 0} & \rightarrow (\text{eq taskTime1 2})
\end{align*}
\]
SYNTAX-GUIDED SYNTHESIS (SYGUS)

Good for data transformation problems
Does not have control flow

REACTIVE SYNTHESIS

Good for control-flow problems
Does not have data transformations
SYNTAX-GUIDED SYNTHESIS
(SYGUS)

Good for data transformation problems

Does not have control flow

REACTIVE SYNTHESIS

Good for control-flow problems

Does not have data transformations
REACTIVE SYNTHESIS AND SYNTAX-GUIDED SYNTHESIS CAN BE FRIENDS!

- **SyGuS** can “teach” *Reactive Synthesis* how to solve data transformation problems
- *Reactive Synthesis* can then handle control flow problems
- Use both to solve problems they excel at, then **communicate**!
- Can synthesize a simple linux scheduler...
- But also C code for the Linux Completely Fair Scheduler!
Synthesizing a ... Temporal Stream Logic Modulo Theories Specification

Control Flow Problem

Data Transformation Problem

Remove function & predicate interpretations

Derive and solve SyGuS problems

Temporal Stream Logic (TSL) Specification

SyGuS function

Combine

Transform SyGuS result to TSL assumption

TSL Specification With Assumptions

Reactive (TSL) Assumption

Reactive Synthesis

Executable Code
SYNTHESIZING THE TWO-TASK SCHEDULER

- Scheduler with two tasks
- Task 1 must run at least twice

Original Specification

\[
\Box ([\text{task} \leftarrow \text{task1}] \lor [\text{task} \leftarrow \text{task2}])
\land [\text{task} \leftarrow \text{task1}] \leftrightarrow [\text{taskTime1} \leftarrow \text{add taskTime1}] \\
\land [\text{task} \leftarrow \text{task2}] \leftrightarrow [\text{taskTime2} \leftarrow \text{add taskTime2}] \\
\land \text{eq taskTime1 0} \rightarrow \Diamond (\text{eq taskTime1 2})
\]
Synthesizing a ... Temporal Stream Logic Modulo Theories

Control Flow Problem

Data Transformation Problem

- Derive and solve SyGuS problems
- SyGuS function
- Transform SyGuS result to TSL assumption
- Reactive (TSL) Assumption
- TSL Specification With Assumptions
- Reactive Synthesis
- Executable Code

Temporal Stream Logic (TSL) Specification

Combine

Remove function & predicate interpretations
Synthesizing a ... Temporal Stream Logic Modulo Theories Specification

Remove function & predicate interpretations

Temporal Stream Logic (TSL) Specification

Combine

TSL Specification With Assumptions

Reactive Synthesis

SyGuS function

Transform SyGuS result to TSL assumption

Reactive (TSL) Assumption

Executable Code
REMOVING FUNCTION & PREDICATE INTERPRETATIONS FROM TSL-MT

- Temporal Stream Logic Modulo Theories to...
  Temporal Stream Logic

- Removes interpretations of `eq` and `add`: make it a pure control flow problem
- But it now doesn’t know that \(0 + 1 + 1 = 2\)
Synthesizing a Temporal Stream Logic Modulo Theories Specification

- Derive and solve SyGuS problems
- SyGuS function
- Transform SyGuS result to TSL assumption
- Reactive (TSL) Assumption

Remove function & predicate interpretations

Temporal Stream Logic (TSL) Specification

Combine

TSL Specification With Assumptions

Reactive Synthesis

Executable Code
Synthesizing a Temporal Stream Logic Modulo Theories Specification

- Control Flow Problem
- Data Transformation Problem

Derive and solve SyGuS problems

- SyGuS function
- Transform SyGuS result to TSL assumption

Remove function & predicate interpretations

Combine

TSL Specification With Assumptions

Reactive Synthesis

Executable Code
Synthesizing a Temporal Stream Logic Modulo Theories Specification

Remove function & predicate interpretations

\( (\text{task} \leftarrow \text{task1}) \lor (\text{task} \leftarrow \text{task2}) \\
\land (\text{task} \leftarrow \text{task1}) \leftrightarrow [\text{taskTime1} \leftarrow \text{add taskTime1 1}] \\
\land (\text{task} \leftarrow \text{task2}) \leftrightarrow [\text{taskTime2} \leftarrow \text{add taskTime2 1}] \\
\land \text{eq taskTime1 0} \rightarrow \lozenge (\text{eq taskTime1 2}) \)

Combine

TSL Specification With Assumptions

Reactive Synthesis

Executable Code

Derive and solve SyGuS problems

SyGuS function

Transform SyGuS result to TSL assumption

Reactive (TSL) Assumption
## SYNTAX-GUIDED SYNTHESIS (SYGUS)

### Semantic Constraint

<table>
<thead>
<tr>
<th>Input $v_1$</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dr. Eran Yahav</td>
<td>Yahav, E.</td>
</tr>
<tr>
<td>Prof. Kathleen S. Fisher</td>
<td>Fisher, K.</td>
</tr>
<tr>
<td>Bill Gates, Sr.</td>
<td>Gates, B.</td>
</tr>
<tr>
<td>George Ciprian Necula</td>
<td>Necula, G.</td>
</tr>
<tr>
<td>Ken McMillan, II</td>
<td>McMillan, K.</td>
</tr>
</tbody>
</table>

### Syntactic Constraint

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>String expr $P$</td>
<td>$\text{Switch}(b_1, c_1, \ldots, (b_n, c_n))$</td>
</tr>
<tr>
<td>Bool $b$</td>
<td>$d_1 \lor \cdots \lor d_n$</td>
</tr>
<tr>
<td>Conju nct $d$</td>
<td>$\pi_1 \land \cdots \land \pi_n$</td>
</tr>
<tr>
<td>Predicate $\pi$</td>
<td>$\text{Match}(v_i, r, k) \lor \neg \text{Match}(v_i, r, k)$</td>
</tr>
<tr>
<td>Trace expr $e$</td>
<td>$\text{Concatenate}(f_1, \ldots, f_n)$</td>
</tr>
<tr>
<td>Atomic expr $f$</td>
<td>$\text{SubStr}(v_i, p_1, p_2)$</td>
</tr>
<tr>
<td></td>
<td>$\text{ConstStr}(s)$</td>
</tr>
<tr>
<td></td>
<td>$\text{Loop}(\mathbf{aw} : c)$</td>
</tr>
<tr>
<td>Position $p$</td>
<td>$\text{CPos}(k) \lor \text{Pos}(r_1, r_2, c)$</td>
</tr>
<tr>
<td>Integer expr $c$</td>
<td>$k \mid k_1 + k_2$</td>
</tr>
<tr>
<td>Regular Expression $r$</td>
<td>$\text{TokenSeq}(T_1, \ldots, T_m)$</td>
</tr>
<tr>
<td>Token $T$</td>
<td>$C + \mid [-C] \mid {$</td>
</tr>
<tr>
<td></td>
<td>$\text{SpecialToken}$</td>
</tr>
</tbody>
</table>

### Syntax-Guided Synthesis

![Image of Syntax-Guided Synthesis](image.png)
DERIVE AND SOLVE SYGUS PROBLEMS

Original specification

Semantic Constraint

<table>
<thead>
<tr>
<th>Pre-condition</th>
<th>Program</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq taskTime1 0</td>
<td>$S$</td>
<td>eq taskTime1 2</td>
</tr>
</tbody>
</table>

Syntactic Constraint

$S ::= \text{add } S \ 1 \ | \ \text{taskTime1}$

SyGuS-synthesized function

$S = (\text{add} \ (\text{add} \ \text{taskTime1} \ 1) \ 1)$
Synthesizing a Temporal Stream Logic Modulo Theories Specification

- Control Flow Problem
- Data Transformation Problem

Derive and solve SyGuS problems

SyGuS function

Transform SyGuS result to TSL assumption

Reactive (TSL) Assumption

TSL Specification With Assumptions

Executable Code

Remove function & predicate interpretations

Combine

TSL Specification With Assumptions

Reactive Synthesis
Synthesizing a Temporal Stream Logic Modulo Theories Specification

Remove function & predicate interpretations

Combine

TSL Specification With Assumptions

Reactive Synthesis

Executable Code

Reactives (TSL) Assumption

\( S = (\text{add} (\text{add} \text{taskTime1} 1) 1) \)

Transform SyGuS result to TSL assumption
Synthesizing a ... Temporal Stream Logic Modulo Theories Specification

- Control Flow Problem
- Data Transformation Problem

Derive and solve SyGuS problems
- SyGuS function
- Reactive (TSL) Assumption

TSL Specification With Assumptions

Remove function & predicate interpretations

Combine

Executable Code

Reactive Synthesis

S = \((\text{add} \ (\text{add} \ \text{taskTime1} \ 1) \ 1)\)

Transform SyGuS result to TSL assumption

Reactive (TSL) Assumption
HOW TO COMMUNICATE SYGUS RESULT TO REACTIVE SYNTHESIS?

Original specification

\(\Diamond([\text{task} \leftarrow \text{task1}] \lor [\text{task} \leftarrow \text{task2}])\)
\(\land [\text{task} \leftarrow \text{task1}] \leftrightarrow [\text{taskTime1} \leftarrow \text{add taskTime1 1}]\)
\(\land [\text{task} \leftarrow \text{task2}] \leftrightarrow [\text{taskTime2} \leftarrow \text{add taskTime2 1}]\)
\(\land \text{eq taskTime1 0} \rightarrow \Box(\text{eq taskTime1 2})\)

SyGuS-synthesized function

\(S = (\text{add} (\text{add taskTime1 1} 1) 1)\)

Solution: Transform each “level” of the AST into a timestep of computation
TRANSFORMING SYGUS RESULT TO TEMPORAL STREAM LOGIC (TSL)

Original specification

\[ \square([\text{task} \leftarrow \text{task1}] \lor [\text{task} \leftarrow \text{task2}]) \land [\text{task} \leftarrow \text{task1}] \leftrightarrow [\text{taskTime1} \leftarrow \text{add taskTime1} 1] \land [\text{task} \leftarrow \text{task2}] \leftrightarrow [\text{taskTime2} \leftarrow \text{add taskTime2} 1] \land \text{eq taskTime1} 0 \rightarrow \Diamond(\text{eq taskTime1} 2) \]

SyGuS-synthesized function

\[ S = (\text{add} (\text{add taskTime1} 1) 1) \]

SyGuS Result as TSL Assumption

<table>
<thead>
<tr>
<th>Pre-condition</th>
<th>Program</th>
<th>Post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{eq taskTime1} 0</td>
<td>$S$</td>
<td>\text{eq taskTime1} 2</td>
</tr>
</tbody>
</table>

\[ \square((\text{Pre-condition}) S \rightarrow \text{Post-condition}) \]
Synthesizing a Temporal Stream Logic Modulo Theories Specification

Remove function & predicate interpretations

Combine

TSL Specification With Assumptions

Reactive Synthesis

Executable Code

\[ S = (\text{add} \ (\text{add} \ \text{taskTime1} \ 1) \ 1) \]

Transform SyGuS result to TSL assumption

Reactive (TSL) Assumption
Synthesizing a Temporal Stream Logic Modulo Theories Specification

Remove function & predicate interpretations

Combine

TSL Specification With Assumptions

Reactive Synthesis

S = (add (add taskTime1 1) 1)

Transform SyGuS result to TSL assumption

Executable Code
Synthesizing a Temporal Stream Logic Modulo Theories Specification

Remove function & predicate interpretations

\[ \square(\text{task} \leftarrow \text{task1}) \lor \square(\text{task} \leftarrow \text{task2}) \]
\[ \land \square(\text{task} \leftarrow \text{task1}) \leftrightarrow [\text{taskTime1} \leftarrow \text{add taskTime1 1}] \]
\[ \land \square(\text{task} \leftarrow \text{task2}) \leftrightarrow [\text{taskTime2} \leftarrow \text{add taskTime2 1}] \]
\[ \land \square(\text{eq taskTime1} 0) \rightarrow \Diamond(\text{eq taskTime1 2}) \]

Combine

TSL Specification With Assumptions

\[ S = (\text{add (add taskTime1 1) 1}) \]

Transform SyGuS result to TSL assumption

\[ \square(\text{eq taskTime1} 0) \land [\text{taskTime1} \leftarrow \text{add taskTime1 1}] \]
\[ \land \Diamond(\text{eq taskTime1} 2) \]

Executable Code
COMBINE CONTROL SPECIFICATION WITH THE DATA ASSUMPTION

Temporal Stream Logic (TSL) Specification

\[ ([\text{task} \leftarrow \text{task1}] \lor [\text{task} \leftarrow \text{task2}]) \]
\[ \land [\text{task} \leftarrow \text{task1}] \leftrightarrow [\text{taskTime1} \leftarrow \text{add taskTime1 1}] \]
\[ \land [\text{task} \leftarrow \text{task2}] \leftrightarrow [\text{taskTime2} \leftarrow \text{add taskTime2 1}] \]
\[ \land \text{eq taskTime1 0} \rightarrow \diamond \text{(eq taskTime1 2)} \]

SyGuS Result as TSL Assumption

\[ (\square (\text{eq taskTime1 0}) \]
\[ \land [\text{taskTime1} \leftarrow \text{add taskTime1 1}] \]
\[ \land \bigcirc [\text{taskTime1} \leftarrow \text{add taskTime1 1}] \]
\[ \rightarrow \bigcirc \bigcirc \text{eq taskTime1 2} \]

TSL specification with assumptions: Teaching reactive synthesis that 0+1+1=2!

\[ (\square (\text{eq taskTime1 0}) \]
\[ \land [\text{taskTime1} \leftarrow \text{add taskTime1 1}] \]
\[ \land \bigcirc [\text{taskTime1} \leftarrow \text{add taskTime1 1}] \]
\[ \rightarrow \bigcirc \bigcirc \text{eq taskTime1 2} \]
RESULT CAN NOW BE SYNTHESIZED!

- From our original TSL-MT specification, we obtained the TSL specification with assumptions.
- We know how to synthesize TSL! (CAV ‘19, Haskell ‘19)

TSL specification with assumptions: Teaching reactive synthesis that $0+1+1=2$!

\[
\begin{align*}
\Box(( & \text{eq taskTime1 0} \\
& \land [\text{taskTime1} \leftarrow \text{add taskTime1 1}] \\
& \land \bigcirc [\text{taskTime1} \leftarrow \text{add taskTime1 1}] \\
& \rightarrow \bigcirc\bigcirc \text{eq taskTime1 2})) \rightarrow \\
\Box([\text{task} \leftarrow \text{task1}] \lor [\text{task} \leftarrow \text{task2}] \\
& \land [\text{task} \leftarrow \text{task1}] \leftrightarrow [\text{taskTime1} \leftarrow \text{add taskTime1 1}] \\
& \land [\text{task} \leftarrow \text{task2}] \leftrightarrow [\text{taskTime2} \leftarrow \text{add taskTime2 1}] \\
& \land \text{eq taskTime1 0} \rightarrow \Diamond(\text{eq taskTime1 2})
\end{align*}
\]
Synthesizing a ... Temporal Stream Logic Modulo Theories Specification

Remove function & predicate interpretations

$S = (\text{add} \ (\text{add taskTime1 1}) \ 1)$

Transform SyGuS result to TSL assumption

Executable Code
Synthesizing a Temporal Stream Logic Modulo Theories Specification

- Control Flow Problem
- Data Transformation Problem

1. Derive and solve SyGuS problems
2. SyGuS function
3. Transform SyGuS result to TSL assumption
4. Reactive (TSL) Assumption

TSL Specification With Assumptions

- Temporal Stream Logic (TSL) Specification
- Combine

Reactive Synthesis

Executable Code

\[
S = (\text{add} (\text{add taskTime1 1}) 1)
\]

Transform SyGuS result to TSL assumption
REACTIVE SYNTHESIS

Temporal Logic Specification

Guarantee 3. When a length-four locked burst starts, no other accesses to HREADY is high, so the current burst ends at the fourth occurrence of true initially separately from the case in which it is not.

\[ \square((\text{HMASTLOCK} \land \text{HBURST} = \text{BURST4} \land \text{START} \land \text{HREADY}) \rightarrow \]
\[ \Diamond(\neg \text{START} \lor [3](\neg \text{START} \land \text{HREADY}))), \]
\[ \square((\text{HMASTLOCK} \land \text{HBURST} = \text{BURST4} \land \text{START} \land \neg \text{HREADY}) \rightarrow \]
\[ \Diamond(\neg \text{START} \lor [4](\neg \text{START} \land \text{HREADY})))), \]

Guarantee 6. If we do not start an access in the next time step, the bus

For each master i,

\[ \square(\Diamond(\neg \text{START})) \rightarrow \left( (\text{HMASTER} = i \leftrightarrow \Diamond(\text{HMASTER} = i)) \land \right. \]
\[ \left( \text{HMASTLOCK} \leftrightarrow \Diamond(\text{HMASTLOCK}) \right) \). \]

Assumption 4. We assume that all input signals are low initially.

\[ \bigwedge_{i} (\neg \text{HBUSSREQ}[i] \land \neg \text{HLOCK}[i]) \land \neg \text{HREADY}. \]
# EVALUATION OF TEMOS (FOR TSL-MT)

| Benchmark (φ)         | | ψ | | P | | F | | ψ Generation (s) | TSL Synthesis (s) | Sum (s) | Synthesized LoC |
|-----------------------|---|---|---|---|---|---|-----------------|------------------|---------|----------------|
| **Music Synthesizer** |   |   |   |   |   |   |                 |                  |         |                |
| Vibrato               | 10| 2 | 2 | 21|   |   | 0.431           | 0.914            | 1.345   | 206            |
| Modulation            | 33| 4 | 4 | 41|   |   | 2.012           | 3.983            | 5.995   | 1352           |
| Intertwined           | 58| 4 | 4 | 41|   |   | 2.157           | 3.178            | 5.335   | 1366           |
| Multi-effect          | 27| 6 | 6 | 45|   |   | 3.145           | 81.470           | 84.615  | 1463           |
| **Pong**              |   |   |   |   |   |   |                 |                  |         |                |
| Single-Player         | 27| 1 | 1 | 5 |   |   | 0.043           | 0.571            | 0.614   | 169            |
| Two-Player            | 49| 2 | 2 | 12|   |   | 0.181           | 0.625            | 0.806   | 195            |
| Bouncing              | 27| 3 | 2 | 25|   |   | 0.418           | 0.808            | 1.226   | 169            |
| Automatic             | 27| 5 | 2 | 54|   |   | 0.541           | 0.988            | 1.529   | 214            |
| **Escalator**         |   |   |   |   |   |   |                 |                  |         |                |
| Simple                | 29| 1 | 2 | 2 |   |   | 0.011           | 0.434            | 0.445   | 166            |
| Counting              | 57| 2 | 2 | 8 |   |   | 0.100           | 0.592            | 0.692   | 241            |
| Bidirectional         | 57| 5 | 11| 9 |   |   | 0.340           | 2.291            | 2.631   | 279            |
| Smart                 | 65| 8 | 2 | 34|   |   | 3.034           | 0.935            | 3.969   | 179            |
| **CPU Scheduler**     |   |   |   |   |   |   |                 |                  |         |                |
| Round Robin           | 21| 2 | 4 | 16|   |   | 0.149           | 0.740            | 0.889   | 252            |
| Load Balancer         | 39| 3 | 4 | 12|   |   | 0.531           | 2.128            | 2.659   | 208            |
| Preemptive            | 54| 4 | 4 | 12|   |   | 0.548           | 0.765            | 1.313   | 356            |
| CFS                   | 81| 8 | 5 | 12|   |   | 0.533           | 2.443            | 2.976   | 2825           |
SOME USEFUL LINKS

- Rajeev Alur’s tutorial on SyGuS (additional material: real world applications): https://simons.berkeley.edu/talks/syntax-guided-program-synthesis
- Roderick Bloom’s tutorial on reactive synthesis (additional material: shield synthesis): https://www.newton.ac.uk/seminar/36472/
- Bernd Finkbeiner’s tutorial on reactive synthesis (additional material: bounded synthesis, synthesis of distributed systems): https://simons.berkeley.edu/talks/reactive-synthesis
- Simons program on synthesis: https://simons.berkeley.edu/workshops/synthesis-models-systems/schedule#simons-tabs
CONCLUSIONS

- Software synthesis is an exciting idea that started as an interesting theoretical question (“can we derive the program automatically?”) but today is a part of software development used by millions of users.
- Various types of software synthesis:
  - Reactive synthesis
  - Deductive synthesis / functional synthesis
  - Syntax-guided synthesis
- Which synthesis type to choose (and what is your specification) depends on the application and goal.
- Various applications: network, cyber-psychical systems, AI correctness.
- Synthesis today: connecting many different fields of research.