Syntroids: Synthesizing a Game for FPGAs using Temporal Logic Specifications

Gideon Geier, Philippe Heim, Felix Klein and Bernd Finkbeiner
Reactive Systems Group, Saarland University, Germany
{geier,heim,klein,finkbeiner}@react.uni-saarland.de

***Abstract*—We present *Syntroids*, a case study for the automatic synthesis of hardware from a temporal logic specification. *Syntroids* is a space shooter arcade game realized on an FPGA, where the control flow architecture has been completely specified in Temporal Stream Logic (TSL) and implemented using reactive synthesis. TSL is a recently introduced temporal logic that separates control and data. This leads to scalable synthesis, because the cost of the synthesis process is independent of the complexity of the handled data.**

In this case study, we report on our experience with the TSL-based development of the *Syntroids* game and on the implementation quality obtained with synthesis in comparison to manual programming. We also discuss solved and open challenges with respect to currently available synthesis tools.

I. Introduction

Computationally controlled systems that are embedded into physical products are of ever-growing importance in modern life. They range from simple devices, like a kitchen timer or a heating controller, to enormously complex ones like autonomous vehicles and aircraft. For safety-critical systems, the standard design flow is to first manually write an implementation and then verify the implementation against a formal specification.

An attractive alternative to this design flow is offered by *reactive synthesis*, which automatically creates a correct-by-construction implementation from a specification given in a temporal logic. In practice, however, applying the currently available synthesis tools is difficult. Even though there has been some success in synthesizing *control-intensive* systems, such as the AMBA arbiter [1], synthesis tools often fail due to the complexity of the handled *data*. For standard reactive synthesis, all data structures must be encoded on the bit-level, which, for complex data, results in a far too large state space.

Recently, a new temporal logic, *Temporal Stream Logic* (TSL), has been introduced that specifically addresses this problem [2]. TSL separates the specification of the control structure from the data transformations. This leads to scalable synthesis, because the cost of the synthesis process is independent of the complexity of the handled data.

Supported by the European Research Council (ERC) Grant OSARES (No. 683300) and the German Research Foundation (DFG) as part of the Collaborative Research Center Foundations of Perspicuous Software Systems (TRR 248, 389792660).

In this paper, we describe a case study in which we apply TSL-based synthesis to the development of a non-trivial arcade game. We develop *Syntroids*, a space shooter game realized on an FPGA. The design of the game involves several data-intensive features that need to be handled by TSL, like reading data from an external sensor using an SPI interface, displaying data on a multi-color LED matrix, and managing an open number of enemies in the game's world. To the best of our knowledge, this is the most complex case study for reactive synthesis to date.

The *Syntroids* game is controlled via the orientation and movement of a physical screen, similar to modern smartphone games. The player is inside a spaceship and has to shoot asteroids, also referred to as enemies, rushing at the spaceship from all directions. If the spaceship is hit by an asteroid, then the game is over. The player gets a point for every asteroid taken down. As the game progresses, the difficulty increases with new enemies moving faster and faster. At all times, the player can switch back and forth between three different game modes:

1) If the device is held horizontally (screen is upward) the game switches to *radar mode*. The radar mode shows a top-view of the environment, in which the player can easily determine where and how close enemies are.
2) If the device is held vertically (screen towards the player), the game switches to *cockpit mode*. Cockpit mode shows the view through the windshield of the spaceship. To look into different directions, the player has to turn accordingly. Once enemies are close enough to be visible, the player can shoot at them by aiming the device straight at one of the enemies and pushing the device quickly into the direction of the enemy and back again. The spaceship's laser gun then instantly destroys the asteroid.
3) The *score mode* shows the score, depicted as one dot per point. This mode is chosen by holding the device upside down over the player's head. The score is shown in the color of the most dangerous asteroid.

When the player's spaceship is hit by an asteroid, a game over screen is shown and the game can be restarted by doing a shooting gesture. Pictures of the device's hardware, as well as radar and cockpit mode are shown in Fig. 1.

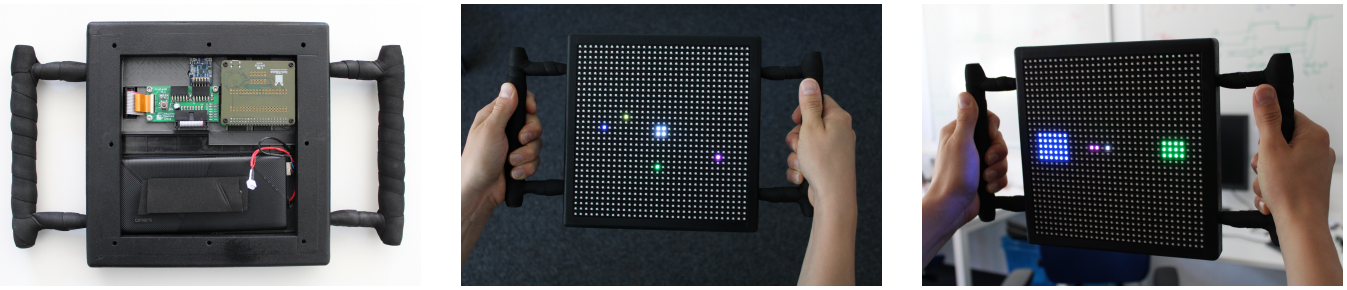


Fig. 1. The hardware (left), radar mode (center) revealing four enemies, and cockpit mode (right) showing three enemies facing the player.

II. Game Architecture

The game's architecture, depicted in Fig. 2, can be organized according to four tasks: management of the game logic, generating LED matrix output, SPI in- and output handling, and controlling sensor data acquirement, as well as storing and converting them into usable signals.

1) Gameplay: The game logic is implemented by the *GameLogic* module and the *EnemyModules*. Every *EnemyModule* stores the data of a single enemy and moves or resets the enemy to a new position, represented as a polar coordinate. Each enemy gets an individual move-clock-signal from *GameLogic* indicating if it must move.

The *GameLogic* also generates random starting angles as respawn points for the enemies. Therefore, it checks, whether an enemy was shot or the player is hit to generate game over or reset signals. Furthermore it handles the score, which increases with every enemy taken down and resets if the game restarts. Overall, the *GameLogic* module can handle a variable number of enemies.

2) Video Output: The *GameModule* manages the drawing process of the game. The three drawing modules (*Cockpit-*, *Radar-* and *ScoreBoard*) work in parallel. They iterate over all pixels, where they output data for one pixel every clock cycle. Based on the game mode and whether the game is over the *GameModule* chooses the right pixel to be written to the video-memory of the *LedMatrix* module.

The *ScoreBoard* manages the printing of the score and the game over screen, depending on the state of the game. The *RadarBoard* prints the radar by calculating Cartesian coordinates for every enemy periodically with respect to the screen's orientation. The cockpit mode is managed by the *CockpitBoard*, whose main task is drawing the enemies as squares depending on their distance and rotation, as well as on the orientation of the player. Therefore, the module checks for every displayed pixel, whether and which enemy it displays.

The LED matrix screen is controlled by the *LedMatrix* module also providing the video memory. Furthermore, it allows to change pixels independently of the update process of the physical LED matrix.

3) User input: A sensor device, attached via PMod interface, is used to determine orientation and movement of the device. To this end, the accelerometer and gyroscope embedded as part of the sensor are used. The communication with the sensor works via a 4-pin SPI.

The *SPI* module implements standard SPI communication by coordinating the *SPI Write* and *SPI Read* modules, which are split into several submodules. For reading and writing, respectively, one of the submodules manages a state, while the others use it to generate output. The *SPI Read* module also has to retrieve data from the `sdo` input pin. The modules all work for variable serial clock speeds.

The sensor submodules are coordinated by the *Sensor* module. In the beginning the *Sensor* module selects the *SensorInit* module, which initializes the device. Afterwards the *SensorPart* modules are scheduled to read their assigned registers, e.g., all accelerometer registers. They communicate with the *SPI* module, the *RegisterManager* and the *SensorSelector*, where the *SubmoduleChooser* only forwards the signals of the currently selected module. The device uses chip select pins for accessing the different sensors. The *SensorSelector* integrates these with the SPI communication and selects them according to which part of the sensor the module is currently communicating with.

4) Data handling: To read the sensor data asynchronously and independent of the current communication state, the data is stored in separate *SensorRegisters*. The *RegisterManager* controls all the registers. Moreover since the sensor values are read in two steps, it caches the intermediate values.

The conversion of the data to meaningful inputs of the game is handled by three converter modules using several (empirical gained) threshold values for the specific sensor outputs. The *RotationCalculator* calculates the player's absolute rotation from his starting position, using the gyroscope's x- and z-axis, depending on the mode of the game. The rotation is gained by integrating the gyroscope's output to the rotation speed. The *GamemodeChooser* works on the gyroscope's y-axis, which is used to differentiate between the three game modes based on the rotation. The *ActionConverter* recognizes shooting and resetting the game, for which the z-axis of the accelerometer is used. To avoid misdetection due to centrifugal force it also utilizes the three gyroscope values.

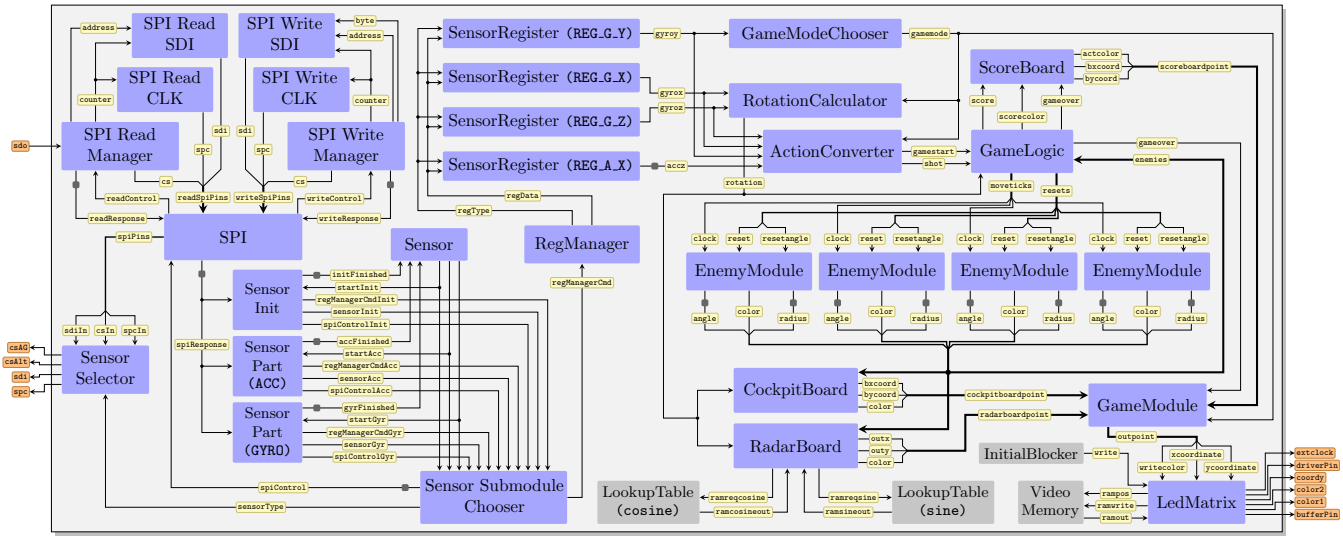


Fig. 2. The architecture of the game. The blue highlighted components have been synthesized from specifications written in TSL.

III. Temporal Stream Logic

The control flow behavior of all highlighted modules (blue) in the architecture of Fig. 2 has been specified using Temporal Stream Logic (TSL) [2]. The logic introduces a clean separation between pure data transformations and temporal control. If a TSL specification is realizable, then it can be turned into a Control Flow Architecture, an abstract representation of the hardware architecture that covers all possible behavior switches. In combination with concretizations for pure data transformations and the functional hardware description language CλaSH [3], the control flow then is implemenented on the FPGA.

Temporal Stream Logic builds on the notion of *updates*, such as $[\![\mathtt{y} \leftarrowtail \mathtt{f\ x}]\!]$ expressing that on every clock cycle the pure function `f` is applied to the input stream `x` and the result is piped to the output stream `y`. These updates are combined with predicate evaluations guiding the temporal control flow decisions. In combination with Boolean and temporal operations, the logic allows for expressing even complex, temporally evolving architectures using only a short, but precise description of the temporal control.

The advantage of TSL is that function and predicate names, as used by the specification, are only considered as symbolic literals. The semantics of the logic then guarantee that synthesized systems satisfy the specified behavior for all possible implementations of these literals. They are only classified according to their arity, i.e., the number of other function terms, they are applied to, as well as by their type: input, output, cell, function or predicate.

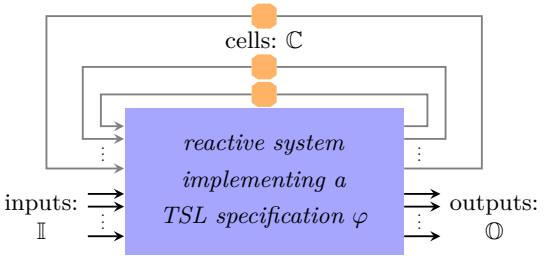


Fig. 3. TSL System Architecture

TSL specifications are evaluated on a synchronous system architecture as shown in Fig. 3. The syntax utilizes a term based notion, build from input streams $\mathtt{i} \in \mathbb{I}$, output streams $\mathtt{o} \in \mathbb{O}$, memory cells $\mathtt{c} \in \mathbb{C}$, and function and predicate literals $\mathtt{f} \in \mathbb{F}$ and $\mathtt{p} \in \mathbb{P}$ with $\mathbb{P} \subseteq \mathbb{F}$, respectively. The purpose of cells is to memorize data values that had been output to a cell at time $t \in \mathcal{Time}$ for providing them again as inputs at time $t+1$. We differentiate between function terms $\tau_F \in \mathcal{T}_F$ and predicate terms $\tau_P \in \mathcal{T}_P$, build according to the following grammar:

$$\begin{aligned} \tau_F &:= \mathtt{s_i} \mid \mathtt{f}\ \tau_F^0\ \tau_F^1\ \cdots\ \tau_F^{n-1} \\ \tau_P &:= \mathtt{p}\ \tau_F^0\ \tau_F^1\ \ldots\ \tau_F^{n-1} \end{aligned}$$

Here, $\mathtt{s_i} \in \mathbb{I} \cup \mathbb{C}$ is either an input stream or a cell. In a TSL formula φ, function terms are then combined to updates, extended with predicate terms, Boolean connectives, and temporal operators:

$$\varphi := \tau_P \mid [\![\mathtt{s_o} \leftarrowtail \tau_F]\!] \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\varphi$$

where $\mathtt{s_o} \in \mathbb{O} \cup \mathbb{C}$ is either an output signal or a cell.

The semantics of a TSL formula φ utilize a universally quantified assignment function $\langle\cdot\rangle\colon \mathbb{F} \to \mathcal{F}$, fixing an implementation for each predicate and function literal, as well as input streams ι. We only give an intuitive description of the semantics here. For a fully formal description the interested reader is referred to [2]. Intuitively, the semantics of TSL are summarized as follows:

- **Predicate terms** evaluate to either `true` or `false`, by first selecting implementations for all function and predicate literals according to $\langle\cdot\rangle$, and then applying them to inputs, as given by ι, and cells, using the stored value at the current time t. The content of a cell thereby is fixed iteratively, by selecting the past values

piped into the cell over time. Cells are initialized using a special constant, provided as part of $\langle\cdot\rangle$.

- **Function terms** evaluate similar to predicate terms, except that they evaluate to values of arbitrary type.
- **Updates** are used to pipe the results of function term evaluations to output streams or cells. Therefore, updates, as they appear in a TSL formula, semantically are typed as Boolean expressions. In that sense, update expressions state that a specific flow is executed at a specific time, where an update evaluates to `true` if it is used and to `false`, otherwise. Outputs or cells only can receive a single update at any time.
- The **Boolean operators** *negation* $[\neg]$ and *conjunction* $[\wedge]$, and the **temporal operators** *next* $[\bigcirc]$ and *until* $[\mathcal{U}]$ have standard semantics and feature the default derived operators such as *release* $[\varphi\,\mathcal{R}\,\psi \equiv \neg((\neg\psi)\mathcal{U}(\neg\varphi))]$, *finally* $[\Diamond\varphi \equiv true\,\mathcal{U}\,\varphi]$, *always* $[\Box\varphi \equiv false\,\mathcal{R}\,\varphi]$, and the *weak* version of *until* $[\varphi\,\mathcal{W}\,\psi \equiv (\varphi\,\mathcal{U}\,\psi)\vee(\Box\varphi)]$. The precedence order of the listed operators matches the listed order, except that $\Box$ and $\Diamond$ have higher precedence than $\mathcal{U}$ and $\mathcal{R}$.

The synthesis problem of creating a control flow architecture A that satisfies a TSL specification φ is stated by

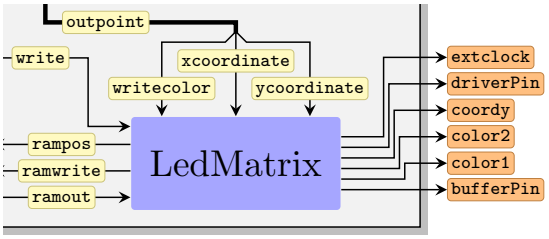
$$\exists A.\ \forall\iota.\ \forall\langle\cdot\rangle.\ A\wr\iota, \iota \models_{\langle\rangle} \varphi$$

where $A\wr\iota$ denotes the output produced by A under the input ι. Note that A must satisfy the specification for all possibly chosen function and predicate implementations, as selected by $\langle\cdot\rangle$, and all possible inputs ι.

IV. Module Specifications

We describe the development process of using TSL for the creation of the *Syntroids* game components. We discuss the full step-by-step design process for the LedMatrix module and highlight some insights for the other modules.

A. LedMatrix



The LedMatrix module is responsible for displaying images on the physical LED matrix screen. Therefore, it has to send control data over the hardware pins and needs to interact with a memory module serving as video memory. It receives writing commands for individual pixels, passed via `writecolor` and the coordinates `xcoordinate` and `ycoordinate`, and a control bit `write`, indicating whether a pixel must be written to the video memory. The module is also able to provide the pixel's `color`, determined by the delivered coordinates (with a certain delay), which is, however, not used by our application at the moment.

The LED matrix hardware interface splits the screen into two halves, which are operated in parallel. On every half, the same single column is active at a time, while all other columns are turned off. The active column is selected by the 16-bit $\texttt{coord}_y$-Pin. Each half additionally uses a register for holding the content of the shown column and a buffer register of the size of one row. By turning `driverPin` high the driver register can be turned off. If `bufferPin` is high, then the content of the buffer register is moved to the driver register.

The writing procedure for operating the matrix cycles through all columns for writing to the buffer registers and for flushing the content to the driver register, before showing the corresponding column. The buffer registers are shift registers, which shift their content each time `extclock` rises (also referred to as clock) and hold 3-bit color values outputted to $\texttt{color}_1$ and $\texttt{color}_2$, one for each half, respectively. Due to electrical characteristics of the LED matrix multiple LEDs may turn on, even if only a single LED is lighted up, if the data is written to fast. This effect is known as ghosting and is avoided by slowing down the writing process.

For writing to the video memory over `ramwrite`, a command consisting of a color and an address needs to be transmitted. The video memory also delivers a pixel `ramout`, if requested with an address over `rampos`. Writing and reading are independent, but it is only possible to read a single value at a time.

The specification of the LedMatrix module is given by $\varphi_A \rightarrow (\bigwedge_{i=1}^{2}\psi_i^I) \wedge \Box(\bigwedge_{i=1}^{21}\psi_i)$ and covers the following tasks:

1) Memory Interaction: Whenever a color is taken from the video-memory, it must be preceded by a lookup action.

$$\begin{aligned}\psi_1 \;:=\; &(\bigcirc[\![\texttt{color}_1 \leftarrowtail \texttt{ramout}]\!]) \\ &\rightarrow [\![\texttt{rampos} \leftarrowtail \texttt{rampos}_1\ \texttt{coord}_x\ (\texttt{coord}_y + 1)]\!]\end{aligned}$$

$$\begin{aligned}\psi_2 \;:=\; &(\bigcirc[\![\texttt{color}_2 \leftarrowtail \texttt{ramout}]\!]) \\ &\rightarrow [\![\texttt{rampos} \leftarrowtail \texttt{rampos}_2\ \texttt{coord}_x\ (\texttt{coord}_y + 1)]\!]\end{aligned}$$

$$\begin{aligned}\psi_3 \;:=\; &(\bigcirc[\![\texttt{color}_R \leftarrowtail \texttt{ramout}]\!]) \\ &\rightarrow [\![\texttt{rampos} \leftarrowtail \texttt{rampos}_R\ \texttt{xcoordinate}\ \texttt{ycoordinate}]\!]\end{aligned}$$

However, there is no lookup action initially.

$$\begin{aligned}\psi_1^I \;:=\; &\neg[\![\texttt{color}_1 \leftarrowtail \texttt{ramout}]\!] \wedge \neg[\![\texttt{color}_2 \leftarrowtail \texttt{ramout}]\!] \wedge \\ &\neg[\![\texttt{color}_R \leftarrowtail \texttt{ramout}]\!]\end{aligned}$$

The literals $\texttt{coord}_x$ and $\texttt{coord}_y$ are cells storing the x- and y-coordinates internally. Note that we use notions such as $(+1)$ for TSL literals without pre-assigned semantics for improved readability reasons.

Only if the write signal is high, the passed color is written to the video memory. Otherwise it remains unchanged.

$$\begin{aligned}\psi_4 \;:=\; &\texttt{write} \rightarrow [\![\texttt{ramwrite} \leftarrowtail \\ &\texttt{writeram}\ \texttt{writecolor}\ \texttt{xcoordinate}\ \texttt{ycoordinate}]\!]\end{aligned}$$

$$\psi_5 \;:=\; \neg\texttt{write} \rightarrow [\![\texttt{ramwrite} \leftarrowtail \texttt{writeramnone}()]\!]$$

Finally, the color is output infinitely often.

$$\psi_6 \;:=\; \Diamond[\![\texttt{color}_R \leftarrowtail \texttt{ramout}]\!]$$

2) External Clock Generation: The clock is initially low and toggles between low and high infinitely often. We use () after function literals to mark them as constants.

$$\psi_2^I := [\![\texttt{extclock} \leftarrowtail \texttt{low}()]\!]$$
$$\psi_7 := \Diamond[\![\texttt{extclock} \leftarrowtail \texttt{high}()]\!] \wedge \Diamond[\![\texttt{extclock} \leftarrowtail \texttt{low}()]\!]$$

Whenever the clock is high, then the outputs are stable.

$$\psi_8 := [\![\texttt{extclock} \leftarrowtail \texttt{high}()]\!] \rightarrow [\![\texttt{color}_1 \leftarrowtail \texttt{color}_1]\!] \wedge [\![\texttt{color}_2 \leftarrowtail \texttt{color}_2]\!] \wedge [\![\texttt{coord}_x \leftarrowtail \texttt{coord}_x]\!] \wedge [\![\texttt{coord}_y \leftarrowtail \texttt{coord}_y]\!]$$

3) Pixel Updates: The module changes the x-coordinate $\texttt{coord}_x$ infinitely often for printing out at every pixel.

$$\psi_9 := \Diamond[\![\texttt{coord}_x \leftarrowtail \texttt{coord}_x + 1]\!]$$

For writing the correct color between each generated clock cycle we output colors at both colors pins:

$$\psi_{10} := [\![\texttt{extclock} \leftarrowtail \texttt{low}()]\!] \rightarrow \big([\![\texttt{color}_1 \leftarrowtail \texttt{ramout}]\!] \mathcal{R} \neg[\![\texttt{extclock} \leftarrowtail \texttt{high}()]\!]\big)$$
$$\psi_{11} := [\![\texttt{extclock} \leftarrowtail \texttt{low}()]\!] \rightarrow \big([\![\texttt{color}_2 \leftarrowtail \texttt{ramout}]\!] \mathcal{R} \neg[\![\texttt{extclock} \leftarrowtail \texttt{high}()]\!]\big)$$

However, both of these color settings are preceded by a ram lookup that requires the correct coordinates. Since the led matrix works by using a shift register, the module has to adjust the internal x-coordinate before looking up the colors, but only once for every clock cycle.

$$\psi_{12} := [\![\texttt{extclock} \leftarrowtail \texttt{low}()]\!] \rightarrow \big([\![\texttt{coord}_x \leftarrowtail \texttt{coord}_x + 1]\!] \mathcal{R} \neg([\![\texttt{color}_1 \leftarrowtail \texttt{ramout}]\!] \vee [\![\texttt{color}_2 \leftarrowtail \texttt{ramout}]\!] \vee [\![\texttt{rampos} \leftarrowtail \texttt{rampos1 coord}_x\ (\texttt{coord}_y + 1)]\!] \vee [\![\texttt{rampos} \leftarrowtail \texttt{rampos2 coord}_x\ (\texttt{coord}_y + 1)]\!] \vee [\![\texttt{extclock} \leftarrowtail \texttt{high}()]\!])\big)$$
$$\psi_{13} := [\![\texttt{coord}_x \leftarrowtail \texttt{coord}_x + 1]\!] \rightarrow \bigcirc\big([\![\texttt{extclock} \leftarrowtail \texttt{high}()]\!] \mathcal{R} \neg[\![\texttt{coord}_x \leftarrowtail \texttt{coord}_x + 1]\!]\big)$$

When reaching the maximum x-value the module has to adjust the y-coordinate for writing the next row

$$\psi_{14} := [\![\texttt{coord}_x \leftarrowtail \texttt{coord}_x + 1]\!] \wedge (\texttt{coord}_x = \texttt{size}_x() - 1) \leftrightarrow [\![\texttt{coord}_y \leftarrowtail \texttt{coord}_y + 1]\!]$$

while exactly then, it also prints the content of the buffer.

$$\psi_{15} := [\![\texttt{bufferPin} \leftarrowtail \texttt{high}()]\!] \leftrightarrow \big(\bigcirc[\![\texttt{coord}_x \leftarrowtail \texttt{coord}_x + 1]\!] \wedge (\texttt{coord}_x = \texttt{size}_x() - 1)\big)$$
$$\psi_{16} := [\![\texttt{bufferPin} \leftarrowtail \texttt{high}()]\!] \vee [\![\texttt{bufferPin} \leftarrowtail \texttt{low}()]\!]$$

Note that formula ψ_{15} avoids that: if not changing the output, the buffer pass-through is never active. As the driver pin is not used we fix it to be always low:

$$\psi_{17} := [\![\texttt{driverPin} \leftarrowtail \texttt{low}()]\!]$$

4) Ghosting Elimination: The last subroutine introduces delay to avoid ghosting. The delay itself is handled by `waitcounter`. If the clock is low, then the counter starts. The x-coordinate does not change until it reaches zero again due to an overflow.

$$\psi_{18} := [\![\texttt{extclock} \leftarrowtail \texttt{low}()]\!] \rightarrow \neg[\![\texttt{waitcounter} \leftarrowtail \texttt{waitcounter} + 1]\!] \wedge [\![\texttt{coord}_x \leftarrowtail \texttt{coord}_x + 1]\!]$$
$$\psi_{19} := \neg(\texttt{eqz waitcounter}) \rightarrow [\![\texttt{waitcounter} \leftarrowtail \texttt{waitcounter} + 1]\!]$$
$$\psi_{20} := [\![\texttt{waitcounter} \leftarrowtail \texttt{waitcounter} + 1]\!] \rightarrow (\texttt{waitcounter} = 0) \vee [\![\texttt{extclock} \leftarrowtail \texttt{low}()]\!]$$

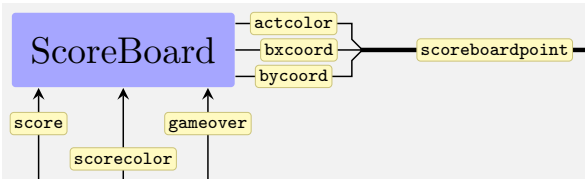
However, until the overflow, $\texttt{coord}_\texttt{x}$ does not change

$$\psi_{21} := [\![\texttt{extclock} \leftarrowtail \texttt{low}()]\!] \rightarrow \bigcirc\big(\neg[\![\texttt{coord}_x \leftarrowtail \texttt{coord}_x + 1]\!] \mathcal{U} (\texttt{waitcounter} \neq 0)\big)$$

and because of ψ_{12} the whole writing process is stalled. The stalling must end eventually, which is satisfied since the counter overflows. However, this behavior requires information on the data, i.e., the assumption

$$\varphi_A := \square\Diamond(\texttt{waitcounter} \neq 0)$$

B. Scoreboard

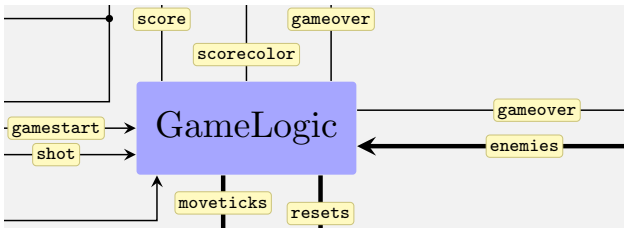


The Scoreboard module receives the `score`, its color `scorecolor` and a Boolean signal `gameover` indicating if the game is over. It provides a single point which consist of an x-coordinate `bxcoord`, a y-coordinate `bycoord` and a color value `actcolor`, which may be written to the video-memory. To draw the right image the module cycles over all pixels and writes the appropriate colors:

$$\square[\![\texttt{xcoord} \leftarrowtail \texttt{xcoord} + 1]\!] \wedge \square\big((\texttt{xcoord} = \texttt{size}_x() - 1) \leftrightarrow [\![\texttt{ycoord} \leftarrowtail \texttt{ycoord} + 1]\!]\big)$$

where `xcoord` and `ycoord` are internal 5-bit values that may overflow. Using such counters is on the one hand necessary, since currently available synthesis tools are not able to handle specifications with large numbers of $\bigcirc$-chains, and on the other hand useful, as they automatically parameterize the module for possibly different screen sizes.

C. GameLogic



The GameLogic module manages the logic and state of the game. It chooses between game over and running mode, selects the score, and coordinates the enemies with the actions of the player. Among others, it receives Boolean signals indicating the `gamestart` and whether the player has `shot`, as well as the data of all enemies bundled together to `enemies`. Properties of individual enemies are selected with functions like `getenemyangle` and `getenemyradius` using the enemy index, realized through the internal cell `counter`. A useful design feature of TSL is that it automatically ensures conflict free management of streams, even at different places. An example is the output stream `score`, which depends on the game state satisfying

$$\square\,\big(\texttt{gamestart} \leftrightarrow [\![\,\texttt{score} \leftarrowtail \texttt{zeroScore}()\,]\!]\big)$$

The property states that the `score` is reset to zero if the game restarts. At the same time the condition

$$\square\,\big([\![\,\texttt{gameover} \leftarrowtail \texttt{high}()\,]\!] \wedge \neg\texttt{gamestart} \rightarrow [\![\,\texttt{score} \leftarrowtail \texttt{score}\,]\!]\big)$$

requires that if the game is over and is not restarted, then the score does not change, which ensures that the score is not changed when the game is over. The semantics of TSL ensure that both properties are realizable simultaneously, while the synthesis engine takes care that there indeed is a conflict free resolution. The feature especially pays off as soon as more properties are added. For example, another requirement is the correct coordination of enemies and player actions. Especially, if the player shoots an enemy the score must be increased.

$$\begin{aligned}\square\bigcirc\big(&\neg\texttt{gamestart} \wedge [\![\,\texttt{gameover} \leftarrowtail \texttt{low}()\,]\!]\\ &\rightarrow \big([\![\,\texttt{score} \leftarrowtail \texttt{score} + 1\,]\!] \leftrightarrow \big((\texttt{shotCounter} > 0) \wedge\\ &\qquad \texttt{hitenemy}\ (\texttt{getenemyangle}\ \texttt{enemies}\ \texttt{counter})\\ &\qquad\quad (\texttt{getenemyangle}\ \texttt{enemiescounter})\ \texttt{rotation}\big)\big)\big)\end{aligned}$$

D. Sensor

The *Sensor* module coordinates the different sensor submodules via the `partControl` output selecting the signal from each module. It has to meet the following properties:

1) The sensor must be initialized first.
2) All reading submodules must be started repeatedly.
3) An active module has to finish before starting the next one.
4) It is forbidden to repeat the initialization.

The second and third condition are declared as follows:

$$\begin{aligned}\psi_1 &:= \square\diamondsuit[\![\,\texttt{partControl} \leftarrowtail \texttt{accOn}()\,]\!]\\ \psi_2 &:= \square\diamondsuit[\![\,\texttt{partControl} \leftarrowtail \texttt{gyrOn}()\,]\!]\\ \psi_3 &:= \square(\neg[\![\,\texttt{partControl} \leftarrowtail \texttt{initOn}()\,]\!] \wedge\\ &\qquad \neg\texttt{gyrFinished} \wedge \neg\texttt{accFinished} \wedge \neg\texttt{initFinished}\\ &\qquad\quad \rightarrow [\![\,\texttt{partControl} \leftarrowtail \texttt{noCmd}()\,]\!])\end{aligned}$$

The update $[\![\,\texttt{partControl} \leftarrowtail \texttt{initOn}()\,]\!]$ is necessary for the specification to be realizable. Otherwise, the initialization would be forbidden, since there is no finished signal initially. It is assumed that the other modules will return a *finished* signal after being started

$$\begin{aligned}\varphi_1 &:= [\![\,\texttt{partControl} \leftarrowtail \texttt{accOn}()\,]\!] \rightarrow \bigcirc\diamondsuit\,\texttt{accFinished}\\ \varphi_2 &:= [\![\,\texttt{partControl} \leftarrowtail \texttt{gyrOn}()\,]\!] \rightarrow \bigcirc\diamondsuit\,\texttt{gyrFinished}\end{aligned}$$

which is necessary for realizability, since otherwise the eventuality cannot be satisfied if all inputs are always low.

E. SensorPart

This module is configured using six register addresses, a sensor type determining the right chip select and a module type to choose the correct register. In the specification the following structure is used repeatedly

$$\varphi_{\mathtt{A}}\,\mathcal{R}\,((\varphi_{\mathtt{A}} \rightarrow \varphi_{\mathtt{B}}) \wedge (\neg\varphi_{\mathtt{A}} \rightarrow \varphi_{\mathtt{C}}))$$

where $\varphi_{\mathtt{A}}$ depends on an input and $\varphi_{\mathtt{B}}$ and $\varphi_{\mathtt{C}}$ are output assigning updates, the formula specifies a state in which the module waits for $\varphi_{\mathtt{A}}$, meanwhile outputting $\varphi_{\mathtt{C}}$. If $\varphi_{\mathtt{A}}$ happens, then it switches the state with output $\varphi_{\mathtt{B}}$. The formula is used to ensure a sequence of actions. A followup state is defined using a $\bigcirc$-operation and the same structure as in the specification above.

$$\varphi_{\mathtt{B}} \rightarrow \bigcirc(\varphi_{\mathtt{A}}\,\mathcal{R}\,((\varphi_{\mathtt{A}} \rightarrow \varphi_{\mathtt{D}}) \wedge (\neg\varphi_{\mathtt{A}} \rightarrow \varphi_{\mathtt{C}})))$$

The formula structure is used to sequentially read all six registers, to finish and to wait for the next start signal. We use ANSWER as an alias for `spiFinished spiResponse`.

$$\begin{aligned}\square\big([\![\,\texttt{spiControl} \leftarrowtail\ &\texttt{readCmd}\ \texttt{reg}_1()\,]\!] \rightarrow \bigcirc\big(\textsc{answer}\\ \mathcal{R}\big(\ &(\ \textsc{answer} \rightarrow [\![\,\texttt{spiControl} \leftarrowtail \texttt{readCmd}\ \texttt{reg}_2()\,]\!])\\ &\wedge(\neg\textsc{answer} \rightarrow [\![\,\texttt{spiControl} \leftarrowtail \texttt{noCmd}()\,]\!])\big)\big)\end{aligned}$$

The property is repeated for registers $\texttt{reg}_2(), \ldots, \texttt{reg}_6()$. There is a *RegManager* command generated after each reception of an answer, which is specified by equivalence with the next read command, executed at the same time.

$$\begin{aligned}\square\,[\![\,&\texttt{spiControl} \leftarrowtail \texttt{readCmd}\ \texttt{reg}_1()\,]\!]\\ &\leftrightarrow [\![\,\texttt{regManagerCmd} \leftarrowtail \texttt{setRegister}\\ &\qquad \texttt{moduleType}()\ \texttt{zero}()\ \texttt{spiResponse}\,]\!]\end{aligned}$$

V. Experimental Results

With specifications for all modules at hand, we first synthesize the control using the TSL synthesis toolchain [4] in combination with the game based LTL synthesizer Strix [5] and the bounded synthesizers BoSy [6] and BoWSer [7]. As a result, we obtain a source code module for every synthesized component that is implemented for the hardware description language CλaSH [3] and parameterized in the universally quantified functions. These parameters then are instantiated with manually created implementations for 42 functions, 24 predicates and 10 data types, implemented with less than 200 lines of CλaSH code. The modules then are wired together according to Fig. 2 and compiled to Verilog code using the CλaSH compiler. Finally, using the open synthesis framework Yosys and the place-and-route tool Nextpnr [8] the code is turned into a binary to be uploaded to the FPGA.

The project is implemented on an *icoBoard* with an *iCE40 hx8k FPGA* providing 7680 LCs and a 100MHz

TABLE I

Module	G		A		M	Bosy			Bowser			Strix		
	L	T	L	T		Time	Lat	Gat	Time	Lat	Gat	Time	Lat	Gat
ActionConverter (AC)	4	0	0	0	1	**0.316**	1	8	4.384	**0**	**4**	1.192	**0**	**4**
Cockpitboard (CB)	12	0	0	0	1	1548.48	1	11	227.136	**0**	**7**	**6.512**	**0**	**7**
EnemyModule (EM)	4	0	0	0	1	**0.272**	1	6	0.904	**0**	**2**	1.196	**0**	**2**
Gamelogic (GL)	15	6	0	0	$3 \vee 4$	> 99999	-	-	13230.5	**2**	226	**696.288**	**2**	**29**
GamemodeChooser (GC)	7	0	4	0	1	111.072	1	100	13247.0	**0**	2377	**2.164**	1	**35**
Gamemodule (GM)	3	0	3	0	1	**0.328**	1	10	1.056	**0**	**3**	1.288	1	11
LedMatrix (LM)	14	13	0	1	≤ 32	> 99999	-	-	> 99999	-	-	**53732.0**	**5**	**101**
Radarboard (RB)	13	0	0	0	1	41319.2	1	10	**26.448**	**0**	**6**	79.376	**0**	**6**
RegisterManager (RM)	5	0	0	0	1	0.292	1	4	**0.164**	**0**	**0**	1.084	**0**	**0**
RotationCalculator (RC)	5	0	3	0	1	**1.324**	1	18	8045.37	**0**	**9**	1.66	1	22
SPI (SPI)	6	9	2	0	3	> 99999	-	-	13214.7	**3**	413	**3.608**	**3**	**72**
SPIReadClk (SPI_R)	2	0	0	0	1	**0.272**	1	4	0.348	**0**	**2**	1.168	**0**	**2**
SPIReadManag (SPI_R)	9	2	1	1	2	3497.26	1	31	11821.0	**1**	**10**	**14.684**	2	27
SPIReadSdi (SPI_R)	2	0	2	0	1	**0.304**	1	5	0.844	**0**	**1**	1.36	1	5
SPIWriteClk (SPI_W)	2	0	0	0	1	**0.276**	1	6	3.628	**0**	**4**	1.196	**0**	**4**
SPIWriteManag (SPI_W)	7	2	3	1	2	196.808	**1**	**6**	61.704	**1**	**6**	**2.22**	**1**	**6**
SPIWriteSdi (SPI_W)	3	0	5	0	1	**0.396**	1	15	12.536	**0**	**4**	1.3	1	11
Scoreboard (SB)	7	0	0	0	1	**1.26**	1	8	15.576	**0**	**4**	1.516	**0**	**4**
Sensor (Sen)	2	4	0	4	4	7429.74	**2**	**29**	> 99999	-	-	**1.912**	4	70
SensorInit (Sen)	2	12	0	0	9	159.076	**4**	95	6613.8	**4**	84	**3.676**	**4**	**46**
SensorPart (Sen)	9	9	0	0	8	1985.21	**3**	34	12224.9	**3**	64	**13.864**	**3**	**30**
SensorRegister (RM)	1	0	0	0	1	0.292	1	2	**0.048**	**0**	**0**	1.188	**0**	**0**
SensorSelector (SS)	5	0	4	0	1	> 99999	-	-	**37.884**	**0**	**0**	277.288	1	17
SensorSubmodulChooser (Sen)	1	4	6	0	4	766.084	**2**	44	13007.8	**2**	369	**3.176**	3	**39**

clock. Due to timing constraints, however, it runs on a prescaled clock of 10MHz. The screen is an Adafruit LED matrix consisting of 32×32 RGB LEDs. Input movements are obtained from a Digilent PModNav module featuring an accelerometer and a gyroscope sensor. All sources of the project are available at:

`react.uni-saarland.de/casestudies/syntroids`

Our experimental results for synthesizing control flow architectures from TSL are depicted in Table I. For each module we counted the number of guarantees (**G**) and assumptions (**A**) split into temporal (**T**) and non-temporal (**L**) sub-formulas. Whenever possible, we used BoWSer to determine the number of states of the smallest Mealy machine satisfying the specification (**M**). For each tool and module, we measured the synthesis time in seconds (**Time**) and the number of AIGER latches (**Lat**) and gates (**Gat**) of the generated circuit, where for each module the highlighted result was used in the final implementation.

We also compared the synthesized game with a manually created reference implementation with respect to the number of logic cells (LCs) used and timing guarantees provided by NextPNR. The results of Fig. 4 show differences in LCs and timing when swapping a module in the hand-made game with a synthesized one. The used abbreviations are defined in Table I. Similar results are shown by Fig. 5, except that there a group of modules is swapped. All modules being swapped is reflected by *All*.

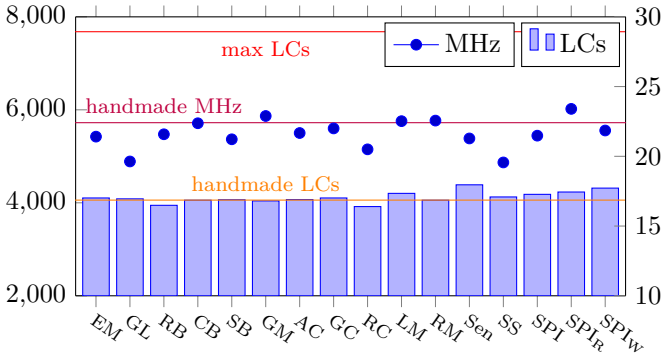


Fig. 4. LCs & timing for a single module swapped.

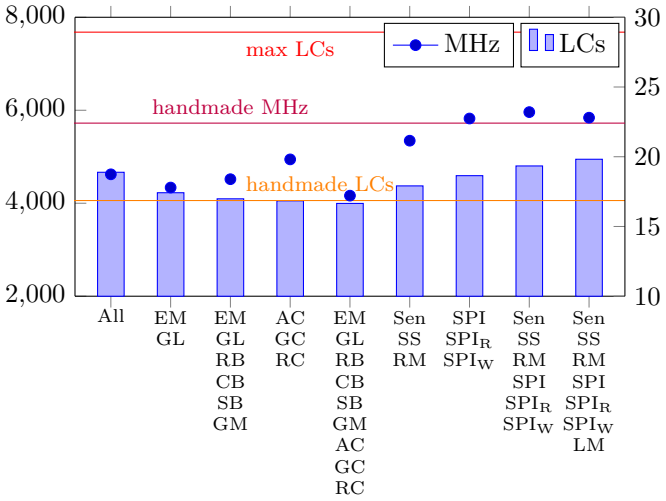


Fig. 5. LCs & timing for multiple modules swapped.

VI. Discussion

Our study shows that TSL synthesis provides several advantages over manual programming.

1) Behavior Descriptions: One major advantage of synthesis is that a specification describes control behavior

much better than a classic program or hardware description. The following situations provide some examples:

a) Data Manipulation at different Places: If data is manipulated that depends on many different logical conditions, which might even be part of different subroutines, then TSL outperforms classically created code. An example is the `score` value of the GameLogic module, which is manipulated at different places and depends on many different conditions. When handling `score` by hand, e.g., in CλaSH or Verilog, the value that is output to `score` must be handled consistently. Hence, it must be guarded by the right conditions, which, however, are also affected at all positions, where `score` is currently used. This is not only tedious, but also a highly error-prone task.

b) Scheduling by Order Constraints: Using a partial order for describing how events must follow each other is much easier than always fixing a total order. Examples are the conditions described in the SensorPart module of the form $\varphi_{\mathtt{B}} \rightarrow \bigcirc(\varphi_{\mathtt{D}}\,\mathcal{R}((\varphi_{\mathtt{D}} \rightarrow \varphi_{\mathtt{E}}) \wedge (\neg\varphi_{\mathtt{D}} \rightarrow \varphi_{\mathtt{F}})))$ or conditions that reference updates, which will happen in the future, as in the LedMatrix module.

$$\square\big((\bigcirc[\![\,\mathtt{color}_1 \leftarrowtail \mathtt{ramout}\,]\!]) \rightarrow [\![\,\mathtt{rampos} \leftarrowtail \mathtt{rampos}_1\ \mathtt{coord}_x\ (\mathtt{coord}_y + 1)\,]\!]\big)$$

c) Schedulability: Stating that updates happen repeatedly is easy to specify using $\square\diamondsuit$, without the need of giving any fixed order, e.g., in the Sensor module:

$$\psi_1 := \square\diamondsuit[\![\,\mathtt{partControl} \leftarrowtail \mathtt{accOn}()\,]\!]$$
$$\psi_2 := \square\diamondsuit[\![\,\mathtt{partControl} \leftarrowtail \mathtt{gyrOn}()\,]\!]$$

2) Optimally Timed Solutions: Another interesting observation is that synthesis tools are able to create optimally timed solutions, e.g. if there is an update that happens repeatedly, then it is possible to specify its length by using a hard bound in form of a $\bigcirc$-chain. An example is a simplified version of the LedMatrix specification that does not take care of ghosting. In this case, the internal x-coordinate must be increased infinitely often. The cycle length between these increases can be specified using $\bigcirc^n[\![\,\mathtt{coord}_x \leftarrowtail \mathtt{coord}_x + 1\,]\!]$ for $n \in \mathbb{N}$. In a separate test series we found, that with $n = 5$ the module is realizable, but with $n = 4$ it is not. Therefore, the minimal cycle length is five. Hence, it is easily possible to enforce the minimal "time density" of cyclic behaviour, which would be hard to provably achive in a manual implementation.

3) Easy expandability: Another advantage is that modules can be easily expanded by adding new properties to the specification. An example is the Ghosting Elimination, which can be added to the LedMatrix specification without the need of changing any of the remaining properties.

4) Modification and Reuse: Due to the separation of data and control, a module can be modified through small changes on the data level, without affecting its properties on the control level. For example the stalling time of the LedMatrix or the screen size handled by Scoreboard are easy to change on the data level. Also modules can be reused for similar tasks, which differ only in the data they work on. An example is the SensorPart module, which is used multiple times to implement different parts of the sensor by instantiating it with different function implementations on the data level.

5) Verification: Synthesis has the verification problem included. It is especially easy to add new conditions, which would not be necessary for determining the behavior, but are important additional safety conditions.

VII. Further Work

There are are also several open challenges.

1) Synthesis Times: TSL synthesis is a fairly hard problem such that it was foreseeable that synthesis tools took quite some time for the more sophisticate specifications (cf. Table I). Thus, these specifications indicate benchmarks, for which synthesis tools still have to leverage improvements for the future.

2) Next Chains: Synthesis tools yet are not able to cope with long chains of $\bigcirc$, e.g., when describing a bounded waiting process. Although TSL allows to circumvent the problem by pushing waiting times to data counters, it may be the intend of the developer to specifically constraint the bounded behavior at the control level.

3) Specification Debugging: As specifications are still written by humans, and humans are prone to make mistakes, the specifications still might be incorrect, especially finding unrealizability reasons is difficult. Hence, we need better debugging tools that help with identifying the mistakes and provide strategies for their resolution.

4) Module Distribution: TSL synthesis allows the creation of modules independently of each other, to be finally composed to a single architecture, like we did with Fig. 2. However, it might be necessary to specify global properties of the system to ensure the correct interaction of multiple modules as well. For example, a related problem, that we encountered, was that using multiple specifications does not prevent the introduction of latch-free cycles on paths between multiple modules. We had to introduce them manually (cf. unlabeled gray boxes in Fig. 2) while taking care that they indeed preserve the intended behavior.

VIII. Conclusion

We have presented *Syntroids*, the first interactive and reactive hardware game that has been completely specified with Temporal Stream Logic and is synthesized from the created specifications using current state-of-the-art synthesis tools. Our experience shows that Temporal Stream Logic is indeed a feasible design flow for the development of reactive systems, providing significant advantages over manual programming. We also identified challenges that remain to be solved in order to accomplish a robust TSL-based development process.

Acknowledgements. We thank Dan Gisselquist for debugging a design issue with us that lead to nondeterministic timing behavior and the Yosys and CλaSH development teams for rapidly resolving our reported issues.

References

[1] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer, "Automatic hardware synthesis from specifications: A case study," in *2007 Design, Automation Test in Europe Conference Exhibition*, April 2007, pp. 1–6.

[2] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito, "Temporal stream logic: Synthesis beyond the bools," in *Computer Aided Verification - 31th International Conference, CAV 2019, New York, NY, USA, July 15-18, 2019, Proceedings, Part I*, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-25540-4_35

[3] C. Baaij, "Digital circuit in cλash: functional specifications and type-directed synthesis," Ph.D. dissertation, 1 2015, eemcs-eprint-23939.

[4] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito, "Synthesizing functional reactive programs," *CoRR*, vol. abs/nnnn.nnnnn, 2019, available at https://www.react.uni-saarland.de/publications/FKPS19b.html.

[5] P. J. Meyer, S. Sickert, and M. Luttenberger, "Strix: Explicit reactive synthesis strikes back!" in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10981. Springer, 2018, pp. 578–586. [Online]. Available: https://doi.org/10.1007/978-3-319-96145-3_31

[6] P. Faymonville, B. Finkbeiner, and L. Tentrup, "Bosy: An experimentation framework for bounded synthesis," in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, ser. Lecture Notes in Computer Science, R. Majumdar and V. Kuncak, Eds., vol. 10427. Springer, 2017, pp. 325–332. [Online]. Available: https://doi.org/10.1007/978-3-319-63390-9_17

[7] S. Jacobs, R. Bloem, M. Colange, P. Faymonville, B. Finkbeiner, A. Khalimov, F. Klein, M. Luttenberger, P. J. Meyer, T. Michaud, M. Sakr, S. Sickert, L. Tentrup, and A. Walker, "The 5th reactive synthesis competition (SYNTCOMP 2018): Benchmarks, participants & results," *CoRR*, vol. abs/1904.07736, 2019. [Online]. Available: http://arxiv.org/abs/1904.07736

[8] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic, "Yosys+nextpnr: an open source framework from verilog to bitstream for commercial fpgas," *CoRR*, vol. abs/1903.10407, 2019. [Online]. Available: http://arxiv.org/abs/1903.10407