Applying High-Level Synthesis for Synthesizing Hardware Runtime STL Monitors of Mission-Critical Properties

K.Selyunin[1] T.Nguyen[2] A.D.Basa[3] E.Bartocci[4] D.Nickovic[5] R.Grosu[6]

[1,4,6]Vienna University of Technology, Treitlstr.3, Vienna, Austria,
{konstantin.selyunin, ezio.bartocci, radu.grosu}@tuwien.ac.at

[2,3]Infineon Technologies Austria AG, Siemenstrasse 2, Villach, Austria
{thang.nguyen, andrei-daniel.basa}@infineon.com

[5]AIT Austrian Institute of Technology,Tech Gate Vienna, Donau-City-Strasse 1, Vienna, Austria
dejan.nickovic@ait.ac.at

Abstract

Runtime monitoring is an important technique for catching failures. This work shows how to synthesize hardware runtime monitors using High-Level Synthesis to check system requirements that are formalized and expressed in Signal Temporal Logic. We describe our flow starting from a natural language requirement to hardware implementation. As a case study, we apply our flow to monitor a mission-critical property of a missile launch.

I. Introduction

Today's embedded systems are integrated and connected with other systems. Interconnection with the physical world and other systems forms cyber-physical systems with both digital and analog components. If the components of such a system are safety critical (e.g. airbag system or electronic power system in automotive domain), correct system integration is of prior importance.

Verification and validation of such safety-critical systems poses major challenges for the industrial community: simulation and manual testing, which are the prevalent state-of-the-art practices are time consuming, ad-hoc and prone to human errors. Moreover, these techniques might not be applicable in case of analog parts because of simulation time and resource consumption. We see the application of formal methods as a promising direction to improve and speed up state-of-the-art practice in verification and validation.

Runtime monitoring [1] is a formal technique that checks if a property of interest holds during the current run of a system. Signal Temporal Logic (STL) [2] is a declarative formal language, that we use for formalizing requirements of cyber-physical systems. We propose to use STL as a specification language for building runtime monitors. This allows to check if runs of an emulated system (or the implementation of the system) satisfy the pre-defined mission-critical properties. To achieve this we present the flow from a natural language requirement to implementation of a monitor.

Although our target application is checking a protocol in the automotive domain, for illustrative purposes we consider a case study which represent easy-to-grasp, illustrative example of a mission-critical property of mutually related timing properties. In this work we are concerned with runtime monitors that work in parallel with the DUT. To keep up with the speed of the DUT, our goal is to synthesize monitors in the same hardware.

The key paper contributions can be summarized as follows:

- A study on how to synthesize STL specifications into hardware;
- Propose a complete flow from natural language requirement to synthesizable HDL code using STL for high-level specification and HLS hardware monitor generation;
- Demonstrated a successful application of the proposed flow to enabling runtime monitor of mission-critical properties example on a cost-effective hardware platform Zedboard.

II. Related work

Temporal Logic (TL), introduced by Pnueli in 1977, is a formal language to reason about sequences of events on a time axis. Since then, variants of TL have been successfully applied for reasoning about reactive systems [3], verification of programs [4], in hardware design [5], and in design and verification of cyber-physical systems [6]. In the research community TL has attracted significant attention, resulting in various extensions that made it more expressive. Metric Temporal Logic (MTL) [7] extends LTL by adding bounds on temporal operators. This allows one to restrict evaluation of temporal operators to bounded intervals and use MTL as a specification language for defining timing properties of digital systems [8]. Metric Interval Temporal Logic [9] is a restriction of MTL which does not allow punctual intervals (i.e. containing only one time point: the form $[a, a]$ or $\{a\}$) and benefits from this restriction by having an efficient decision procedure.

Signal Temporal Logic [2] is an extension of MTL or MITL which allows reasoning about mixed analog digital systems. STL handles the analog part in the logic by comparing analog variables with thresholds. The authors in [10] successfully used STL for formalizing discovery mode of a Distributed System Interface (DSI3), in [11] the authors generate hardware monitors for checking STL properties.

Runtime monitoring (or runtime verification [1]) is a state-of-the-art technique to ensure system-specification conformance. The ability to derive a verdict from observing the system's outputs without having its model, relatively low overhead fosters interest to runtime monitoring both from theoretical [12,13] and practical points of view [14,15]. Verifying properties at runtime also allows to avoid logging and post-processing large amounts of data which can significantly speed up finding bugs.

III. Background

In this section we a give a concise overview of STL, which is used as a specification language for formalizing mission-critical requirements. We consider a general formulation with both past and future temporal operators and standard qualitative semantics. We then succinctly review how to integrate runtime verification with a DUT.

A. Signal Temporal Logic

The syntax of an STL formula φ with past and future operators over a set of boolean variables $P = \{p_1, \cdots, p_m\}$ and real-valued variables $X = \{x_1, \cdots, x_n\}$ is defined by the following grammar [10]:

$$\varphi := p \,|\, x \sim c \,|\, \neg\varphi \,|\, \varphi_1 \vee \varphi_2 \,|\, \varphi_1 \mathcal{U}_I \varphi_2 \,|\, \varphi_1 \mathcal{S}_I \varphi_2, \tag{1}$$

where $p \in P$, $x \in X$; $c \in \mathbb{Q}$ is a constant; $\sim$ is a binary relation of the form: $\{\leq, <, =, >, \geq\}$; interval I is of the form $[a, b]$, where $a, b \in \mathbb{N}$ and $0 \leq a \leq b$. An STL specification φ is interpreted over a mixed signal w which is a partial function: $w \;:\; \mathcal{T} \to \mathbb{B}^m \times \mathbb{R}^n$, where $\mathcal{T}$ is an interval $[0, T)$ with arbitrary finite value T i.e. a signal is a combination of boolean and real-valued variables that are at most T in length.

The semantics of an STL formula w.r.t. to a signal w at a time point i is defined as follows:

$$\begin{array}{lcl}
(w, i) \models p & \leftrightarrow & p[i] = \top \\
(w, i) \models x \sim c & \leftrightarrow & w_x[i] \sim c \\
(w, i) \models \neg\varphi & \leftrightarrow & (w, i) \not\models \varphi \\
(w, i) \models \varphi_1 \vee \varphi_2 & \leftrightarrow & (w, i) \models \varphi_1 \text{ or } (w, i) \models \varphi_2 \\
(w, i) \models \varphi_1 \mathcal{U}_I \varphi_2 & \leftrightarrow & \exists j \in (i + I) \cap \mathbb{T} : (w, j) \models \varphi_2 \\
 & & \text{and } \forall i < k < j, (w, k) \models \varphi_1 \\
(w, i) \models \varphi_1 \mathcal{S}_I \varphi_2 & \leftrightarrow & \exists j \in (i - I) \cap \mathbb{T} : (w, j) \models \varphi_2 \\
 & & \text{and } \forall j < k < i, (w, k) \models \varphi_1,
\end{array} \tag{2}$$

where $w_x[i]$ is x^{th} component of w.

Other STL operators are derived from the definition in a standard way: $\top = \varphi \vee \neg\varphi$; $\bot = \neg\top$; eventually $\Diamond_I \varphi = \top \mathcal{U}_I \varphi$; once $\diamondminus_I \varphi = \top \mathcal{S}_I \varphi$; always $\Box_I \varphi = \neg \Diamond_I \neg\varphi$; historically $\boxminus_I \varphi = \neg \diamondminus_I \neg\varphi$. Temporal operators: eventually, always, once and historically also admit a natural direct definition of their semantics:

$$\begin{aligned}(w,i) \models \Diamond_I \varphi \quad &\leftrightarrow \quad \exists j \in (i+I) \cap \mathbb{T} : (w,j) \models \varphi \\ (w,i) \models \Box_I \varphi \quad &\leftrightarrow \quad \forall j \in (i+I) \cap \mathbb{T}, (w,j) \models \varphi \\ (w,i) \models \diamondminus_I \varphi \quad &\leftrightarrow \quad \exists j \in (i-I) \cap \mathbb{T} : (w,j) \models \varphi \\ (w,i) \models \boxminus_I \varphi \quad &\leftrightarrow \quad \forall j \in (i-I) \cap \mathbb{T}, (w,j) \models \varphi.\end{aligned} \tag{3}$$

Figures 1 and 2 pictorially show definitions of STL future and past temporal operators: at the point t_i on the time axis, we look either forward (future) or backward (past) on the time axis to evaluate corresponding temporal operators.

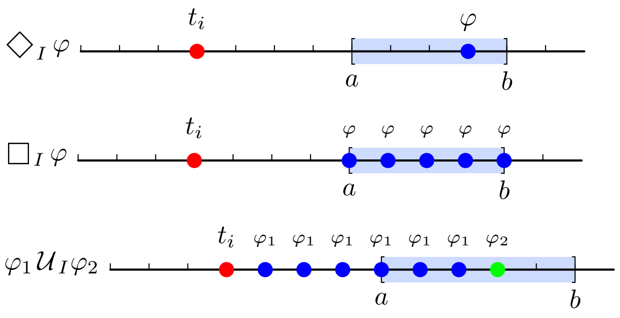


Figure 1: STL future temporal operators

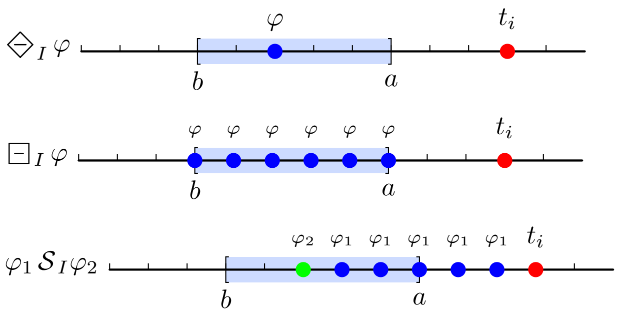


Figure 2: STL past temporal operators

B. *Runtime monitoring*

Apart from the formal verification, which in general tries to answer the question whether all possible runs of a hardware or software system adhere to given correctness properties, runtime monitoring [1] tackles a more modest problem: *"Does a current execution meet a correctness property?"*. Stating the problem that way allows one to ensure that actual implementation (apart from the model) satisfies the correctness properties and employ runtime monitoring as a redundancy mechanism in safety-critical systems.

Figure 3 shows a general way of integrating runtime monitoring with an existing system: based on the observations from the DUT (Hardware/Software System), a monitor delivers a verdict if a predefined specification is satisfied/violated. If the monitor is intrusive, it might perform controlling actions on the observed system upon specification violation (dashed line). In this work we consider only non intrusive monitoring (i.e. without influence on the DUT).

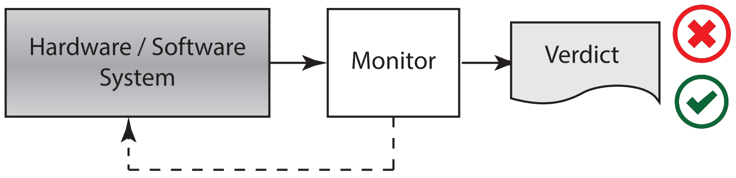


Figure 3: Runtime monitoring concept

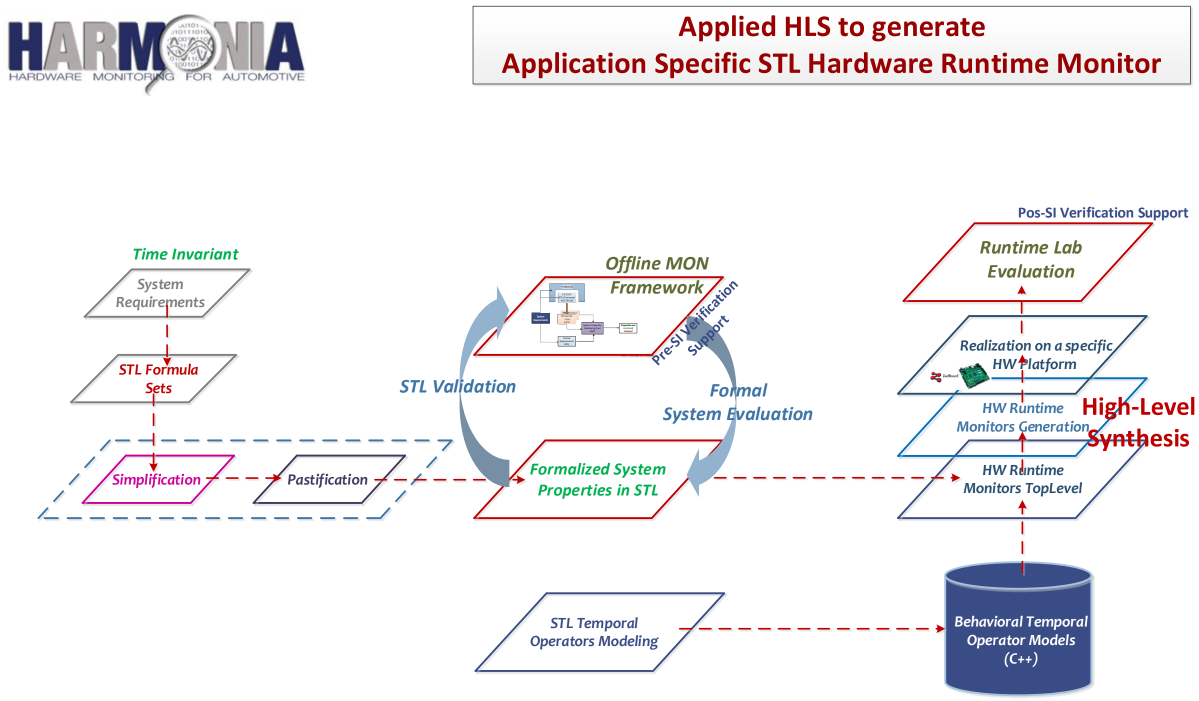
IV. Problem formulation

In this paper we study the problem of STL runtime monitoring. Given a system requirement expressed in a natural language, our task is to create a complete flow which results in a runtime monitor checking if the system requirement has been fulfilled. Our goal is to explore applicability of a general purpose programming language such as `C++` for creating STL runtime monitors: we study how to use High-Level Synthesis for generating hardware runtime monitors. We aim to obtain both software simulations and a synthesizable HDL code that can be run on a low-cost hardware (ZedBoard). `C++` implementation of STL operators fosters reuse and will allow us to employ full capabilities of a Zynq chip by splitting a monitor in hardware and software components.

V. Hardware Monitor Generation Flow

In this section we describe STL runtime monitor generation flow: we start by giving a high-level picture of our approach and what steps we perform to get the synthesizable STL monitors; we then elaborate on every step in detail.

Figure 4 shows the *complete flow* from natural language requirements to a synthesizable HDL code using STL for high-level specification and HLS hardware monitor generation. First, time invariant system requirements must be formalized to obtain a set of STL formulae which are used for hardware monitor generation. The formulae to be hardware-synthesizable need to be converted in a specific format (containing only past temporal operators i.e. pastified) and simplified. We use these formulae after the conversion step to simulate and check if the requirement has been captured correctly. The off-line simulation provides us with additional guarantee of an STL property being correct. Each STL temporal operator has a corresponding `C++` implementation, which is then used to construct an STL monitor from the previously obtained formula: the monitor comprised of STL temporal operators with appropriate time bounds connected in accordance with the parse tree of the formula. We obtain synthesizable IPs for each temporal operator using High-Level Synthesis.



HARMONIA
HARDWARE MONITORING FOR AUTOMOTIVE
Applied HLS to generate
Application Specific STL Hardware Runtime Monitor
Time Invariant
System Requirements
STL Formula Sets
Simplification
Pastification
Offline MON Framework
Pre-SI Verification Support
STL Validation
Formal System Evaluation
Formalized System Properties in STL
STL Temporal Operators Modeling
Pos-SI Verification Support
Runtime Lab Evaluation
Realization on a specific HW Platform
HW Runtime Monitors Generation
High-Level Synthesis
HW Runtime Monitors TopLevel
Behavioral Temporal Operator Models (C++)

Figure 4: Hardware Runtime STL Monitor Generation Flow using High-Level Synthesis at Infineon Technologies Austria AG

A. System Requirements

In the first step, natural-language requirements must be converted to STL formulae to eliminate ambiguity and possible misinterpretation. It is a non-trivial problem and still an open issue [16] how to express mutual dependencies and temporal behavior and translate it in a formal language. We illustrate how the conversion is done in a case study in Section VI.

B. STL Formula Sets

A set of STL formulae represents requirements in a formal way, that will be implemented in hardware. The conversion from System requirements to STL formulae must be done by an expert and at this time cannot be fully automated: an expert must understand the requirement and then devise a corresponding formula. To facilitate this procedure one can restrict the way system requirements can be stated in natural language and provide templates for common cases.

C. Pastification & Simplification

As seen from Figures 1 and 2, to evaluate STL temporal operators at a point t_i we need to look either forward (future) or backward (past) on a time axis. Clearly, we cannot evaluate a formula with future STL temporal operators at a time point t_i, since we do not know what will happen in future. Instead, we convert the formula containing future STL temporal operators to an equisatisfiable formula which contains only past operators and postpone a verdict of a monitor until all required information is available. We call this conversion pastification. The authors in [11, 17] proposed pastification rules that we use in our flow. Every bounded future STL formula can be pastified.

Pastification of a bounded future STL formula φ is done in the following steps:

1. find a temporal depth $D(\varphi)$ of the formula (Fig. 5). Temporal depth is a minimal time window that we need to buffer in our implementation to reason about the satisfaction of φ.

2. set displacement $d \geq D(\varphi)$. The displacement is used in the next step of conversion and represents a number of time steps that we need to look behind to evaluate a formula from a current time step.

3. apply the conversion from Fig. 6 recursively.

$$\begin{aligned}
D(p) &= 0 \\
D(\neg\varphi) &= D(\varphi) \\
D(\varphi_1 \wedge \varphi_2) &= max\{D(\varphi_1), D(\varphi_2)\} \\
D(\varphi_1 \rightarrow \varphi_2) &= max\{D(\varphi_1), D(\varphi_2)\} \\
D(\bigcirc\, \varphi) &= 1 + D(\varphi) \\
D(\varphi_1 \mathcal{U}_{[a,b]} \varphi_2) &= b + max\{D(\varphi_1), D(\varphi_2)\} \\
D(\Box_{[a,b]}\, \varphi) &= b + D(\varphi) \\
D(\Diamond_{[a,b]}\, \varphi) &= b + D(\varphi)
\end{aligned}$$

Figure 5: Computing Temporal Depth

$$\begin{aligned}
\Pi(p, d) &= \diamondminus_{\{d\}}\, p \\
\Pi(\neg\varphi, d) &= \neg\Pi(\varphi, d) \\
\Pi(\varphi_1 \wedge \varphi_2, d) &= \Pi(\varphi_1, d) \wedge \Pi(\varphi_2, d) \\
\Pi(\varphi_1 \rightarrow \varphi_2, d) &= \Pi(\varphi_1, d) \rightarrow \Pi(\varphi_2, d) \\
\Pi(\bigcirc\, \varphi, d) &= \Pi(\varphi, d-1) \\
\Pi(\Diamond_{[a,b]}\, \varphi, d) &= \diamondminus_{[0,b-a]}\, \Pi(\varphi, d-b) \\
\Pi(\Box_{[a,b]}\, \varphi, d) &= \boxminus_{[0,b-a]}\, \Pi(\varphi, d-b) \\
\Pi(\varphi_1 \mathcal{U}_{[a,b]} \varphi_2, d) &\leftrightarrow \bigvee_{i=1}^{z} \Pi(\varphi_1^i \mathcal{U}_{[a,b]} \varphi_2, d) \\
&\leftrightarrow \bigvee_{i=1}^{z} \Pi(\bigcirc\, \varphi_1^i \wedge \\
&\quad \Diamond_{[a-1,b-1]}(\varphi_1^i \wedge \bigcirc\, \varphi_2), d)
\end{aligned}$$

Figure 6: Pastification Conversion

D. Temporal Operator Primitives

STL temporal operators are implemented in `C++`, which allows us to use this implementation both for off-line simulation and hardware implementation. Typical STL specification contains of different temporal operators, that can be nested and binded with logical connectives. A runtime monitor for such a specification is constructed from temporal operator primitives, instantiated with appropriate time bounds and connected according to the parse tree of the formula.

E. Off-line Monitoring Framework

We use simulations to test our software implementation. In off-line monitoring we are able to generate a trace in software, supply it to a monitor and log results. We use `matplotlib` to visualize results from a log file. This allows us to raise our confidence that (i) monitor implementation is bug-free (ii) an STL property is stated correctly.

F. High-Level Synthesis

We use High-Level Synthesis from Xilinx [18] to synthesize STL temporal primitives as an IP. We use hardware-specific data types for utilizing hardware resources efficiently. For each temporal operator: (i) perform a test whether an implementation with hardware-specific data types correspond to one with a generic software types; (ii) synthesize HDL code using Vivado HLS; (iii) test if synthesized code and software generate the same results; (iv) export the IP.

VI. Case Study and Experimental Result

As a case study we build a hardware monitor for a safety property that describes the launch of a missile from a ship. Suppose that the launch is governed by three signals: "launch enable": ℓ, "fire enable": f and "detonation": d. The signal ℓ characterizes whether the missile is allowed to be launched; the signal f describes the moment when a missile has been fired; the d signal is sent to trigger detonation.

The property we monitor is a safety property of the ship in a sense that it describes correct order of actions, timing and absence of damage to the ship from its missile:

"When the missile received the launch enable signal, it must see the fire enable signal followed within the next four time points. After fire en has arrived, no detonation is allowed for the next five time points." Fig. 7 graphically represents the specification.

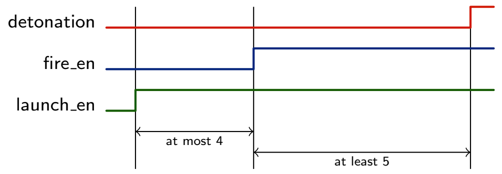


Figure 7: Missile timing property specification

We now cover the translation step from the requirement in a natural language to STL. Receiving ℓ, f, and d signals is equivalent to their rising edges (Fig. 7). We denote these rising edges with "$\uparrow$" in the Equation 4 where $\uparrow p$ is a shortcut for an STL formula $p \wedge \ominus \neg p$. The relation that the arrival of ℓ signal must follow within four time steps by $\uparrow f$ and subsequent absence of detonation, is by definition bounded eventually operator in STL (see Fig. 1); the absence of $\uparrow d$ signal for five time points is a bounded always of a negated d signal:

$$\uparrow \ell \rightarrow \Diamond_{[0;4]} \left(\uparrow f \wedge \Box_{[0;5]} \neg d \right). \tag{4}$$

We then convert the bounded future STL formula (Eq. 4) to an equisatisfiable past one. After performing pastification (Fig. 6) step we obtain the following past STL specification:

$$\diamondminus_{\{9\}} \uparrow \ell \rightarrow \diamondminus_{[0;4]} \left(\diamondminus_{\{5\}} \uparrow f \wedge \boxminus_{[0;5]} \neg d \right) \tag{5}$$

In the next step we construct a circuit which represents a monitor. Each sub-formula from Eq. 5 is converted to a corresponding STL temporal primitive. After composing a circuit we simulate it using our C++ implementation. Fig. 8 shows simulation results which are interpreted as follows: each blue dot represents a satisfaction of a corresponding input or sub-formula. We model several scenarios of receiving ℓ, l and f signals. The topmost signal represents an output of a monitor, where the absence of a dot in a time stamp corresponds to specification violation: the second scenario corresponds to d appearing too early, the fourth scenario corresponds to f appearing too late.

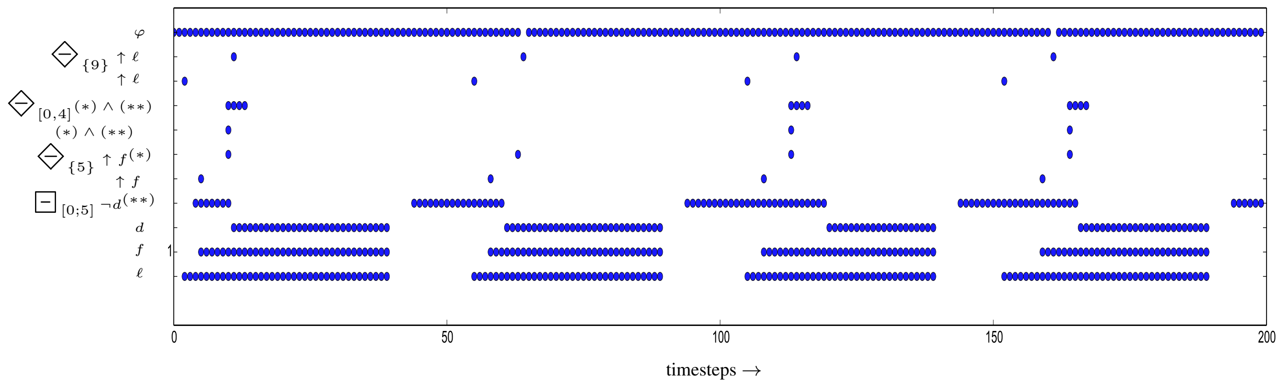


Figure 8: Missile monitor: software simulation results

We use Zedboard from Xilinx and Vivado System Edition [19] (University License) for generating runtime monitors (see Fig. 9). During HLS conversion we leveraged hardware specific data types to assign all the flags in the model exactly one bit-width and reduced native data types proportional to the width of Xilinx multipliers to save hardware resources. Each operator has been synthesized separately to foster reuse for other specifications. We then implemented the demonstrator on FPGA which generates ℓ, f and d signals and checks the monitor under nominal conditions and fault injections.

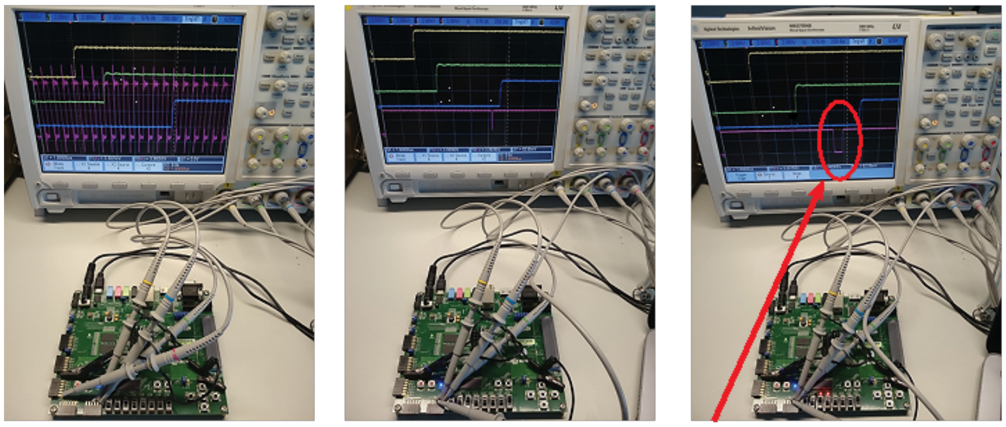


Figure 9: Experimental setup (see [20] for oscillograms)

VII. Conclusion

In this paper we showed how to use STL and High-Level Synthesis for generation of hardware runtime monitors. We proposed a flow from system level requirements to synthesizable hardware monitors. We demonstrated our approach on a case study where we build a hardware runtime monitor for a missile launch from natural language requirement. In future we plan to investigate advanced functionality such as splitting functionality of a monitor between CPU and hardware as well as building a unified framework based on STL and HLS for both qualitative and quantitative reasoning about mixed-signals.

Acknowledgment

This research is supported by the project HARMONIA (845631), funded by a national Austrian grant from FFG (Österreichische Forschungsförderungsgesellschaft) under the program IKT der Zukunft.

References

[1] Martin Leucker. Teaching Runtime Verification. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 34–48. Springer Berlin Heidelberg, 2012.

[2] Alexandre Donz, Oded Maler, Ezio Bartocci, Dejan Nickovic, Radu Grosu, and Scott Smolka. On temporal logic and signal processing. In Supratik Chakraborty and Madhavan Mukund, editors, *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, pages 92–106. Springer Berlin Heidelberg, 2012.

[3] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[5] G.V. Bochmann. Hardware specification with temporal logic: An example. *Computers, IEEE Transactions on*, C-31(3):223–231, March 1982.

[6] Rajeev Alur. *Principles of Cyber-Physical Systems*. MIT Press, 2015.

[7] Dejan Nickovic and Nir Piterman. From mtl to deterministic timed automata. In Krishnendu Chatterjee and ThomasA. Henzinger, editors, *Formal Modeling and Analysis of Timed Systems*, volume 6246 of *Lecture Notes in Computer Science*, pages 152–167. Springer Berlin Heidelberg, 2010.

[8] Thomas Reinbacher, Matthias Fgger, and Jrg Brauer. Runtime verification of embedded real-time systems. *Formal Methods in System Design*, 44(3):203–239, 2014.

[9] Oded Maler, Dejan Nickovic, and Amir Pnueli. From MITL to Timed Automata. In Eugene Asarin and Patricia Bouyer, editors, *Formal Modeling and Analysis of Timed Systems*, volume 4202 of *Lecture Notes in Computer Science*, pages 274–289. Springer Berlin Heidelberg, 2006.

[10] Thang Nguyen and Dejan Nikovi. Assertion-based monitoring in practice checking correctness of an automotive sensor interface. In Frdric Lang and Francesco Flammini, editors, *Formal Methods for Industrial Critical Systems*, volume 8718 of *Lecture Notes in Computer Science*, pages 16–32. Springer International Publishing, 2014.

[11] Stefan Jakšić, Ezio Bartocci, Radu Grosu, Reinhard Kloibhofer, Thang Nguyen, and Dejan Ničković. From Signal Temporal Logic to FPGA Monitors. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*, pages 218–227, 2015.

[12] ScottD. Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, ScottA. Smolka, and Erez Zadok. Runtime Verification with State Estimation. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, volume 7186 of *Lecture Notes in Computer Science*, pages 193–207. Springer Berlin Heidelberg, 2012.

[13] Kenan Kalajdzic, Ezio Bartocci, ScottA. Smolka, ScottD. Stoller, and Radu Grosu. Runtime Verification with Particle Filtering. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification*, volume 8174 of *Lecture Notes in Computer Science*, pages 149–166. Springer Berlin Heidelberg, 2013.

[14] C. Watterson and D. Heffernan. A runtime verification monitoring approach for embedded industrial controllers. In *Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on*, pages 2016–2021, June 2008.

[15] D. Borrione, Miao Liu, K. Morin-Allory, P. Ostier, and L. Fesquet. On-line assertion-based verification with proven correct monitors. In *Information and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on*, pages 125–143, Dec 2005.

[16] Alessandro Cimatti, Marco Roveri, Angelo Susi, and Stefano Tonetta. Formalization and validation of safety-critical requirements. In *Proceedings FM-09 Workshop on Formal Methods for Aerospace, FMA 2009, Eindhoven, The Netherlands, 3rd November 2009.*, pages 68–75, 2009.

[17] Oded Maler, Dejan Nickovic, and Amir Pnueli. On synthesizing controllers from bounded-response properties. In *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 2007.

[18] Vivado High-Level Synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html *(Accessed 05.01.2016).*

[19] Vivado System Edition. http://www.xilinx.com/products/design-tools/vivado.html *(Accessed 05.01.2016).*

[20] Oscillograms and Lab Results. https://www.dropbox.com/sh/ s3jy1zux5y9uvk5/AAC9KOAWFzYSAXOJntR6CosIa?dl=0 *(Accessed 05.01.2016).*