Die approbierte Originalversion dieser Dissertation ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

The approved original version of this thesis is available at the main library of the Vienna University of Technology.

http://www.ub.tuwien.ac.at/eng

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Neural Models For Monitoring and Control

with Applications in Automotive Domain

PhD THESIS

submitted in partial fulfillment of the requirements for the degree of

Doctor of Technical Sciences

within the

Vienna PhD School of Informatics

by

Dipl.Ing. Konstantin Selyunin
Registration Number 01228206

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.rer.nat. Radu Grosu
Second advisor: Asst.-Prof. Dr. Ezio Bartocci
Industrial Co-advisor: Dr. Thang Nguyen

External reviewers:
Prof. Dr. Martin Leucker. Universität zu Lübeck, Germany.
Assc. Prof. Ph.D. Yliés Falcone. Univ. Grenoble Alpes, France.

Vienna, 9th October, 2017

Konstantin Selyunin
Radu Grosu

Declaration of Authorship

Dipl.Ing. Konstantin Selyunin
1040 Wien, Treitlstr. 3, CPS Group

I hereby declare that I have written this Doctoral Thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

Vienna, 9th October, 2017

Konstantin Selyunin

Acknowledgements

This thesis represents the accomplishments of my academic-research career at the Cyber-Physical-Systems Group of the TU Wien. Looking back to the past five years of my PhD Study with the Vienna PhD School of Informatics and the CPS Group, I see this time as a fantastic experience which broadened and deepened my expertise in cyber-physical systems, robotics, neural models, runtime monitoring, and in computer science in general.

I would like to thank Prof. Hannes Werthner, Prof. Hans Tompits, Prof. Andreas Steininger and Ms. Clarissa Schmid for organizing and smoothly running the “Vienna PhD School of Informatics”, a program that gave me the opportunity to conduct the research at the Faculty of Informatics of the TU Wien. During my studies Ms. Clarissa Schmid and Ms. Gerda Belkhofer helped me in organizational and administrative issues. I would like to equally express my gratitude to Prof. Radu Grosu and Dr. Ezio Bartocci that agreed to work with me, accepted me at the Cyber-Physical Systems Group, and guided me through my research and studies. I am also thankful to Dr. Thang Nguyen who became my industrial co-advisor, and gave his valuable advice and guidance to shape my work in a way to be applicable in automotive electronic industry.

Furthermore, I thank all my colleagues from the Cyber-Physical-Systems Group and the PhD School for the fruitful discussions, support and collaboration. Notably, I appreciate Denise Ratasich, Alena Rodionova, Stefan Jaksic, Anna Lukina, Haris Isakovich, Bernhard Frömel, Christian Hirsch, and Aysylu Gabdulkhakova for their help in shaping my research and inspiring discussions that contributed to the results presented in this thesis.

I am also grateful for the financial support from the Vienna PhD School of Informatics and the project HARMONIA (845631), funded by a national Austrian grant from FFG (Österreichische Forschungsförderungsgesellschaft) under the program IKT der Zukunft.

My special thanks are devoted to my family and friends, in particular to Lyudmila Selyunina, Ekaterina Lashmanova, Rostislav Zabolotnyi, Maryna Kostikova, Ivan Zykov, and Roland Mayr who helped me to overcome difficulties and supported me emotionally, shared with me adventures, organized vacations, and helped to recharge.

Abstract

Cyber-physical systems (CPS), which incorporate physical as well as computational components, are a grand challenge of academia and industry in terms of their development, verification, and maintenance. In order for CPS to serve their purpose and ultimately make human lives safer, easier, more enjoyable, and convenient, both academia and industry needs to develop new methods for control and monitoring of such systems. Neural models are a very promising and far looking direction for the design of CPS controllers and monitors. In this thesis we first show how neural models can be applied in CPS control to quantify the uncertainty of the system. We then present how digital spiking neural model, called TrueNorth, can be used in the runtime monitoring of temporal-logic specifications for mission-critical systems. In order to be able to deliver not only a qualitative verdict, but also to reason in a quantitative way, we propose an approach for modeling arithmetic-functions with spiking neurones, and implement neural monitors for (signal) temporal logic specifications based on circular convolution.

In the applied part of the thesis we demonstrate how runtime monitoring can speed up the verification and validation phases in automotive electronic development. We identify phases where runtime monitoring can facilitate both pre- and post-silicon verification and testing. To build runtime monitors that are capable of keeping up with the speed of the physical sensors, we developed an approach to convert formalized requirements to hardware monitors, which are then synthesized in an FPGA. The results of this work enable long-term requirements evaluation and foster reuse of the monitors from pre- to post-silicon verification phases using high-level synthesis. We illustrate our approach by formalizing, creating hardware monitors, and evaluating the results in the lab environment for electrical and timing requirements of the industrial SENT and SPC protocols.

Contents

Abstract **vii**

Contents **ix**

List of Figures **xi**

List of Tables **xiv**

Thesis Publications **xvi**

1 Introduction **1**
1.1 Motivation 2
1.2 Problem Statement 3
1.3 Research Questions 4
1.4 Scientific Contributions 5
1.5 Structure of the Work 7

2 Background **11**
2.1 Neural Models 11
2.2 Runtime Verification 15
2.3 Automotive Electronic Development 18

3 State of the Art Analysis **21**
3.1 Related Work on CPS Control with Neural Models 21
3.2 Related Work on Neural Models in Monitoring 23
3.3 Related Work on Runtime Verification in Automotive 25
3.4 Related Work on Protocol Verification 27

4 Neural Models for Control & Quantifying Uncertainty **31**
4.1 Preliminaries 33
4.2 Key Components of Neural Programs 34
4.3 Bayesian-Network Learning 38
4.4 Case studies 41
4.5 Summary 49

5 Neural Models for Qualitative Monitoring 51
5.1 Qualitative Monitoring with the TrueNorth model 52
5.2 Neural Temporal Testers 55
5.3 Case Study and Experimental Results 59
5.4 Summary 61

6 Neural Models for Quantitative Monitoring 63
6.1 Neuron and Synapse Modeling 64
6.2 Computations with Neural Models 66
6.3 Neural filters as temporal logic monitors 70
6.4 Summary 73

7 Runtime Monitoring in Automotive Electronic Development 75
7.1 Use Case 1: Runtime Monitors in Simulation 78
7.2 Use Case 2: Runtime Monitors for Lab Evaluation 78
7.3 Case Study: Automotive Sensor Interface 79
7.4 Summary 80

8 Industrial Case studies and Evaluation 83
8.1 Timed Regular Expressions 84
8.2 Formalization of the SENT and SPC Protocols 85
8.3 Runtime Monitoring with Recovery 92
8.4 Runtime Monitoring of SENT and SPC protocols 95
8.5 Summary 99

9 Conclusions and Future Work 101
9.1 Summary of Contributions 101
9.2 Critical Reflections 102
9.3 Research Questions Revisited 103
9.4 Future Work 104

Glossary 105

Acronyms 107

Bibliography 111

Appendix: SENT/SPC Monitoring 131
Installation Instructions 131
Tutorial 134
SENT/SPC Monitoring: Details 142
Behavioral Code and Test data 147
User Study 149

Curriculum vitæ 151

List of Figures

1.1 Thesis contributions 6
1.2 Outline of the thesis 9

2.1 Biophysical and Artificial neurons: a Bird's Eye View 13
2.2 Runtime Monitoring: System's overview 16
2.3 STL past and future temporal operators 18

3.1 High-level overview of the related work 29

4.1 Left plot: "Soft" (colored lines) and "hard" (bold black line) thresholds; Right plot: PDFs and the quantiles for $x = 1$ and $a = 0$ 36
4.2 Passing RVs through conditions 37
4.3 Gaussian Bayesian Network for parking 39
4.4 Example trajectories for the parking task 39
4.5 Experimental platform: Pioneer Rover 42
4.6 Parking system architecture 43
4.7 Tap withdrawal circuit of C. Elegans 45
4.8 Deterministic simulations of potentials of `AVA` and `AVB` neurons (left); stochastic simulations using `nif` condition (right) 47
4.9 Controller for parallel parking as a neural circuit 48

5.1 Neural Monitor Generation Flow 53
5.2 Two input TrueNorth circuit 56
5.3 Neural Temporal Testers 58
5.4 Missile timing property specification 60
5.5 Missile monitor: simulation results 60
5.6 Experimental setup (see [osc] for oscillograms) 61

6.1 Normalized EPSC in response to pre-synaptic action potentials (V_{pre}) . . 65
6.2 Neural models for computations on spike rates 66
6.3 Addition in the spike rates using the TrueNorth model 67
6.4 Addition of spike rates with the Hodgkin-Huxley model 68
6.5 Constant multiplication and division of the spike rates with the TrueNorth model: $2 \cdot$ blue = green, $\frac{1}{3} \cdot$ blue = red (n_1 – blue; n_2 – green; n_3 – red) . 69

6.6 Constant multiplication of the spike rates with the Hodgkin-Huxley model 69
6.7 Subtraction in the spike rates with the TrueNorth model: blue − green = red (n_1 – blue; n_2 – green; n_3 – red). 70
6.8 Computing min/max using the TrueNorth model: $r_4 = max(r_1, r_2) + r_3$, and $r_5 = min(r_1, r_2) - r_3$ 70
6.9 Circular Convolution using TrueNorth: the Circuit 71
6.10 Computing $\diamondminus_{[a,b]}$ using TrueNorth 72
6.11 Computing $\boxminus_{[a,b]}$ using TrueNorth 73
6.12 Computing $p\,\mathcal{S}_{[a,b]}\,q$ using TrueNorth 74

7.1 Electronic power steering application 76
7.2 Electronic Throttle Control for Engine Management 77
7.3 Runtime Monitoring Generation Flow 78
7.4 The runtime monitor in simulation: setup and results 80
7.5 Runtime Monitor in Hardware: Lab Setup 81
7.6 Runtime Monitor in Hardware: Chip Scope Results 82

8.1 A SENT frame starts with a synchronisation pulse (SYNC), followed by a status nibble (ST), data nibbles (D1, D2, D3), rolling counters (RC1, RC2), bit inverse of D1 (ND1), cyclic redundancy check (CRC), and finishes with an optional pause. 86
8.2 SENT nibble pulse: A pulse starts (N_{start}) with a falling edge F, followed by a low region L, followed by a rising edge R, followed by a high region H. . 86
8.3 SPC bus mode: Specification signal for two sensors: ID0 and ID3 – communication starts with the trigger pulse, which is followed by a response from a sensor 87
8.4 SPC trigger pulse: A pulse of duration t_{mtr} starts with a low duration t_{mlow}, which encodes the sensor id, followed by a high region 88
8.5 Monitoring an asynchronous serial protocol with recovery^1 93
8.6 Monitor Generation 95
8.7 Runtime Monitoring of the SENT: Hardware Setup 97
8.8 Runtime TRE monitoring: Vivado functional simulation 98
8.9 Runtime monitoring of SPC protocol 99
8.10 Runtime monitoring of the STL requirements 100

1 Installation Directory Structure 132
2 Workspace launcher 133
3 COSIDE Environment 133
4 COSIDE GUI 135
5 Expanded `'sent_spc_python_gui'` project 135
6 Launching the SENT/SPC Monitor GUI 136
7 SENT/SPC Monitor GUI 136
8 Applying `'sent_config_Vdd5500_DUT2.ini'` configuration 137
9 Plotting original sensor data 138

10 Plotting signals from the `monitor_dump.vcd` 139
11 Loading the SPC configuration 141
12 SPC Monitoring Results 142
13 Specifying the parameters of the protocol and defining the frame structure 144
14 Right panel: elements for the monitor generation 145
15 Plot panel 146
16 Bottom panel 146
17 Excerpt from the monitoring report 147
18 Example configuration file 148
19 Behavioral code structure 149
20 Summary of user study results 150

List of Tables

4.1 BN variances and coefficient dependences 45
4.2 Comparison of simulation results 48

5.1 Neural Parameters of Logical Operators 57
5.2 Neural Temporal Testers 59
5.3 Missile monitor: implementation results 61

6.1 Parameters of neuron and synapse model 65
6.2 Parameters of the TrueNorth Model 71

8.1 SENT Requirements in natural language 87
8.2 SPC Requirements in natural language 89
8.3 STL Monitors Generation: FPGA & HLS resources 99

Thesis Publications

The thesis is based on the author's work published in scientific conferences and workshops. For quick-reference and brevity reasons, these core papers, which build the foundation of the thesis, are listed here once and for all, and will not generally be explicitly referenced again. Parts of these papers are contained in the thesis in verbatim.

- **Konstantin Selyunin**, Denise Ratasich, Ezio Bartocci, Md. Ariful Islam, Scott A. Smolka, Radu Grosu: "Neural Programming: Towards Adaptive Control in Cyber-Physical Systems." *In Proceedings of the 54th IEEE Conference on Decision and Control (CDC 2015), Osaka, Japan.* pp.6978-6985, December 15-18, 2015.
- **Konstantin Selyunin**, Thang Nguyen, Ezio Bartocci, Dejan Nickovic, and Radu Grosu: "Monitoring of MTL Specifications With IBM's Spiking-Neuron Model." *In Proceedings of the 19th Design, Automation and Test in Europe Conference and Exhibition (DATE 2016), Dresden, Germany.* pp.924–929, March 14-18, 2016.
- **Konstantin Selyunin**, Thang Nguyen, Ezio Bartocci, Radu Grosu: "Applying Runtime Monitoring for Automotive Electronic Development." *In Proceedings of the 16th International Conference on Runtime Verification (RV 2016), Madrid, Spain.* pp.462-469, September 23-30, 2016.
- Thang Nguyen, Ezio Bartocci, Dejan Nickovic, Radu Grosu, Stefan Jaksic, **Konstantin Selyunin**: "The HARMONIA Project: Hardware Monitoring for Automotive Systems-of-Systems." *In Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications (ISoLA 2016) Corfu, Greece.* pp.371-379, October 10-14, 2016.
- **Konstantin Selyunin**, Thang Nguyen, Andrei Daniel Basa, Ezio Bartocci, Dejan Nickovic, Radu Grosu: "Applying High-Level Synthesis for Synthesizing Hardware Runtime STL Monitors of Mission-Critical Properties." *In Proceedings of the 13th Design and Verification Conference and Exhibition (DVCon 2016), San Jose, CA, USA.* pp.1-8, February 28-March 3, 2016.

- **Konstantin Selyunin**, Ramin M, Hasani, Ezio Bartocci, Radu Grosu: “Computing with Biophysical and Hardware-efficient Neural Models.” *In Proceedings of the 14th International Work-Conference on Artificial Neural Networks (IWANN 2017), Cadiz, Spain.* pp.535-547, June 14-16, 2017.
- **Konstantin Selyunin**, Stefan Jaksic, Thang Nguyen, Christian Reidl, Udo Hafner, Ezio Bartocci, Dejan Nickovic, Radu Grosu: “Runtime Monitoring with Recovery of the SENT Communication Protocol.” *In Proceedings of the 29th International Conference on Computer Aided Verification, (CAV 2017), Heidelberg, Germany.* pp.336-355, July 24-28, 2017.

CHAPTER 1

Introduction

Herbert A. Simon [Sim96] outlined the fundamental difference between the natural sciences and the "sciences of the artificial" as follows: While a natural scientist starts from *the physical world* and tries to build a model of the phenomenon under study, possibly by revising the model, an engineer starts with the *model* and tries to realize the model in the physical world, possibly revising realizations.

Despite of this fundamental difference, and dating back to Leonardo da Vinci's flying machine in the *"Codex on the Flight of Birds"* (c.a. 1505, itself very much inspired by the bat's wings), inspiration from nature greatly affected technological development. Unlike market economy, nature is not under strict time-to-market pressure, and crafts its solutions by millions of years of evolution. This makes its results optimal with respect to energy usage, environmental adaptation, perception, and robustness to failure.

Employing solutions from nature resulted in scientific disciplines such as biomimicry[Ben02] and bionics[NW15]. Their goal is to study the application of ideas from biological systems in the engineering context, with prominent examples such as hair sensors for flow control [GS12], running-dog robots from Boston Dynamics [RBNP08], or self-cleaning nano-materials [GT11] (inspired by the lotus flower). These successes encourage the application of principles mastered by nature in Cyber-Physical Systems (CPS), too.

CPS [BG12] pose grand challenges in terms of managing system complexity, ensuring safety, and developing optimal solutions. The large number of interconnected devices, and the huge amount of data generated by their components, result in considerable obstacles for processing and reasoning about CPS. In this thesis, we are primarily concerned with the *cyber* component of CPS. This has to be reliable, fault tolerant, and satisfy strict safety requirements. Although formal verification [BK08] and model-based design [BKKS12], are in principle capable of providing a formal-correctness proof for a software component of the cyber part, these methods have limitations when applied to real-world industrial problems, due to the sheer complexity of the associated systems.

1.1 Motivation

From the “sciences of the artificial” perspective, it is vital to understand how the principles adopted by nature can be applied in the context of the “design sciences”.

An animal’s brain, which is a very prominent area of investigation[GKNP14, DA05], is an excellent example of the astonishing progress of research: From the early beliefs (e.g. when nerve cells were considered to be pipes transferring the “spirituous liquor”, William Croone, 1667, [Ros32]), to the birth of the neuroscience in 1888 by Ramón y Cajal [She15], and to state-of-the-art methods like the connectomics [Per14], optogenetics [HQB+09] and brain-machine interfaces [CLC+03], we are witnessing nowadays a much deeper understanding of the brain’s function: From the micro scale perspective (e.g. release of neuro-transmitters and change of ionic concentrations during generation of an action potential), to the macro-scale perspective (e.g. modelling the cortical activity of a macaque’s brain [PWD+12]). This is all possible due to the incredible research progress in neuroscience.

These advancements have important implications: Neurons are now seen as plastic devices [Doi07], constantly adapting and changing their function, according to the external stimuli. Starting from the paper of McCulloch & Pitts [MP43], the advances in brain research provide a source of inspiration for different fields of computer science. The concept of a neuron, as a universal computing device that is capable to dynamically adapt to the external changes, is particularly promising for designing solutions for CPS.

As CPS operate in dynamic uncertain environments, where computational components are tightly coupled with the physical world, it is important to develop controllers for CPS that are capable of taking into account the uncertainty of the environment, together with the prior knowledge about the system. From the computational perspective, CPS are regarded as reactive systems [AILS07]. For them, it is vital not only to provide control inputs, but also to provide mechanisms for monitoring properties of interest at runtime, in order to capture and report violations, and to trigger recovery mechanisms.

To leverage neuronal functional/structural plasticity, it is essential to understand how models of neurons (the universal information processing elements in biological systems), can be used from the “sciences of the artificial” perspective, in the context of CPS.

From a practical point of view, it is necessary to cope with the complexity and the sheer amount of data generated by the CPS. For example, in the automotive electronic-development domain, which is responsible for producing the CPS components (consider the angular sensors based on the Hall effect and their associated controllers) it is of utter importance to develop methods that speed up the testing process, and eliminate errors at the early stages of the development cycle. Manufacturers strive for additional safety barriers to capturing violations during product verification phases, aim for automated solutions and reduced workload for an engineer, during the design and testing process.

1.2 Problem Statement

The aforementioned research/applications challenges uncover a broad range of questions that have to be solved. To narrow down the scope of the thesis, we outline the main research problems in this section, and specify a list of research questions in Section 1.3.

From a control perspective, we need to address the decisions that have to be made under uncertainty in CPS. We aim to use neural models that are capable quantify the uncertainty, and give on average "smooth" control decisions.

From a monitoring perspective, it is necessary to develop methods allowing the use of neural circuits as monitors, to check the properties of CPS at runtime. In recent years, we have seen the emergence of new, brain-inspired computer-hardware architectures [SMN11, WHTvS14, CMA$^+$13]. We target the application of state-of-the-art spiking neural-architecture models, for monitoring temporal properties of CPS. We do this for both qualitative and quantitative properties, with the same neural model, by reconfiguring its parameters. This approach leverages the functional plasticity of the neural models.

To cope with the system's complexity, and to provide an additional safety mechanism, it is important in many practical scenarios, to show at runtime, that the system has not violated its safety requirements. Runtime verification [Leu12], a light-weight state-of-the-art verification technique, treats the system under investigation as a black-box, and reports system's conformance to formal requirements in a current run. In the automotive electronic-development domain, it is necessary to speedup the testing process for sensing products. In particular, runtime verification solutions are of great interest, since they are non-intrusive, and in principle, capable to check in real-time, the specification conformance of sensing blocks at various abstraction layers, or collect and report violations.

The main theoretical goal of this thesis, is *to study neural models for the control and monitoring of CPS.* The main practical goal of this thesis is *to build efficient runtime monitors in hardware and software, that are applicable to the state-of-the-art components of CPS, in particular, of the automotive electronic-development domain CPS.*

These research-statement goals imply that the solutions to be developed within the scope of this thesis, should possess the attributes of:

1. *Simplifying controller design in CPS* while taking into account both, the uncertainty about the environment, and prior knowledge about the system;
2. *Unifying qualitative- and quantitative-monitoring techniques* for checking temporal system-properties, by exploiting plasticity of neural models;
3. *Speeding up product testing* by allowing the runtime monitoring of the CPS under investigation, in both hardware and software.

Solutions with the aforementioned attributes are attractive from two perspectives. First, they facilitate the design of CPS controllers as well as CPS monitors by using neural models as a common basis. Second, they allow to introduce additional safety mechanisms, which are not only capable to catch errors (during several phases of product development), but also speed up the testing process of CPS (e.g. angular sensing product).

1.3 Research Questions

The research work that is conducted within the scope of this thesis is mainly aligned around the following research questions:

RQ 1: *How does one design controllers for CPS using neural models?*

The ability of neurons to adapt to an uncertain and dynamically changing environment through structural and functional plasticity [DPM$^+$11], renders their models as a very natural candidate for CPS controller design. CPS controllers also operate in an uncertain and dynamically changing environment, and they also need to adapt.

Efficient designs should consider controllers that are capable to adapt through plasticity (i.e. learning), while taking into account uncertainty and prior experience. It is therefore very important to understand how nature's principles, discovered by neuroscience, can be mapped to CPS controller design. One of them is for example, the availability of a measure, quantifying controller's uncertainty. This can be used whenever deciding which action to take. From a practical point of view, the measure has to be representable through programming constructs, that are familiar to software engineers.

Summarizing, the aim of this work is to develop adaptive CPS controllers inspired by neural models. These should quantify their uncertainty when taking decisions. Moreover, they should aim towards smooth average decisions, thus increasing their robustness.

RQ 2: *How does one perform qualitative and quantitative monitoring of temporal CPS properties, at runtime, when using neural models?*

The evolution of CPS observable outputs as well as the one of CPS inferred states, must satisfy the CPS requirements specification (RS). Deviations from these RS might manifest errors in the system, which could lead to failures with possibly catastrophic consequences (e.g. if the CPS is safety-critical [Kop11]). Monitoring the evolution of the CPS in real-time, allows to catch important deviations from the RS early on, and opens the possibility to bringing the CPS back to a safe state. Temporal logics [Lon89] allow to formally define an RS, as the set of all permissible CPS evolutions in time. Thus, they provide a formal notation for RS compliance.

CPS with mixed-critical properties [BD13] require not only a qualitative "pass-fail" verdict, but also a quantitative verdict, measuring the amount of satisfaction/violation of the RS. This could signify, for instance, a level of service degradation, for non-safety-critical requirements. Quantitative interpretations of the RS assess the distance from satisfaction/violation. This can be viewed as an additional robustness metric.

The use of neural models for monitoring CPS allows to unify the qualitative and quantitative measurement of temporal logic properties. Moreover, neurons as universal atomic components for building monitors, allow to apply both functional and structural changes, in order to configure the best circuit producing the monitoring verdict.

RQ 3: *How does one speed up testing within automotive electronic development?*

From the industrial point of view, e.g. for Infineon [Inf17], which is an automotive Tier 2 supplier, it is important to reduce CPS testing time and engineering effort. Developing solutions that meet such needs, by introducing safety mechanisms capable of capturing errors in different phases of the design process, is of utmost importance. They allow CPS suppliers to meet strict safety standards (e.g. ISO 26262 [iso11]).

Although testing is an accepted industrial practice, it involves substantial manual work. Moreover, it is by no means complete, as it is mainly performed over short fragments of recorded data. Together with post-silicon verification, testing accounts up to 60-70% of total development time [NN16]. Shortening this while also increasing coverage is crucial.

Since monitors are capable to check the requirements at the system's runtime, they bring substantial potential for reducing the amount of data stored. They help identifying the relevant portions of data to be saved (as opposed to post-processing of the recorded data). Runtime monitoring also provides a way for checking classes of requirements at different abstraction layers. Hence, it is necessary to identify how runtime monitoring can be incorporated in the existing design practice in the automotive electronic industry.

RQ 4: *What are the necessary steps to building efficient runtime monitors in hardware that are applicable in the industrial state-of-the-art practice?*

For runtime monitoring to be accepted in the existing industrial practice, it needs to be expressive enough, in order to formalize the RS of real products, provide sufficient performance to keep up with the speed of monitoring systems (enable real-time data processing), facilitate easy reconfiguration, and present monitoring results in a comprehensive way.

Current industrial practice relies on RSs that are written using natural language and graphics. An engineer has to interpret these requirements, and develop the corresponding tests. In order to eliminate ambiguities in RS interpretation, it is possible to specify the temporal evolution of a system in a suitable variant of temporal logic, which as an added benefit, allows the automatic translation of the RS to the monitoring code.

It is very important to develop a methodology enabling the translation of formal RSs to hardware-monitor implementations, capable of giving their verdicts in real time.

1.4 Scientific Contributions

This section gives more details about the scientific contributions of this thesis. Figure 1.1 is a “birds-eye-view” of the work, which is then elaborated in more detail.

Contribution 1: We develped a design method for adaptive CPS control, which is based on either artificial or biological neural models. We show the

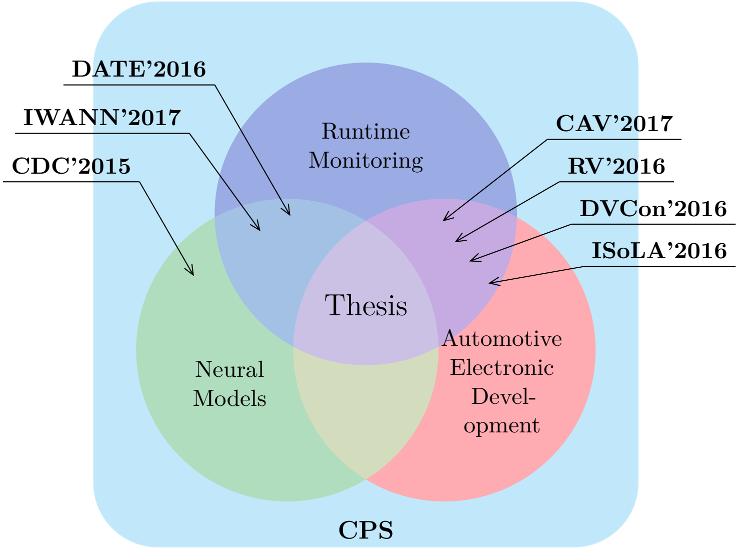


Figure 1.1: Thesis contributions

utility of (neural) programs which contain branching statements that are capable to incorporating and quantifying uncertainty. The method consists of two steps. In the first step, one writes a program (a controller) skeleton, whose sigmoidal stochastic branching statements leave their mean and variance unspecified. In the second step, one learns the proper means and their associated variances in a supervised fashion, from (good) teacher's traces.

This adaptive-controller design method has been validated, by developing and testing a parallel-parking controller for a Pinoneer 3-AT rover, which is available in our CPS laboratory. We used artificial neural-networks models for the learning purpose, as well as biological neural-networks models based on the C.elegans nematode's neurons.

Details on applying neural models in the design of adaptive CPS controllers are discussed in Chapter 4. Contribution 1 has been published in [SRB$^+$15].

Contribution 2: We developed both qualitative and quantitative techniques for CPS monitoring, based on spiking-neurons networks models. Spiking neural networks [Vre03], which were pioneered by the analysis of the action potential of the giant squid by Hodgkin and Huxley [HH52], have now became the classical biophysical model of a neuron. A digital spiking neural model, as proposed by [CMA$^+$13], allows efficient realizations of neural circuits in the digital state-of-the-art hardware.

We developed a method that allows to create a neural circuit and configure its neurons to perform runtime monitoring of temporal logic properties of analog signals. The circuit receives the input signal and outputs the monitoring verdict for a temporal logic specification. For quantitative evaluation we show how to perform fundamental arithmetic operations over spike rates using neural models. These operations are the necessary components for computing quantitative semantics of a temporal logic specification at runtime. If the entire (or sufficiently large portions of the) signals are given in advance, we show how a temporal logic semantics, defined using convolution, can be utilized for constructing neural circuits that are capable to compute the monitoring verdict.

Details on qualitative monitoring using spiking neurons are discussed in Chapter 5. Quantitative monitoring with spiking neurons are presented in Chapter 6. The results of Contribution 2 have been published in [SNB$^+$16a, SHR$^+$17].

Contribution 3: We applied and validated our runtime verification techniques in the automotive electronic-development domain. As discussed in Section 1.3, it is very important to develop methods that are capable to speed up CPS testing. We analyzed the design and production phases of an automotive sensing component, showed how runtime verification can be applied, and what benefits and limitations it brings. Our results show that during the testing process, runtime monitoring can be applied in the pre-silicon concept-design phase, as well as in the post-silicon verification phase, to check the requirements conformance of the manufactured sensing component.

The non-intrusive nature of runtime monitoring allows one to use it as an additional safety mechanism and integrate it into existing design practice. Details on applying runtime monitoring for automotive electronic development are discussed in Chapter 7. The results of Contribution 3 have been presented in [SNBG16, NBN$^+$16].

Contribution 4: We formalized the industrial-strength protocols Single Edge Nibble Transmission (SENT) and Short PWM Code (SPC), automatically generated monitors out of this formalization, and checked the conformance with these protocols of an Infineon's angular-sensing component at runtime. To show the applicability of the results on the industry-relevant case study, we illustrate the process of runtime monitoring requirements for the SENT and SPC protocols.

For these protocols we implemented the full flow from the analysis of existing documentation, formalization of requirements classes, creating simulation environment and violation scenarios, software/hardware implementation of runtime monitors and testing monitors on the real test data in industrial environment. For a subclass of asynchronous serial protocols we also defined a procedure to construct runtime monitors that are capable (i) to recover after violations, (ii) collect and report error types to the user.

Electrical and timing requirements of SENT and SPC protocols were formalized using two formalisms and compared in terms of resources, speed and conciseness of the formalized specifications. Details on runtime monitoring of the SENT and SPC protocols are discussed in Chapter 8. Contribution 4 was published in [SJN$^+$17, SNB$^+$16b].

1.5 Structure of the Work

The remainder of the thesis is organized as follows (see also Figure 1.2):

- Chapter 2 provides background information, and introduces the terminology and concepts used throughout the thesis. In particular, it reviews types of neural models, runtime monitoring, and testing processes in automotive electronic development.
- Chapter 3 gives an overview of the existing scientific and industrial work, that is related in some way to the contributions of this thesis.
- Chapter 4 describes neural programs, how to define their skeleton, and how to learn means and variances of their branching statements. It also shows the use of stochastic branching in quantifying the uncertainty of adaptive CPS controllers.
- Chapter 5 focuses on performing qualitative monitoring of CPS using a digital spiking neural model, and presents a method for constructing a monitoring circuit, which computes a verdict of the temporal-logic properties.
- Chapter 6 discusses quantitative monitoring using digital spiking neural models, and the method for configuring neurons to perform computations over spike rates.
- Chapter 7 elaborates on the applications of runtime monitoring in the very rich and important domain of automotive electronic development.
- Chapter 8 discusses the formalization of the SENT and SPC protocols as industrial case studies, and the use of this formalization in runtime verification. It also provides details on recovery after detecting violations in asynchronous serial protocols.
- Chapter 9 concludes the thesis, offers a critical summary of the work, and discusses future research directions that emerged as a consequence of this work.

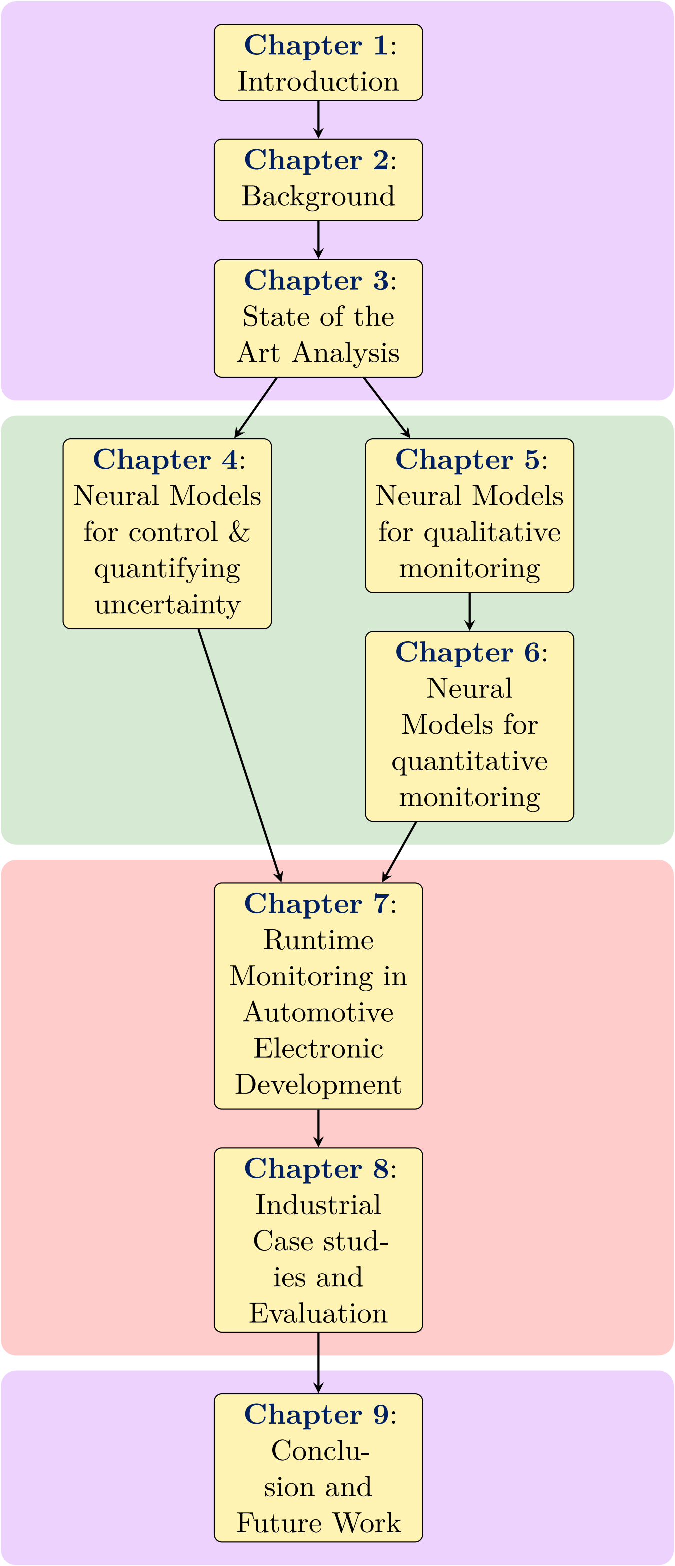

Chapter 1: Introduction
Chapter 2: Background
Chapter 3: State of the Art Analysis
Chapter 4: Neural Models for control & quantifying uncertainty
Chapter 5: Neural Models for qualitative monitoring
Chapter 6: Neural Models for quantitative monitoring
Chapter 7: Runtime Monitoring in Automotive Electronic Development
Chapter 8: Industrial Case studies and Evaluation
Chapter 9: Conclusion and Future Work

Figure 1.2: Outline of the thesis

CHAPTER 2

Background

This chapter summarizes information about well-established concepts and methodologies that form the basis of the thesis and are essential for understanding the remaining part of the work. In Section 2.1 we elaborate on neural models, emphasizing similarities and fundamental differences between artificial and biological neurons. The aforementioned neuronal models will then be used in Chapters 4, 5, and 6 for designing controllers and monitors for the *cyber* part of CPS. We then recap basic notions of the runtime monitoring in Section 2.2, discuss the specification languages for defining properties and online monitoring approaches. We will then use neural models for performing runtime monitoring in Chapters 5 and 6. In Section 2.3 we provide the details about the testing process in automotive electronic development, and then build up on this information in Chapters 7 and 8 when talking about industry-relevant parts of the work.

2.1 Neural Models

Although the history of the scientific development of neuron theory can be reconstructed since the XIX century with the fundamental breakthroughs of Camillo Golgi [She15], Ramón y Cajal [LMBA06], Hodgkin & Huxley [HH52], and many other scientists [She15] who contributed both theoretically and experimentally to the development of the neuroscience, in the field of informatics neurons start attracted significant attention from the 1943 paper of McCulloch & Pitts [MP43], where neuronal "all-or-none" activity was discussed as as a property for building logical calculus. Since then, advances in artificial neural networks are showcased by their successful applications in different types of tasks, e.g., classification (handwritten digits recognition [LBD+90]), speech [GrMH13] and image recognition [SZ14], artistic style transfer [GEB15] and many others [DY14]. The main conceptual differences of the models and architectures of biophysical and artificial neurons and networks are discussed in subsequent section.

2.1.1 Biophysical and Artificial Neurons Models and Networks

As already mentioned in Chapter 1, from biophysical point of view the primary concern is to discover “how things are”, while to solve practical problems in the field of computer science it is necessary to understand “how things ought to be”. Both perspectives see neurons as non-linear information processing elements and the interaction of these elements allows to solve complex tasks. In this section we refer to a neuron with a radial-basis activation function as an “artificial neuron”; we also refer to a generalization of Hodgkin & Huxley [HH52] model (which is not limited to sodium and potassium ionic channels, e.g. [DMSS11]) as a “biophysical model”. Formally, the artificial neuron can be described as a tuple: $(\mathbf{x}, \mathbf{w}, f, \text{out})$, where $\mathbf{x} = \{x_0, x_1, \dots, x_n\}$ and $\mathbf{w} = \{w_0, w_1, \dots, w_n\}$ are respectively n-dimensional vectors of inputs and weights; an activation function $f : \mathbb{D}_1 \to \mathbb{D}_2$ defines neural computation and maps inputs to an output, $\mathbb{D}_2$ is usually normalized to $[0, 1]$; out $\in \mathbb{D}_2$ is a variable that holds the computation result:

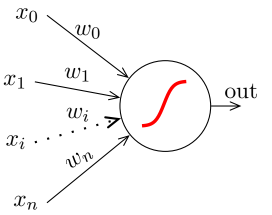
$$\text{out} = f(\mathbf{w}^T \mathbf{x}). \tag{2.1}$$

The biophysical neural model is a tuple $(V, s, I_{in}, \mathbf{g}, \mathbf{params}, \mathbf{f}_g, \mathbf{E}_i, C_m)$, where V is the membrane potential of the cell; s is the spike output; I_{in} represents input stimulus current; $\mathbf{g} = \{g_1, g_2, \dots, g_m\}$ is a vector of conductances for each ionic channel g_i, (also leak channel); a tuple $\mathbf{params} = (\mathbf{params}_1, \dots, \mathbf{params}_m)$ holds auxiliary parameters for describing temporal evolution of the conductances g_i; a vector $\mathbf{f}^{(g)} = \{f_1^{(g)}, \dots, f_m^{(g)}\}$ defines functions $f_i^{(g)} : \mathbf{params}_i \to \mathbb{R}$ that capture specific time dependence of each ionic channel, $\mathbf{E}_i = \{E_1, \dots, E_m\}$ represents potentials of each ionic channel; C_m is a membrane capacitance. Membrane potential of a neuron is then described by the ordinary differential equation (ODE):

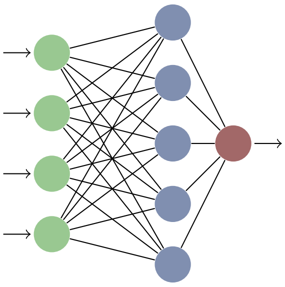
$$C_m \frac{dV}{dt} = I_{in} - \sum_{i=1}^{m} g_i f_i^{(g)} \left(\mathbf{params}_i\right) \left[V - E_i\right]. \tag{2.2}$$

The output of an artificial neuron (see Figure 2.1 and Equation 2.1) is a value of the activation function, computed over a weighted sum of inputs, while the output of a neuron in a biological simulation is a rapid depolarization of the membrane potential called *action potential* (or *spike*), which can be seen as an “all-or-none” event. The neuron outputs a spike when the following two conditions are met: (i) the membrane potential from the Equation 2.2 is above the pre-defined threshold (usually in the range 40-60 mV), (ii) the membrane potential reaches the local maximum.

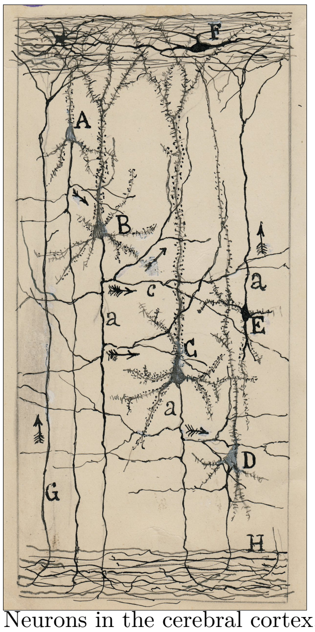
The artificial model introduces two simplifications: (i) the model is untimed and memoryless (i.e., no time dependence in Equation 2.1), while for the biophysical model (Equation 2.2) the temporal evolution of the membrane potential is an essential property; (ii) each artificial neuron has access to the state of the parent’s neuron (i.e., the “out” variable), while in the biophysical model the state (the membrane potential) and output (*spike*) are decoupled: communication between neurons happens via spikes, which lead to neuro-transmitter release and opening of ionic channels that, in turn, cause the flux of currents in the membrane. The synaptic transmission in the artificial neuron is modelled as excitatory (or inhibitory) weights, values of which are obtained during training phase, while in the biophysical model the synapses have much more intricate structure [DA05]. In the aforementioned biophysical model the synaptic transmission can be represented via introducing additional ionic channels (e.g. Ca [Lli99]) or as an external stimulus current I_{in}.



Artificial “point” neuron



Artificial neural network



Neurons in the cerebral cortex, Ramon y Cajal
Image credit: Instituto Cajal del Consjo Superior de Investigaciones Cientificas, Madrid/CSIC

Figure 2.1: Biophysical and Artificial neurons: a Bird’s Eye View

The Hodgkin-Huxley Neuron Model

A seminal model presented by Hodgkin and Huxley in [HH52] can be seen as a refinement of the biophysical model described in Section 2.1.1. The model qualitatively describes the dynamics of the membrane potential as a function of activation and deactivation of sodium and potassium ionic channels. Membrane potential is defined as an ODE, which relates ionic currents of the neuron of giant squid axon (for the detailed description the reader is referred to [HH52]). The membrane potential is computed as follows [HH52]:

$$C_m \frac{dV_m}{dt} = I_{in} - (\bar{g}_K n^4 (V_m - E_K) + \bar{g}_{Na} m^3 h (V_m - E_{Na}) + \bar{g}_L (V_m - E_L)), \qquad (2.3)$$

where the vector of conductances (see Section 2.1.1) $\mathbf{g} = \{g_{Na}, g_K, g_L\}$ comprises of sodium, potassium and leak conductances; $\mathbf{E} = \{E_{Na}, E_K, E_L\}$ represents reversal potentials of the sodium, potassium and the leak channels respectively; **params** $= (\{m,h\}_{Na}, \{n\}_K, \{1\}_L)$ are auxiliary voltage-gated variables to describe channels activation and deactivation; the temporal dependence of the parameters is guided by the functions $\mathbf{f}^{(g)} = \{\{m,h\} \xrightarrow[Na]{} m^3h, \{n\} \xrightarrow[K]{} n^4, 1 \xrightarrow[L]{} 1\}$.

The model possesses the following important properties: (i) the internal state (membrane potential) and output are decoupled; (ii) at each time step neuron integrates its membrane potential together with the inputs; (iii) the membrane potential stabilizes at the resting value in absence of external stimuli; (iv) the frequency of outputting spikes is governed by the refractory period, during which no spike can be initiated.

To make the model computationally efficient for large-scale simulations and applicable in the context of "design sciences" various simplifications and abstractions has been proposed. For instance, the programmable reset of the integration after emitting action potential is described by nonlinear integrate-and-fire models [Izh04, GKNP14]. To be able to run this type of model on the *cyber* components of CPS, careful complexity reduction is required: avoiding computationally-expensive operations in digital hardware (e.g., floating point operations). In the subsequent section we describe the spiking neural model that is suited for implementation on the digital hardware, while still possessing the necessary properties of the biophysical model.

The TrueNorth Neuron Model

The TrueNorth model is proposed by IBM [CMA+13] and describes a digital spiking neuron, which follows the properties of the biophysical model introduced in previous section. All the parameters of the model are either integer or boolean values, which facilitates the implementation of the model in digital hardware (e.g., in Field-Programmable Gate Array (FPGA)). The model performs three computational steps: (i) the synaptic integration of the current membrane potential and the incoming inputs from the pre-synaptic neurons; (ii) the leak integration to model the dissipation of energy in absence of input; (iii) the "threshold-fire-reset" defines the reset behavior of the model upon reaching the thresholds. The model has deterministic and stochastic modes; we review the deterministic part of the TrueNorth model below. For an extended explanation, the reader is referred to [CMA+13].

Synaptic Integration is the first computational step where every neuron sums up the products of its inputs $A_i(t)$ and weights s_{ij}. Every input is enabled by a flag w_{ij}. The result is added to its previous membrane potential $V_j(t-1)$. Although in the original model the maximum number of inputs bounded by 255, we drop this restriction and assume that every neuron has a configurable $N \in \mathbb{N}$ number of inputs (the original assumption comes from the chip restrictions):

$$V_j(t) = V_j(t-1) + \sum_{i=0}^{N} A_i(t)\, w_{ij}\, s_{ij} \tag{2.4}$$

Leak Integration accounts for energy dissipation, self-stimulation, and convergence to an equilibrium in the absence of input. A TrueNorth neuron n_j can exhibit negative, zero or positive leak λ_j. To express divergent and convergent leak behaviors the leak reverse flag ϵ_j can be set: in this case the leak changes its sign with the membrane potential's sign (i.e., when the signs are different, the leak forces V_j converge to zero) [CMA+13]:

$$\Omega_j = (1 - \epsilon_j) + \epsilon_j \mathrm{sgn}(V_j(t)) \tag{2.5}$$

$$V_j(t) = V_j(t) + \Omega_j \lambda_j \tag{2.6}$$

Threshold, Fire, Reset is computed at each time step to generate the binary "all-or-none" output (spike or no spike). A neuron n_j possesses a positive threshold α_j and a negative threshold β_j. When the membrane potential V_j exceeds α_j, the spike is generated, and the membrane potential is reset. The TrueNorth model is extended with three reset modes γ_j: (0) normal, (1) linear, or (2) non-reset. When V_j falls below the negative threshold β_j, no spike is generated, although the membrane potential is updated depending on the reset mode γ_j and the saturation flag κ_j.

$$\begin{array}{lll}
\texttt{if} & V_j(t) \geq \alpha_j & (2.7)\\
 & \textsc{Spike} & (2.8)\\
 & \gamma_j = 0 : V_j(t) = R_j & \\
 & \gamma_j = 1 : V_j(t) = V_j(t) - \alpha_j & \\
 & \gamma_j = 2 : V_j(t) = V_j(t) & (2.9)\\
\texttt{elseif} & V_j(t) < -\beta_j & (2.10)\\
 & \texttt{if}\ \ \kappa_j = 1 & \\
 & \quad V_j(t) = -\beta_j & \\
 & \texttt{else} & \\
 & \quad \gamma_j = 0 : V_j(t) = -R_j & \\
 & \quad \gamma_j = 1 : V_j(t) = V_j(t) + \beta_j & \\
 & \quad \gamma_j = 2 : V_j(t) = V_j(t) & (2.11)
\end{array}$$

2.2 Runtime Verification

Runtime verification[1] [Leu12, FZ12, Mal16, BLS11] is a *lightweight* verification technique, which is concerned with a derivation of a verdict for a formally-expressed correctness property φ. To derive a verdict, a monitor is attached to the observable outputs of the system under scrutiny. Figure 2.2 shows a general way of integrating runtime monitoring with an existing system: based on the observations from the Hardware/Software System, a monitor delivers a verdict if a predefined specification is satisfied/violated. Runtime

---

[1] The terms "runtime verification" and "runtime monitoring" are used interchangeably in the thesis

monitoring is non-intrusive and outputs whether the system satisfies its formal requirements. If, on the other hand, violation of the specification triggers control actions from the observer to the system (i.e. the dashed line in Figure 2.2) the observer is seen as a runtime *enforcer* [FMFR11, PFJ$^+$13] of the property. The questions about enforcement of formal properties are beyond the scope of the thesis.

Apart from the model checking, 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 [Leu12] tackles a more modest problem: *"Does the 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.

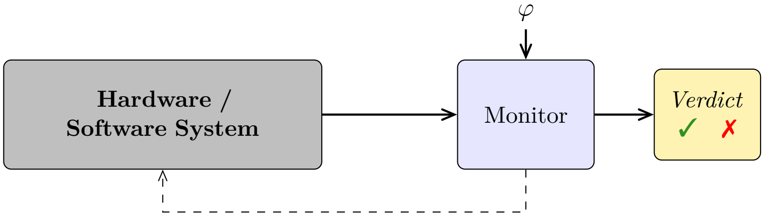


Figure 2.2: Runtime Monitoring: System's overview

Runtime monitoring usually deals with finite traces, and in online case the traces increase in size with the progress of time [BLS11]. In Section 2.2.2 we review the differences of monitoring hardware and software systems. As the monitor checks the formally defined property φ, in the Section 2.2.1 we discuss formal specification of system's properties.

2.2.1 Specifications of Temporal Properties

Although in state-of-the-art engineering practice still significant fraction of requirements are expressed using combination of natural language and pictures (e.g. the requirements for the SENT and SPC protocols), this way of specifying requirements is subject to ambiguities and mis-interpretations between different communicating parties, and can be a source of potential errors. Languages with formal semantics enable an unambiguous interpretation of the properties in question and facilitate automatic monitor generation.

The relation of predicates over progression of time can be formulated using temporal logics [Lon89]. Linear Temporal Logic (LTL) [MP92] is a well-established formalism that allows to reason about temporal relations of events over infinite traces: (i) the notion of time in LTL is rather logical then physical, (ii) temporal operators are unbounded and in general only a subset of formulae can be evaluated over a finite prefixes – these main motivations fostered extensions of the logic to be applicable in the domain of real-time systems. Metric Temporal Logic (MTL) [OW08] is a real-time extension of LTL, which introduced bounded temporal operators. For analog-mixed signal components (which usually interface *physical* and *cyber* components of CPS) not only the time, but also the input variables are usually in domain of reals. We now consider the extension of MTL that is used in the thesis for specifying temporal behavior of analog-mixed signal systems.

Signal Temporal Logic (STL) [DMB+12] allows to specify mixed-signal properties of analog/digital components. 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 [NN14]:

$$\varphi := p \,|\, x \sim c \,|\, \neg\varphi \,|\, \varphi_1 \vee \varphi_2 \,|\, \varphi_1 \mathcal{U}_I \varphi_2 \,|\, \varphi_1 \mathcal{S}_I \varphi_2, \tag{2.12}$$

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 (where $w_x[i]$ we denote x^{th} component of w):

$$\begin{aligned}
(w, i) &\models p && \iff && p[i] = \top && (2.13)\\
(w, i) &\models x \sim c && \iff && w_x[i] \sim c && (2.14)\\
(w, i) &\models \neg\varphi && \iff && (w, i) \not\models \varphi && (2.15)\\
(w, i) &\models \varphi_1 \vee \varphi_2 && \iff && (w, i) \models \varphi_1 \text{ or } (w, i) \models \varphi_2 && (2.16)\\
(w, i) &\models \varphi_1 \mathcal{U}_I \varphi_2 && \iff && \exists j \in (i + I) \cap \mathbb{T} : (w, j) \models \varphi_2 && \\
& && && \text{and } \forall i < k < j, (w, k) \models \varphi_1 && (2.17)\\
(w, i) &\models \varphi_1 \mathcal{S}_I \varphi_2 && \iff && \exists j \in (i - I) \cap \mathbb{T} : (w, j) \models \varphi_2 && \\
& && && \text{and } \forall j < k < i, (w, k) \models \varphi_1. && (2.18)
\end{aligned}$$

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:

$$\begin{aligned}
(w, i) &\models \Diamond_I \varphi && \iff && \exists j \in (i + I) \cap \mathbb{T} : (w, j) \models \varphi && (2.19)\\
(w, i) &\models \Box_I \varphi && \iff && \forall j \in (i + I) \cap \mathbb{T} : (w, j) \models \varphi && (2.20)\\
(w, i) &\models \diamondminus_I \varphi && \iff && \exists j \in (i - I) \cap \mathbb{T} : (w, j) \models \varphi && (2.21)\\
(w, i) &\models \boxminus_I \varphi && \iff && \forall j \in (i - I) \cap \mathbb{T} : (w, j) \models \varphi. && (2.22)
\end{aligned}$$

From now on we consider all STL specifications of the form $\Box \varphi$, for brevity of notation we omit the implicit globally operator from the specification if it is clear from the context. Figure 2.3 shows examples of STL future and past temporal operators: at the time instant t_i, we look either forward (future) or backward (past) over the time axis to evaluate the corresponding operator. Monitoring of the future STL formulae is *acausal*, i.e. the evaluation of the formula at a time instant t_i might depend on time interval $t_i + I$.

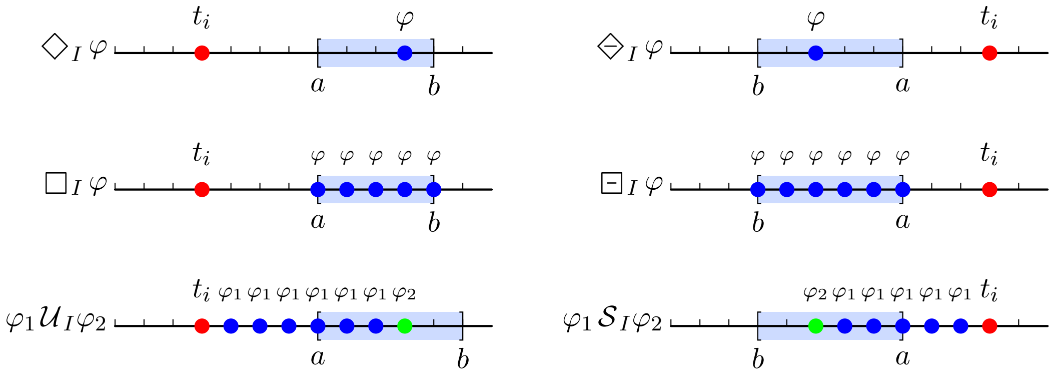


Figure 2.3: STL past and future temporal operators

2.2.2 Runtime Monitoring in Software and Hardware

Programming a software application and designing a hardware component involve various technology stacks, and base on different abstraction layers. A view of a program as an abstract entity, which maps inputs to outputs with a termination as a desired property differs from a view of reactive systems, where hardware and software components are tightly coupled with ongoing interaction with the environment. It is usually the case that the speed of processing events is significantly larger in hardware systems, e.g. in Chapter 8 runtime monitor of the sensor interface evaluates the specification at the rate of $\approx 10^6$ times per second, which cannot be achieved in a software system.

2.3 Automotive Electronic Development

In order to comply with strict safety standards (e.g. ISO 26262 [iso11]) manufacturers in the automotive electronic industry often follow a variant of the V-model [RSB$^+$14], where the development process starts with identifying requirements from the Original Equipment Manufacturer (OEM) or Tier 1 partner, designing a concept or a prototype, followed by the implementation and evaluation activities, testing, and possibly iterating back to the design phases.

The Verification and Validation (V & V) phase in this type of design cycle accounts up to 60% of the total development [NN16], and it is necessary to plan V & V activities early on, alongside with first iterations of the concept development. Main activities include [NN14] (i) *pre-silicon* verification, where the proposed solution is simulated, both at component and system level; (ii) *emulation* of the developed hardware on the existing hardware platform to assess implementation bottlenecks before manufacturing; (iii) *post-silicon* verification, where the manufactured component is evaluated – as seen from these activities, the V & V phase in the state-of-the-art industrial environment comprises extensive product testing under different scenarios, including stress conditions – which is different from verification in a sense of model checking where an formal evidence (e.g. a proof) is obtained that the investigated system will not violate safety properties. Although model-based design (e.g. SCADE [MSH$^+$13]) can be used to generate C-code alongside with the proof, this approach is not directly applicable in the context of integrated circuit development.

To reduce the V & V time without sacrificing coverage it is important to automate the testing phase for the engineer, and to reuse verification scenarios and tools across validation and verification phases.

CHAPTER 3

State of the Art Analysis

This chapter provides an overview and analysis of existing approaches related to the topic of the thesis. Since the presented contributions are closely connected with several research fields, state of the art analysis has the following structure: (i) we first discuss related work concerning adaptive software control of CPS and neural programs in particular; (ii) we then elaborate on applications of neural models for performing qualitative and quantitative runtime monitoring. For the industry-relevant parts of the work we analyze existing approaches for protocol verification, formalization, and hardware runtime monitoring related to the SENT and SPC communication protocols.

3.1 Related Work on CPS Control with Neural Models

In order to apply key concepts of neural models in the context of CPS control (Chapter 4) we elaborate on state-of-the-art methods for creating controller software for CPS. Adaptive and intelligent control are established fields in control theory [AW94, AP93], which aim respectively to adjust the parameters of the controller during system's runtime and use methods from Artificial Intelligence (AI) for achieving control objectives. Components of CPS operate in dynamic and uncertain environments, which should be reflected in design of controller software. From the computer science perspective, we aim to incorporate properties of neurons as functional plasticity and learning in the controller design.

In our work we propose special neural-inspired control-flow statements for controller software (namely *nif* and *nwhile* statements, discussed in detail in Chapter 4), which can be explicitly related within a Gaussian Bayesian network (GBN) and allow functional plasticity (i.e. adapting) of a controller during system's runtime. Moreover, we also considered how to encode within the proposed framework a biological controller of the C. Elegans neural circuit, and implemented the framework on top of the Robot Operating System (ROS), which enabled adaptive parallel parking of the Pioneer [Ade] robot.

Although probabilistic programs, GBNs and neural networks were studied before, the development of control and simulation programs with smooth thresholds and the associated learning part is, to the best of our knowledge, new and not considered before.

Probabilistic programs introduced in [GHNR14] differ from “traditional” ones by the ability to sample at random from the distribution and condition the values of variables via observations, with the result of the program equals to the expectation of the return value. This approach requires static analysis to give an output of the program, which is unpractical for controlling CPS subsystems, since controlling such systems should be done in real-time together and provide capabilities of adapting the outputs of the controller. Kaminski et. al. [KK17] studied execution of probabilistic programs with random variables and the expected values after program termination, and proposed extension of pre-expectation calculus for dealing mixed-signed post-expectations.

In [CSL10], the authors adapted the signal and image processing technique called Gaussian smoothing (GS), for program optimization. Using GS, a program could be approximated by a smooth mathematical function, which is a convolution of a denotational semantics of a program with a Gaussian function. This approximation facilitates solving the parameter synthesis problem. In [CSL11] this idea was extended to define soundness and robustness of smooth interpretation of programs. In these works the authors do not consider any methods for eliminating the re-normalization step of the Probability Density Function (PDF) when a variable is passed through a conditional branch in the execution trace. Moreover, they did not consider to apply their approach for simulating neuronal circuits and in particularly tap withdrawal circuit of C. Elegans.

Our framework allows to associate a GBN with the program skeleton, and learn parameters of the network. In general, learning Bayesian Networks comprises different tasks and problem formulations: *i*) Learning the structure of the network, *ii*) Learning the conditional probabilities for the given structure, and *iii*) Performing querying-inference for a given Bayesian Network [Nea03]. In [HG95] the authors introduce a unified method for both discrete and continuous domains to learn the parameters of Bayesian Network, using a combination of prior knowledge and statistical data. We adapt this method, and use it for updating variances about each control action during the system run.

Fuzzy control [PY97] and our neural control method have different ontological commitments: within our approach statements about the world are either true or false, and partial knowledge renders controller’s beliefs to be probabilistic. In fuzzy-logic control the statements about the world are fuzzy (have a continuous domain of values) whereas the knowledge about the world is total. Moreover, fuzzy logic implicitly assumes (in the definition of conjunction) that variables are independent, whereas we do not require this assumption. In fact the dependence between variables plays a key role in learning parameters of a GBN and identifying variances about control decisions.

As every controller, a neural controller must be aware of the internal state of the process to be able to robustly control the CPS. Sensors measure the outputs of a process, whereof the state can be estimated. The measurements are distorted by noise and the environment may be unpredictable. State estimators [Mit07, TBF06, AMGC02] and in particular Kalman filters [TBF06, WVdM00] are commonly used methods to increase the confidence of the state estimate evaluated out of raw sensor measurements.

In Chapter 4 we showcase our framework on the parallel parking case study of the mobile robot. Various formulations of a mobile parking problem were extensively studied for robots with different architectures [IMDACPR$^+$12, JS99, KD05, SGC$^+$12, LYW$^+$10]. In [LYW$^+$10] the authors use a custom spatial configuration of the ultrasonic sensors and binaural method to perceive the environment and park the robot using predefined rules. Another approach [KD05] is to approximate the trajectory for the parking task with a polynomial curve, that the robot could follow with the constraints satisfied, and minimize the difference between specified trajectory and actual path. In [ACN10] the authors try to infer a "hidden trajectory" from a series of observations, while in our setting in order to adapt we allow all admissible trajectories.

3.2 Related Work on Neural Models in Monitoring

In Chapters 5 and 6 we apply neural models for qualitative and quantitative runtime monitoring of temporal logic properties. Acceptable overhead of runtime monitoring, the ability to check a system as a "black-box" without the system's model, made it appealing both from theoretical [SBS$^+$12, KBS$^+$13] and practical perspectives [WH08, BLMA$^+$05].

The authors in [SBS$^+$12] explore overhead-accuracy trade-off of a runtime software monitor and use Hidden Markov Models (HMMs) to retain high probability of being accurate while introducing gaps in observations. In [MN13a] the authors provide a procedure to directly check properties of signals in continuous time and monitor mixed-signal specifications. Fast, hardware-based, online monitoring has also drawn a great deal of attention in recent years [RFB14, DGG$^+$05a, CES13]. In particular: 1) Synthesis of monitors for safety and liveness LTL properties [CES13], 2) Design of sophisticated architectures to record events for MTL property checking [RFB14], 3) Synthesis of checkers for Property Specification Language (PSL) assertions [DGG$^+$05a]. Although concerned with hardware monitoring, all these works did not, however, consider applying neuronal architectures for the task.

Identifying a suitable neural model for runtime monitoring should be done with care: From a biophysical point of view each neuron has on the order of 10^4 of synapses [SP89]; state-of-the art models account about 20 ionic channels, 150 state variables and 500 parameters [DMSS11]. Moreover, since the synapses are structurally and functionally plastic devices, the dendritic spines of the neuron [TCK$^+$02] and the efficacy of synapses change during the operation (e.g. spike-time dependent plasticity [DP06]). We are aware that taking into account geometrical topology allows to increase the expressiveness and e.g. perform orientation selectivity in the dendritic inputs [JRCK10], though to remain computationally efficient we consider only the single-compartment neural models [Bre15].

In the seminal paper [HH52] Hodgkin and Huxley presented a conductance-based spiking neural model that describes the dynamics of generating an action potential, the role and function of sodium and potassium ionic channels. The model of Hodgkin and Huxley is biophysically accurate [Izh04], and has been refined with other type of ionic channels [DMSS11]. A neuron is modelled as an active RC-circuit, in which the opening of ion channels follows in response to influx of external current stimulus. The membrane potential, inward (sodium), and outward (potassium) currents are modelled as a set of differential equations. Although numerous software and hardware implementations (e.g. [Hin93, GD07] and [SWM92, GBL04, ILBH$^+$11, MH15] respectively) of the Hodgkin and Huxley model are available, to the best of our knowledge we are not aware of the works that study computations with spike rates and explicitly compare the biophysical model with hardware-optimized spiking models.

The leaky-integrate-and-fire (LIF) models approximate biophysical single-compartment neural models, and capture relevant behaviors of neurons [GKNP14]: 1) Neuronal dynamics can be seen as summation process (i.e. integration); 2) Membrane potential of neurons leaks over time to its resting value; 3) Neurons communicate information via spikes, the form of which is not important, and only their number over time is of relevance. The approximation of the neuron behavior can then be summarized as follows:

- *Synaptic Integration:* gather inputs from other neurons;
- *Leak:* decrease membrane potential by a leak amount;
- *Firing & Reset:* spike and reset of excited neurons.

Although the LIF models approximate biological behaviors, with approximations often being vague, empiric, and facing simplification trade-offs due to enormous parameter space, the LIF models have also been analyzed using formal methods and tools. Aman et. al. [AC16] and Ciatto [CMG17] independently from each other developed formalization of the LIF models as timed automata, and then used UPPAAL to check the properties of neural networks defined as temporal logic formulae. The authors in [AC16] identified and checked path properties, such as particular assignments of output neurons after performing the computation. Ciatto [CMG17] formalized a subset of biophysical firing patterns in CTL, and used UPPAAL to verify whether given parameters satisfy specification.

As far as underlying hardware implementation is concerned, Cassidy et. al. [CMA$^+$13], Esser et. al. [EAA$^+$13], and Amir et. al. [ADR$^+$13] in a series of papers introduced the TrueNorth hardware architecture, which is based on a versatile spiking neuron model briefly introduced in Section 2.1.1. The TrueNorth model [CMA$^+$13] is digital with all the parameters being either integers or boolean values, since floating point computations are expensive in hardware. The proposed TrueNorth model is an extension of the LIF models. The model, beside reproducing relevant biological behaviors, can also be used for deterministic and stochastic computations. While the use of the TrueNorth model in capturing biological behaviors is described in some depth in [CMA$^+$13], this is not the case for the definition of logical or arithmetic functions. In Chapters 6 and 7 we investigate how to configure the model to obtain the behaviors of interest: quantitative monitors and computational blocks.

In [ADR^{+}13] the authors present a *corelet*: a programming framework and an abstraction that encapsulates neurons performing a specific task. In [EAA^{+}13] the authors consider the application of IBM's neuronal architecture for digit and tone recognition, HMM sequence modelling, and eye detection, with special spiking retina sensors. The application leveraged the presented programming model, which combined "corelets" corresponding to the different functions. Their description does not reveal implementation details and is not reproducible, the authors also did not consider monitoring as a possible application. In contrast, we build our monitors on top of the TrueNorth neural model starting from single neurons, combining them into a hierarchy: In Chapter 5 of the thesis we look at the problem in a different way and present a method to evaluate MTL specifications φ using a neural circuit of spiking neurons.

Although the authors in [CMA^{+}13, ADR^{+}13, EAA^{+}13] implemented the architecture on the dedicated hardware chip, it is not publicly available on the market. We aim, on the contrary, to develop neuromorphic hardware accelerators on FPGAs and ZYNQ architecture in particular, which are widely available, have an established design flow, and allow Advanced eXtensible Interface (AXI)-style communication between the processing system and the programmable logic. We also implemented a python open-source simulation of both biophysical and digital models which steps towards the target hardware implementation of neuromorphic accelerators on a ZYNQ FPGA processing system.

The work of [RBNG16] draws connection between temporal logic and convolution. In order to compute compliance of a signal with a specification φ, one can obtain the results by performing convolution of the signal of interest with a specific window, which is directly derived from φ. Using *min/max* interpretation of a product/sum, classical qualitative MTL semantics is obtained. We follow on this result and develop a circuit, which we can use for evaluating past MTL formulae. We have developed implementation of this semantics using the TrueNorth neural model.

3.3 Related Work on Runtime Verification in Automotive

Runtime verification of formally defined properties, in its general view, is an extremely diverse research area in terms of requirements-specification languages [Eis07, VR14, DMB^{+}12, MN13b, FMNU15], approaches to construct the monitors [TRV12, SMR15, BZ06, PZ08a], and target applications [JKN10, NN16, KFL15, RRS14].

Evaluating a temporal logic specification φ over a trace or a signal is usually associated with either automata construction [NP10, MNP06] or a concept of temporal testers [PZ08a] by Pnueli. In [NP10] the authors presented a technique to build a deterministic timed automaton that accepts the traces that satisfy an MTL formula φ. Heffernan et. al. [HMF14] analyzed how functional safety properties, identified in the ISO 26262 standard, can be runtime-checked for a gear shift controller by an automaton-based monitor for past-LTL specifications, however the gear-control automaton does not scale well when the number of requirements increase further, which is vital for life cycle system support. Pnueli et.al. [PZ08a] proposed a compositional, a “transducer”-like way of evaluating temporal logic formulae, which was successfully applied in hardware runtime monitoring, e.g. in [JBG$^+$15]. In the thesis, this way of building monitors is used extensively, as it allows to build the monitoring system hierarchically.

The FoC framework of IBM [DGG$^+$05b, Eis07] allows to generate monitors for PSL assertions. Although PSL allows to specify the evolution of a system, the formal semantics is based on the sequence of states and does not include a notion of time explicitly. STL [MNP08] and Timed Regular Expressions (TRE) [ACM02], on the other hand, were designed to deal with real time, and allow to precisely identify time intervals of interest and bound temporal modalities to these intervals.

In automotive industry, strict safety standards (e.g. ISO 26262 [iso11]) require from manufacturers (OEM, Tier 1, Tier 2 suppliers) to provide assurance guarantees of safety properties for corresponding Automotive Safety Integrity Levels (ASILs). Although, as pointed out by Dijkstra [Dij70], “Program testing can be used to show the presence of bugs, but never to show their absence!”, it is still a highly-accepted and predominant practice for providing a witness of requirements conformance. The use of formal verification and model-based design (although have shown their strengths [MSH$^+$13]) have still limited applicability due to complexity and heterogeneity of the associated systems. In [KF12] the authors elaborate on employing static analysis techniques for checking properties of programs to comply with ISO 26262, yet without concrete examples or evaluation.

To facilitate finding errors, several testing techniques have been proposed. Fainekos et. al. [FSUY12] used guided stochastic search for inputs and parameter space of the model under test to steer the simulation to a falsifying trajectory. The steering relies on minimizing a robustness metric which measures distance between a temporal logic specification and a trace. In general, the proposed method can be seen as an extension of “constraint random verification” [YPA06], where to reduce test creation time and remove human bias the Design Under Test (DUT) is run against restricted random stimuli.

Automotive electronic components combine digital and analog parts, which could create intricate influences and hard-to-discover failures, and, in practice, are often needs to be validated over long-term runs. Hardware [SLB$^+$08, NN14, NW14] and software [DWPM11] emulation is increasingly used to speed up the testing process and verify system components in isolation. Drolia et. al. in [DWPM11] developed an automotive testbed, that connects Electronic Control Units (ECUs) with a car simulation environment and a middleware for diagnosis and evaluation. The authors collected runtime information and evaluated modelled and implemented controllers, although they did not define properties to evaluate in a formal way. FPGA-based development [NN14, NW14, NBHT14] accelerates model evaluation and enables real-time testing in safety critical automotive electronic system development at an early stage and helps to overcome simulation bottlenecks.

Hardware runtime monitors [JBG$^+$15, GRS14, RFB14] are usually generated directly in

Hardware Description Languages (HDLs) (i.e. VHDL or Verilog). High-Level Synthesis (HLS) is an alternative way that transforms behavioral code (e.g. C/C++/SystemC) to Register-Transfer Level (RTL) cycle-accurate hardware implementation. As shown in Chapter 7, HLS facilitates the reuse of monitors in different stages of automotive sensor development and helps speeding up the testing process. In addition, the application domains of previous works in hardware temporal logic monitoring [GRS14, RFB14] are quite different from the chip design for the automotive industry.

As far as hardware implementation is concerned, Schumann et. al. [SMR15] propose an FPGA implementation of runtime monitors for the Unmanned Aerial Vehicle (UAV) applications. The authors construct FPGA monitors for security requirements and specify possible attacks that a UAV might undergo. A Bayesian network on top of MTL monitoring allows to estimate system health. The authors do not take into account neither the recovery of monitors after violations nor the electrical characteristics of signals, and define their properties on a higher level of abstraction. On the contrary, in Chapter 8 we focus on formalizing the electrical and timing requirements of low-level SENT and SPC protocols, with an emphasis on monitor recovery after capturing specification violations.

In a similar context we refer to the work of Reinbacher et al. [RFB12]. The authors present a framework for monitoring past-time MTL specifications. In order to achieve the reconfigurability of the system, they introduce an over-complex hardware architecture. In our case, we specifically target asynchronous serial protocols, for which we find the TRE formalization with simpler, automaton-based architecture more appropriate.

3.4 Related Work on Protocol Verification

The need to assure fulfillment of specific protocol requirements is reflected in literature in describing a protocol in Domain Specific Languages (DSLs), (e.g. an early work on LDP [Hof80] and more recent studies on applying DSLs for cache, network, and distributed protocols [CRL99, BBHM09, Adh13]), in applying model checking [DSO13], and formalizing requirements in temporal logics (LTL, MTL) and their extensions e.g. PSL [JMB17]. While, in general, deadlock and liveness properties play crucial roles in protocol verification, validating not only logical, but also physical requirements is essential for correct data transfer. In automotive systems, it is equally important to ensure that the timing and electrical parameters of the protocol are met by the communicating parties in order to guarantee the correct data transmission. In Chapter 9 we elaborate on formalization of requirements and verifying the SENT and SPC protocols.

UPPAAL [BDL04, BDL$^+$11] is a well-established tool for verifying real-time systems which can be modeled as a composition of timed automata. This tool provides a description language for modelling, a simulator, and a model checker. The works of [BGK$^+$02, RNPH05, PGZ$^+$14] showed successful application of the UPPAAL in verifying properties of protocols. The high-level summary of approaches in [BGK$^+$02, RNPH05, PGZ$^+$14] is as follows: (i) to model the communication protocol or its parts within the UPPALL framework; (ii) to define error scenarios; and (iii) verify the model against these scenarios.

The CPS, which embody complex interactions between analog and digital components are often difficult yet almost impossible to model in detail in terms of this formalism, and the bugs that may arise due to interface interactions may be left undiscovered by UPPAAL. In contrast, the goal of the applied part of the thesis is to create standalone monitors for verifying a discrete time system at runtime. Our monitors are ignorant of the model of the system, and check the actual implementation apart from the model.

We are aware of several case studies on monitoring temporal logic specifications - the automotive bus standard [NN14], the DDR2 memory interface [JKN10], and automotive controllers functional requirements [FSUY12]. The authors in [NN14] formalized a part of the discovery mode of the 3rd generation Distributed System Interface (DSI3) in STL and check pre-generated simulation traces. Jones et. al. [JKN10] showed how clock jitter and data strobe alignment properties are formulated in a PSL/STL specification. All of these works focus on offline monitoring and continuous-time semantics, which covers STL and does not consider specifications based on regular expressions, and omit monitor recovery aspects after capturing a violation. In Chapter 9 we compare two formalisms and implementations, to increase integration readiness level for the monitor itself and eliminate the “single source of truth” aspect from the monitoring system.

In [FMNU15] the authors also use TREs with events to evaluate the performance of a controller and sensor implementation. Orthogonally to our work, they define measurement specifications over timed patterns.

Figure 3.1 shows high-level overview of the related work.

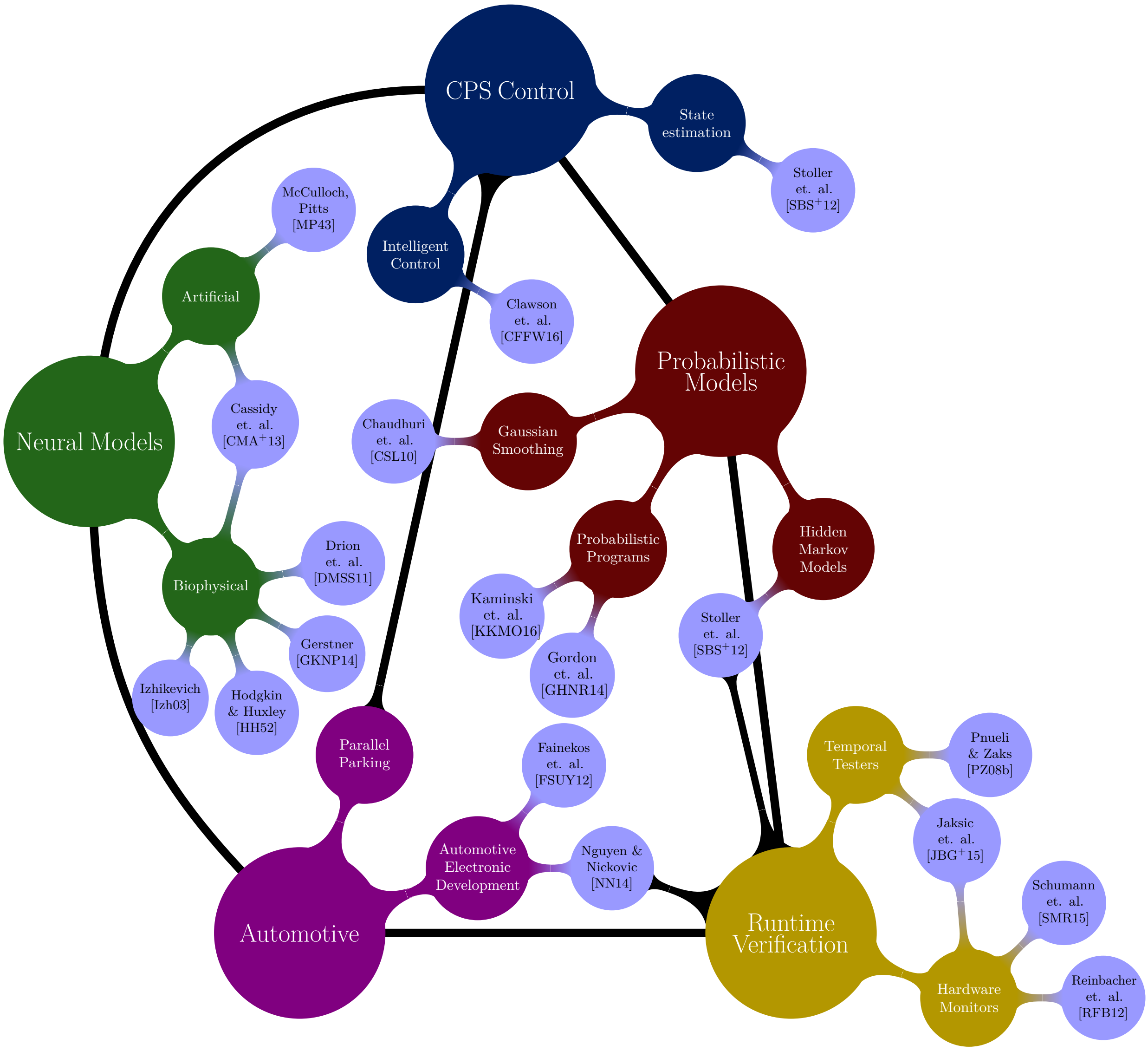


Figure 3.1: High-level overview of the related work

CHAPTER 4

Neural Models for Control & Quantifying Uncertainty

To equip CPS with the ability to adapt and act in uncertain environments, various researchers investigated whether current CPS analysis, design and implementation techniques possess sufficient adaptability capabilities. Parnas, Chaudhuri and Lezama identified in a series of papers [Par85, CSL10, CSL11] that smoothing program executions is a promising method for reasoning about programs that operate random variables. In this chapter, we use techniques inspired by smooth program interpretations [CSL10] and inherent smoothness of the neuron's activation functions to address the first research question of the thesis (**RQ 1**), which was discussed in detail in Chapter 1, to develop methodology that support, on average, "smooth" decisions for CPS controllers.

In a simple decision of the form `if (x > a)`, the predicate $x > a$ acts like a step function (the bold black line in Fig. 4.1), with infinite plateaus to the left and right of the discontinuity point $x = a$. The nesting of if-then-else statements leads to a highly nonlinear program state space inducing a large number of plateaus separated by discontinuous jumps. This has important implications for CPS controller software, which operates in uncertain environments and receive noisy input measurement data.

From a CPS design point of view, where one is interested to find the values of a variable for which an optimization criterion is satisfied, plateaus in the program state space lead to difficulties in CPS optimization. To alleviate this problem, Chaudhuri and Lezama [CSL10] proposed to smoothen the steps by passing a Gaussian input distribution through the CPS. The authors, however, stop short of proposing new methodology that includes neural programming constructs, which act smoothly, similar to a neuron activation function, and are capable to quantify uncertainty of a controller action.

From a CPS implementation point of view, when dealing with random variables and noisy measurements of CPS it is important to be able to (i) adapt to a changing input in a long term, and (ii) tolerate short-term disturbances. In the AI community, where steps are called *hard neurons* and sigmoid curves are called *soft neurons*, adaptation and robustness is achieved by learning a particular form of Bayesian networks with soft-neuron distributions. Such networks have recently achieved noteworthy performance, for example in recognition of sophisticated patterns [CMS12, EBC$^+$10].

Having identified the major challenges in the design and implementation of CPSs, we propose a combined approach for developing adaptive CPS controller software using Neural Programming Constructs (NeuralP)[1]. In the NeuralP framework, a controller (program skeleton) is written representing an underlying neural network. Additional knowledge about the thresholds in the controller's `nif` and `nwhile` conditions, the key NeuralP constructs, is then encoded as a GBNs and updated during the system run.

In contrast to traditional deterministic program controllers, which capture only one execution for any given input, NeuralP controllers capture a set of valid executions. For example, if one has enough space between two cars, there are multiple ways to parallel park, all of which are valid (i.e. result of a car being parked at a dedicated parking spot without collision). There is no reason to restrict the parking task to only one trajectory (as in the traditional controller case), because a small perturbation (for example sliding or target point overshooting), may lead to an invalid trajectory (i.e. trajectory leading to collision). In NeuralP, a small perturbation may eliminate a part of the valid trajectories, but leave enough of the valid ones such that the controller can still finish the mission and adapt. By adaptation we understand the ability to react on environment's change by eliminating trajectories over time that are no longer valid.

To validate our CPS controller design, we define three controllers for two case studies: *Parallel parking* from [CSL] and the *Tap Withdrawal neural circuit* of C. Elegans [WRR96]. In the parallel parking case study, we show how to achieve robustness by expressing the controller as a NeuralP, where the associated GBN helps to compensate for perturbations in the environment. We provide a technique for learning the parameters of a GBN from traces. In the second case study, the controller is C. Elegans' neural circuit for tap withdrawal, expressed in terms of NeuralP. Each synaptic link of a neuron can either fire or not, which corresponds to a `nif` statement. Since NeuralP is a general concept, it can be used as a language for expressing controllers or to model the system. In the third case study, we use knowledge gained from the C. Elegans circuit to provide a controller for parallel parking, expressed as a neural circuit, where the voltage, current, and conductance become position (or angle), velocity, and control flow, respectively.

The main contributions of this chapter can be summarized as follows:

1. *We propose NeuralP, a new programming framework* for the development of CPS controllers where step guards are replaced by smooth counterparts (`nif` and `nwhile`).
2. *We demonstrate the versatility of this new framework* on two case studies: an adaptive parallel parking controller for a Pioneer rover (the YouTube videos are available at [neu]), a tap-withdrawal neural circuit for C. Elegans, and a parallel-parking neural-circuit.

---

[1] Hereafter we use NeuralP to abbreviate both a neural program and neural programming framework

The rest of the chapter has the following structure. Section 4.1 provides the necessary preliminaries. Section 4.2 introduces our programming framework. Section 4.3 focuses on learning parameters of the GBN. Section 4.4 presents our case studies, implementation platform, and experimental results. Section 4.5 gives the chapter summary.

4.1 Preliminaries

Bayesian Networks A probabilistic system is completely characterized by the joint probability distribution of all of its (possibly noisy) components. However, the size of this distribution typically explodes, and its use becomes intractable. In such cases, the Bayes' rule allows to successively decompose the joint distribution according to the conditional dependences among its random variables (RVs). These are both discrete or continuous variables, which associate to each value (or infinitesimal interval) in their range, the rate of its occurrence. Networks of conditional dependencies among random variables are known as Bayesian networks (BNs), and they have a very succinct representation.

Syntactically, a BN is a direct acyclic graph $G = (V, E)$, where each vertex $v_i \in V$ represents a random variable X_i and each edge $e_{ij} \in E$ represents a conditional dependence of the variable X_j on the variable X_i. To avoid the complications induced by the use of the joint probability distribution (or density), each variable X_i is associated with a conditional probability distribution (CPD) that takes into account dependencies only between the variable and its direct parents [RN10, KF09]. Such a compact representation keeps information about the system in a distributed manner and makes reasoning tractable even for large number of variables. The variables of a BN could have discrete (e.g. fault detection, a device might have only a finite number of diagnosable errors, caused by a finite set of faults), or continuous distributions (e.g. in the *parallel parking* running example, a Pioneer rover starting from an initial location, needs to execute a sequence of motion primitives, and which have continuous distribution).

Gaussian Distributions Any real measurement of a physical quantity is affected by noise. Hence, the distances and the angles occurring in the parking example are naturally expressed as continuous RVs. We assume that variables have Gaussian distribution (GD).

An univariate Gaussian distribution (UGD) is denoted by $\mathcal{N}(\mu, \sigma^2)$ and it is characterized by two parameters: The *mean* μ and the *variance* σ^2. In our example, the desired distance in the first motion is associated with μ, which is perturbed by noise with variance σ^2. The PDF of a RV X with values x is defined as follows:

$$\texttt{pdf}_{\mu,\sigma^2}(x) = \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right). \tag{4.1}$$

Parallel parking includes a sequence of motion primitives that are mutually dependent. To express these dependencies we use a multivariate Gaussian distribution (MGD) [GS85], which generalizes the Gaussian distribution to multiple dimensions. For a n-dimensional vector of random variables $\mathbf{X}$ the probability density function is characterized by a

n-dimensional mean vector μ and a symmetric positive definite covariance matrix $\mathbf{\Sigma}$. To express the probability density of a multivariate Gaussian distribution we use the inverse of covariance matrix, called precision matrix $\mathbf{T} = \mathbf{\Sigma}^{-1}$, which will be helpful later during the learning phase. The probability density then can be written as follows[Nea03]:

$$\texttt{pdf}_{\mu,\sigma^2}(\mathbf{x}) = \frac{1}{(2\pi)^{n/2}(det(\mathbf{T}^{-1}))^{1/2}} exp\left(-\frac{1}{2}\Delta^2(\mathbf{x})\right), \tag{4.2}$$

where $\Delta^2(\mathbf{x}) = (\mathbf{x} - \mu)^T \mathbf{T}(\mathbf{x} - \mu)$.

A GBN is a BN where random variables X associated to each node in the network have associated a Gaussian distribution, conditional on their parents X_i.

Probit Distributions In order to smoothen the decisions in a NeuralP framework, we choose a function without plateaus and discontinuities. Since we operate with Gaussian random variables, the natural candidate is their *cumulative distribution function (CDF)*. This is an S-shaped function or a *sigmoid* (see Figure 4.1), whose steepness is defined by σ^2, where erf denotes the error function:

$$\texttt{cdf}_{\mu,\sigma^2}(x) = \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right), \tag{4.3}$$

For a particular value x of X, the function $\texttt{cdf}_{\mu,\sigma^2}(x)$ returns the probability that a random sample from the distribution $\mathcal{N}(\mu, \sigma^2)$ will belong to the interval $(-\infty, x]$.

Since the sensors and actuators of the Pioneer rover are noisy, the trajectories it follows are each time different from the optimal one (assuming that such difference is tolerated by the parking configuration), even if the optimal trajectory of the parking example is known. To be adaptive we use a combined approach: we incorporate probabilistic control structures in the program (introduced in the Section 4.2) and sample commands from a GBN, whose parameters were learned experimentally. To detect changes in the environment and get more accurate position estimates, data from various sensors are combined with a sensor fusion algorithm.

4.2 Key Components of Neural Programs

Traditional inequality relations (e.g. >, ≥, ≤, <) define sharp (or firm) boundaries on the satisfaction of a condition, and can therefore be seen as step functions (see Fig. 4.1). Using firm decisions in a program operating on Normal RVs cuts distributions in half, resulting in unnormalized and invalid PDFs; see Fig. 4.2. In this figure, the upper-right plot shows what happens to the PDF of a Normal RV after passing it through a traditional conditional statement (if or while). To avoid such situations and maintain valid probability density one needs to re-normalize the PDF.

To quantify uncertainty of making each decision in a CPS controller software, and avoid re-shaping its PDF each time an RV x is passed through a conditional, we introduce a new control structure called *neural if*, or **`nif`** for short. The name is coined to express the key novelty of our approach: we propose to use smooth conditionals $\texttt{cdf}_{\mu,\sigma^2}(x)$, which resemble a smooth neural activation function, instead of firm ones.

Let # range over the set of comparison operators $\{>, \geq, \leq, <\}$. `Nif` statements are of the form **`nif`**`(x # a,`σ^2`) S1` **`else`** `S2`, where `x # a` is a predicate and σ^2 is a variance. With every **`nif`** statement, we associate a boolean RV (also called `nif`) with the evaluation of `(x # a,`σ^2`)`. Specifically, if RV **`nif`**`(x # a,`σ^2`)` evaluates to 1 with variance σ^2, statement `S1` will be taken; otherwise `S2`.

The evaluation of **`nif`** statements is performed in two steps: (i) Find an $\mathcal{R}$-interval I representing the confidence in which `S1` will be taken. (ii) Check if a sample from the GD $\mathcal{N}(0, \sigma^2)$ falls within I. For the limit case where $\sigma^2 \to 0$ (no uncertainty), we require the **`nif`** statement to behave as a traditional `if` statement. Note that **`nif`** statements define a family of RVs parameterized by #, a, and σ^2.

To find the I in (i), we calculate the difference `diff(x,a)` between `x` and `a` as follows, where ϵ is a pre-defined small value:

$$\texttt{diff(x,a)} = \begin{cases} \texttt{x - a} - \epsilon & \text{if \# is } >, \\ \texttt{x - a} & \text{if \# is } \geq, \\ \texttt{a - x} - \epsilon & \text{if \# is } <, \\ \texttt{a - x} & \text{if \# is } \leq . \end{cases}$$

Informally, the confidence is characterized by the difference: the larger the value of `diff(x,a)`, the more confident we are about executing `S1`, and the larger the probability of executing `S1`. The probability of **`nif`**`(x # a,`σ^2`) = 1` (the probability of executing `S1`) is given by $\texttt{cdf}_{0,\sigma^2}(\texttt{diff(x,a)})$ and defines the interval $[q_1; q_2]$ by calculating two symmetric quantiles q_1 and q_2 such that:

$$\int_{q1}^{q2} \texttt{pdf}_{0,\sigma^2}(\texttt{x})dx = \texttt{cdf}_{0,\sigma^2}(\texttt{diff(x,a)}). \tag{4.4}$$

In step (ii), a random sample is taken from the distribution $\mathcal{N}(0, \sigma^2)$ and tested to see if it belongs to the confidence interval $[q_1; q_2]$. If it is within the interval, `S1` is executed; otherwise `S2` is executed. The probability to execute `S1` is influenced by the variance σ^2 (see Fig. 4.1). The dependence is twofold: `diff(x,a)` shows how confident we are in making the decision, and σ^2 characterizes the uncertainty.

For the case $\sigma^2 \to 0$, the **`nif`** statement is equivalent to the `if` statement. In this case, the PDF is expressed as a Dirac function $\delta(x)$, which essentially concentrates all of the probability density (PD) in a single point $x = 0$. Hence, the $\texttt{cdf}_{0,\sigma^2 \to 0}(x)$ becomes a step function (the bold black line in Fig. 4.1). The two possible cases for evaluating `nif` statements without uncertainty are: (i) `diff(x,a)` ≥ 0 and (ii) `diff(x,a)` < 0. In the first case, the probability of executing `S1` is equal to 1; hence the interval is $(-\infty; +\infty)$ and includes every sample. For the second case, the probability of taking `S1` is 0; hence the interval is empty and cannot contain any sample.

To illustrate the evaluation of **`nif`** statements, consider the following example, where `x`, `a` $\in \mathcal{R}$, and $\sigma^2 \in \mathcal{R}^+$.

```
nif( x >= a, σ²) S1 else S2
```

Suppose in the current execution `x` $= 1$ and `a` $= 0$. Figure 4.1 illustrates how decisions are made if σ^2 is $0.4^2, \pi, 4^2$. Since `diff(x,a) = 1`, the probability of executing `S1` is defined by $\texttt{cdf}_{0,\sigma^2}(1)$ and for the above cases is equal to 0.994, 0.714 and 0.599, respectively. The intervals I are [-1.095;1.095], [-1.890; 1.890], and [-3.357; 3.357]. In the second step, we sample from the GDs with the corresponding σ^2 ($\mathcal{N}(0, 0.4^2)$, $\mathcal{N}(0, \pi)$, and $\mathcal{N}(0, 4^2)$), and check if the value lies within the intervals. The plot in Fig. 4.1 shows the quantiles and the corresponding intervals I for various values of σ^2.

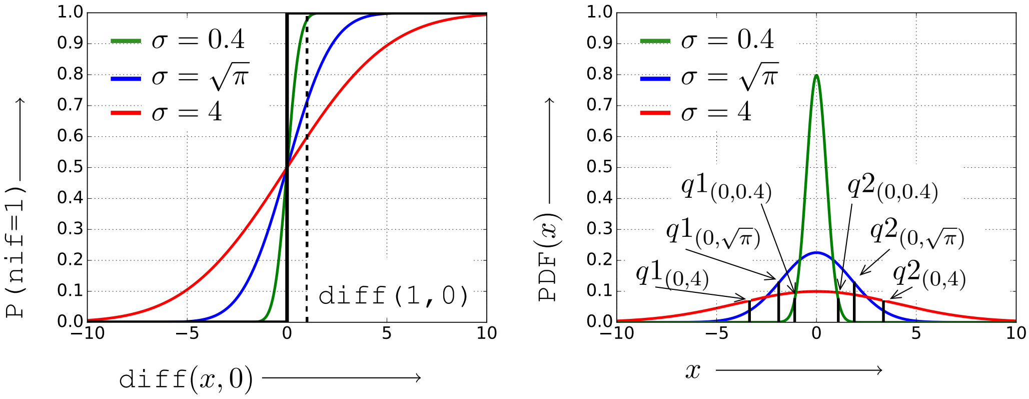
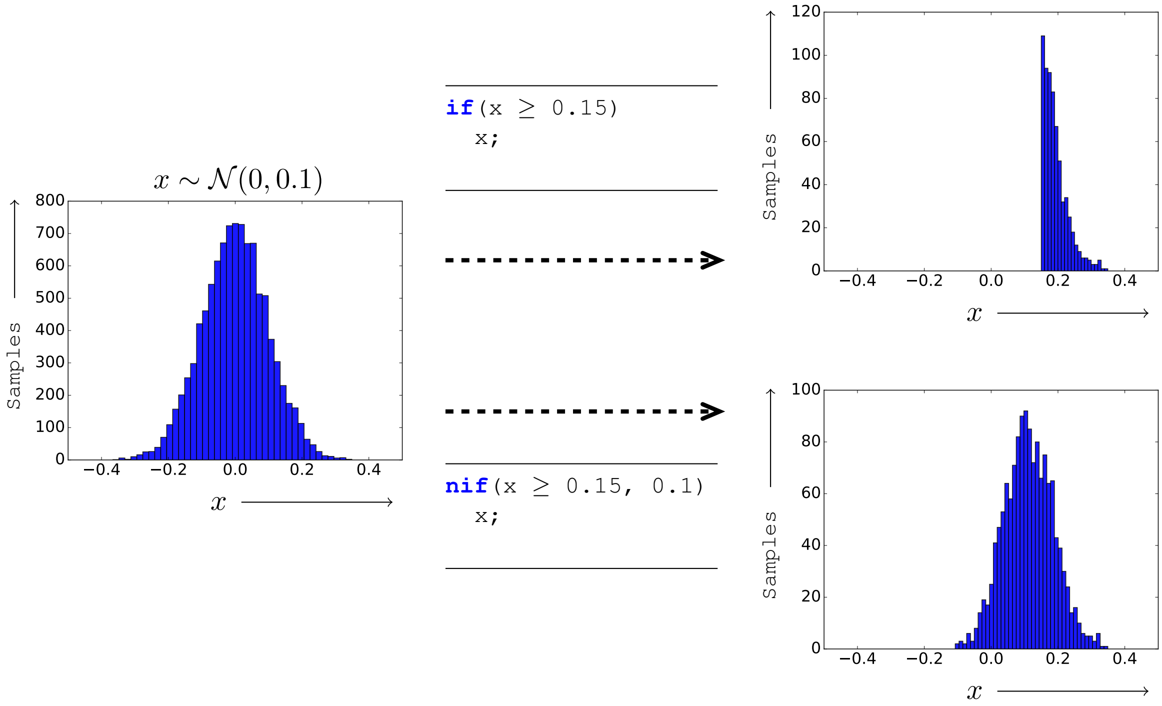


Figure 4.1: Left plot: "Soft" (colored lines) and "hard" (bold black line) thresholds; Right plot: PDFs and the quantiles for $x = 1$ and $a = 0$

So far we have been concerned with execution of single samples $x \sim \mathcal{N}(\mu, \sigma^2)$ on **`nif`** statements. Fig. 4.2 illustrates what happens at the distribution level; in particular, the difference between passing a GD RV `x` $\sim \mathcal{N}(0, 0.1)$ through the statements `if(x >= 0.15)` and **`nif`**`(x >= 0.15, 0.1)`. Since the input RV x has a GD, and a GD is used to evaluate the condition, the result is a product of two GDs, scaled by some constant factor k. Using our approach, the GD is not cut in undesirable ways (upper-right plot in Fig. 4.2), and maintains its GD form after passing the **`nif`** statement.

We can apply "soft" thresholds to loops. The *neural while* **`nwhile`**`( x # a,` σ^2`) {` P_1 `}` statement takes a predicate (`x` $\#$ `a`) and a variance σ^2 and executes the program P_1 according to the following rules: (1) Compute `diff`(`x` $\#$ `a`), find interval I and quantiles q_1, q_2 according to Eq. 4.4. (2) Check if a random sample $x \sim \mathcal{N}(0, \sigma^2)$ is within the interval $[q_1; q_2]$. (3) If the sample belongs to the interval, execute P_1 and go to step (1); else exit. The **`nwhile`** is an extension of the traditional `while` statement that introduces a quantitative measure of uncertainty of executing a loop iteration.



```
if(x ≥ 0.15)
  x;
```

```
nif(x ≥ 0.15, 0.1)
  x;
```

Figure 4.2: Passing RVs through conditions

Since the **nif** and **nwhile** statements subsume the behavior of traditional `if` and `while` statements (the case $\sigma^2 \to 0$), we use them to *define an imperative language with probabilistic control structures.* Binary operators *bop* (e.g. addition, multiplication), unary operators *uop* (negation), and constants c are used to form expressions E. A program P is a statement S or combination of statements.

$$
\begin{aligned}
E &::= \quad \mathtt{x}_i \mid c \mid bop(E_1, E_2) \mid uop(E_1) \\
S &::= \quad \mathtt{skip} \mid \mathtt{x}_i := E \mid S_1; S_2 \mid \\
&\qquad \mathtt{nif(}\, \mathtt{x}_i \# c, \sigma^2 \,\mathtt{)}\ S_1\ \mathtt{else}\ S_2 \mid \\
&\qquad \mathtt{nwhile(}\, \mathtt{x}_i \# c,\ \sigma^2 \mathtt{)\{}\ S_1\ \mathtt{\}}
\end{aligned}
$$

To illustrate CPS controller design using the proposed language, we consider the first case study involving the parallel parking of a mobile robot. The basic operations required are: Go backwards up to a point l_1, turn up to an angle α_1, go backwards up to l_2, turn up to α_2, and go backwards up to l_3. The control-program skeleton for this application can be specified as a sequence of **nwhile** statements, as shown in Listing 4.1.

The versatility of this approach is that the program skeleton is written only once and constitutes and infinite number of controllers (when the distance l_1 is smaller then the target, the next locations to be visited will be re-sampled in order to compensate for the difference, as discussed in Sections 4.3 and 4.4). The question we next need to answer is:

What are the distances and turning angles for each action and how uncertain are we about each of these parameters?

To find the unknown parameters in Listing 4.1, namely the target locations and variances, we use the learning procedure described in Section 4.3.

Listing 4.1: Parallel parking program skeleton

```
nwhile(currentDistance < targetLocation1, sigma1){
  moving();
  currentDistance = getPose();
  }
updateTargetLocations();
nwhile(currentAngle < targetLocation2,    sigma2){
  turning();
  currentAngle = getAngle();
  }
updateTargetLocations();
nwhile(currentDistance < targetLocation3, sigma3){
  moving();
  currentDistance = getPose();
  }
updateTargetLocations();
nwhile(currentAngle < targetLocation4,    sigma4){
  turning();
  currentAngle = getAngle();
  }
updateTargetLocations();
nwhile(currentDistance < targetLocation5, sigma5){
  moving();
  currentDistance = getPose();
  }
```

4.3 Bayesian-Network Learning

In the proposed approach, we start by defining a neural program as a skeleton, and in the next step we identify the corresponding GBN and learn its parameters. To illustrate the process on CPS controller design, we elaborate on the first case study, which is performing parallel parking of a mobile robot.

Parking can be seen as a sequence of moves and turns, where each action depends on the previous one: e.g. the turning angle typically depends on the previously driven distance. Due to sensor noise and imprecision, inertia and friction forces, and also many possible ways to perform a parking task starting from one initial location, we assume that the dependence between actions is probabilistic, and in particular, the RVs are distributed according to GD. We represent the dependencies between actions as the GBN in Figure 4.3, where l_i or α_j denotes a distance or a turning angle of the corresponding action and b_{ij} is a conditional dependence between consecutive actions.

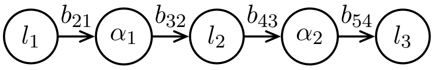


Figure 4.3: Gaussian Bayesian Network for parking

In order to learn the CPD of the GBN in Figure 4.3, and to fill in the `targetLocations` and the `sigmas` in Listing 4.1, we record trajectories of the successful parkings done by a human expert. Figure 4.4 illustrates a set of trajectories used during the learning phase. Informally, given an expert knowledge and a program skeleton, in the first step we learn from experience of a domain expert to give estimates of unknown parameters.

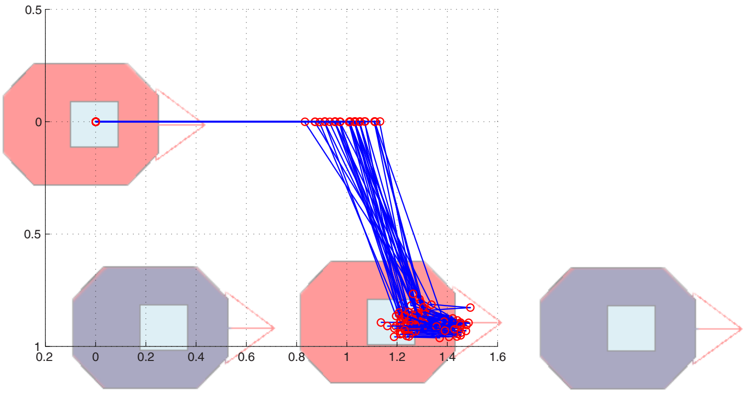


Figure 4.4: Example trajectories for the parking task

We than use the fact that any GBN can be converted to a MGD [Nea03] in our learning process. Identifying the parameters of the GBN can be divided into three steps:

1. Convert the GBN to the corresponding MGD,
2. Update the precision matrix $\mathbf{T} = \Sigma^{-1}$ of the MGD,
3. Extract σ^2s and conditional dependences from $\mathbf{T}$.

1. Conversion step. To construct MGD one needs to obtain the mean vector μ and the precision matrix $\mathbf{T}$. The mean vector μ comprises the means of all the variables from the GBN. To find the symbolic form of the precision matrix, we use the recursive notation from [HG95], where values of the coefficients b_i, will be learned in the update step below.

$$\mathbf{T}_{i+1} = \begin{pmatrix} \mathbf{T}_i + \frac{\mathbf{b}_{i+1}\mathbf{b}_{i+1}^T}{\sigma_{i+1}^2} & -\frac{\mathbf{b}_{i+1}}{\sigma_{i+1}^2} \\ -\frac{\mathbf{b}_{i+1}^T}{\sigma_{i+1}^2} & \frac{1}{\sigma_{i+1}^2} \end{pmatrix} \tag{4.5}$$

In order to apply Equation 4.5 we define an ordering starting with the initial node l_1, which precision matrix is $\frac{1}{\sigma_1^2}$. The vector $\mathbf{b}_{i+1}$ in Equation 4.5 comprises dependence coefficients for node i on all its immediate parents it in the ordering.

Since each action in the parking task depends only on the previous one, we can generalize the precision matrix for the arbitrary number of moves. For a GBN with k moves, all non-zero elements of the precision matrix $\mathbf{T} \in \mathcal{R}^{k;k}$ can be found according to the Equation 4.6, where $\mathbf{T}(r, c)$ is a c-th element in a r-th row of the precision matrix:

$$\begin{aligned} \mathbf{T}(i, i-1) &= -\frac{b_{i(i-1)}}{\sigma_i^2}, \text{where } i \in \{2, .., k\}; \\ \mathbf{T}(i, i) &= \frac{1}{\sigma_i^2} + \frac{b_{(i+1)i}^2}{\sigma_{i+1}^2}, \text{where } i \in \{2, .., k\}; \\ \mathbf{T}(i, i+1) &= -\frac{b_{(i+1)i}}{\sigma_{i+1}^2} \text{where } i \in \{1, .., k\}. \end{aligned} \tag{4.6}$$

Following this approach, the precision matrix for the last action of parking controller has the following form (Equation 4.7):

$$\mathbf{T}_5 = \begin{pmatrix} \frac{1}{\sigma_1^2} + \frac{b_{21}^2}{\sigma_2^2} & -\frac{b_{21}}{\sigma_2^2} & 0 & 0 & 0 \\ -\frac{b_{21}}{\sigma_2^2} & \frac{1}{\sigma_2^2} + \frac{b_{32}^2}{\sigma_3^2} & -\frac{b_{32}}{\sigma_3^2} & 0 & 0 \\ 0 & -\frac{b_{32}}{\sigma_3^2} & \frac{1}{\sigma_3^2} + \frac{b_{43}^2}{\sigma_4^2} & -\frac{b_{43}}{\sigma_4^2} & 0 \\ 0 & 0 & -\frac{b_{43}}{\sigma_4^2} & \frac{1}{\sigma_4^2} + \frac{b_{54}^2}{\sigma_5^2} & -\frac{b_{54}}{\sigma_5^2} \\ 0 & 0 & 0 & -\frac{b_{54}}{\sigma_5^2} & \frac{1}{\sigma_5^2} \end{pmatrix} \tag{4.7}$$

2. Update step. Once we derived the symbolic form of the precision matrix, we use the training set in order to learn the actual values of its parameters, as described in the algorithm from [Nea03]. Each training example $\mathbf{x}^{(i)}$ corresponds to a vector of lengths and turning angles for a successful parking task performed by a human expert. The total number of examples in the training set is M. The procedure allows us to learn iteratively and adjust the prior belief by updating the values of the mean μ and covariance matrix β of the prior, where v is a size of a training set for the prior belief, and $\alpha = v - 1$.

$$\beta = \frac{v(\alpha - n + 1)}{v + 1}\mathbf{T}^{-1}, \tag{4.8}$$

The updated mean value μ^* incorporates prior value of the mean μ and the mean value of the new training examples $\mathbf{x}$. In this way the CPS controller can adapt, if the environment conditions changed, since the new samples will affect the parameter values learned so far.

$$\begin{aligned} \overline{\mathbf{x}} =& \frac{\sum_{i=1}^{M} \mathbf{x}^{(i)}}{M} \\ \mu^* =& \frac{v\mu + M\overline{\mathbf{x}}}{v+M} \end{aligned} \tag{4.9}$$

The size of the training set v^* is updated to its new value:

$$v^* = v + M \tag{4.10}$$

The updated covariance matrix β^* combines the prior matrix β with the covariance matrix of the training set $\mathbf{s}$:

$$\begin{aligned} \mathbf{s} =& \sum_{i=1}^{M} \left(x^{(i)} - \overline{\mathbf{x}}\right) \left(x^{(i)} - \overline{\mathbf{x}}\right)^T \\ \beta^* =& \beta + s + \frac{rm}{v+M} \left(x^{(i)} - \overline{\mathbf{x}}\right) \left(x^{(i)} - \overline{\mathbf{x}}\right)^T \end{aligned} \tag{4.11}$$

Finally, the new value of the matrix β is used to calculate the covariance matrix $(\mathbf{T}^*)^{-1}$, where $\alpha^* = \alpha + M$.

$$(\mathbf{T}^*)^{-1} = \frac{v^* + 1}{v^*(\alpha^* - n + 1)} \beta^* \tag{4.12}$$

3. Extraction step. The new parameters of the GBN can now be retrieved from the updated mean vector μ^* and from $(\mathbf{T}^*)^{-1}$. If new traces are available at hand, one can update the distributions by recomputing μ^* and $(\mathbf{T}^*)^{-1}$ using Equations 4.9-4.12. Unknown parameters from the program skeleton are learned from successful traces and these dependencies are used during the execution phase to sample the commands.

4.4 Case studies

We illustrate the proposed framework on the following case studies: (i) parallel parking of a Pioneer Rover, and (ii) simulation of tap withdrawal neural circuit of a nematode, (iii) controller for parallel parking as a neural circuit.

4.4.1 Parallel parking

We performed the experiments on a `Pioneer P3AT-SH` mobile rover from Adept MobileRobots[Ade] (see Figure 4.5). The rover uses the `Carma Devkit` from SECO with Tegra 3 ARM CPU & GPU running the ROS on Ubuntu 12.04.

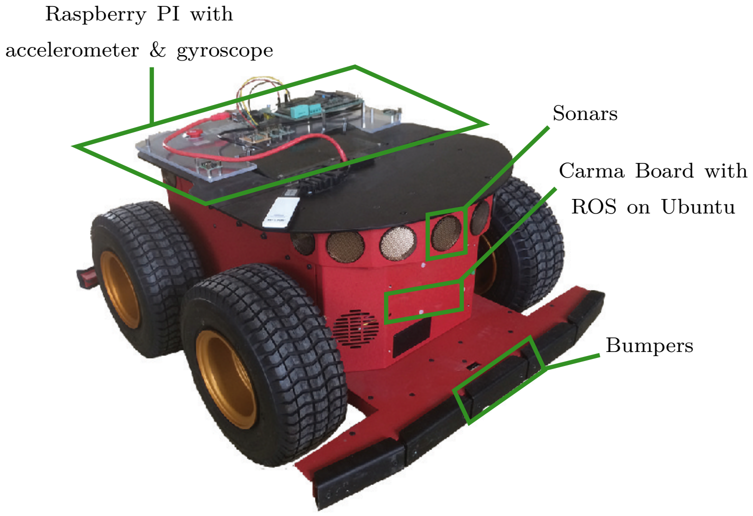


Figure 4.5: Experimental platform: Pioneer Rover

We use the following dymanics model of the rover for estimating the state of the robot:

$$x(k+1) = x(k) + v(k)\Delta t cos\left[\Theta(k+1)\right] + w_x \Delta t \tag{4.13}$$

$$y(k+1) = y(k) + v(k)\Delta t sin\left[\theta(k+1)\right] + w_y \Delta t \tag{4.14}$$

$$\Theta(k+1) = \Theta(k) + \omega\Delta t + w_\Theta \Delta t \tag{4.15}$$

$$v(k) = \frac{\omega_L(k) r_w + \omega_R(k) r_w}{2} + a\Delta t \tag{4.16}$$

$$\omega(k) = \frac{\omega_L(k) r_w - \omega_R(k) r_w}{R_b}, \tag{4.17}$$

where w_x, w_y and w_Θ are zero mean Gaussians taking into account noise during the observations; ω_L and ω_R are angular velocities of left and wheels respectively, r_w is a radius of a wheel and R_b is a base (0.3 m).

Structure of the Parking System

The parking system can be separated into several building blocks (see Figure 4.6). The *Rover Interface* block senses and controls the rover, that is, it establishes an interface to the hardware. The block *Sensor Fusion* takes the sensor values from the *Rover Interface* block, and provides the estimated pose of the rover to the high-level controller *Engine*.

The *Engine* uses the GBN block to update the motion commands based on the estimated pose. Furthermore, the *Engine* maps the (higher level) motion commands to velocity commands needed by the *Rover Interface* to control the rover.

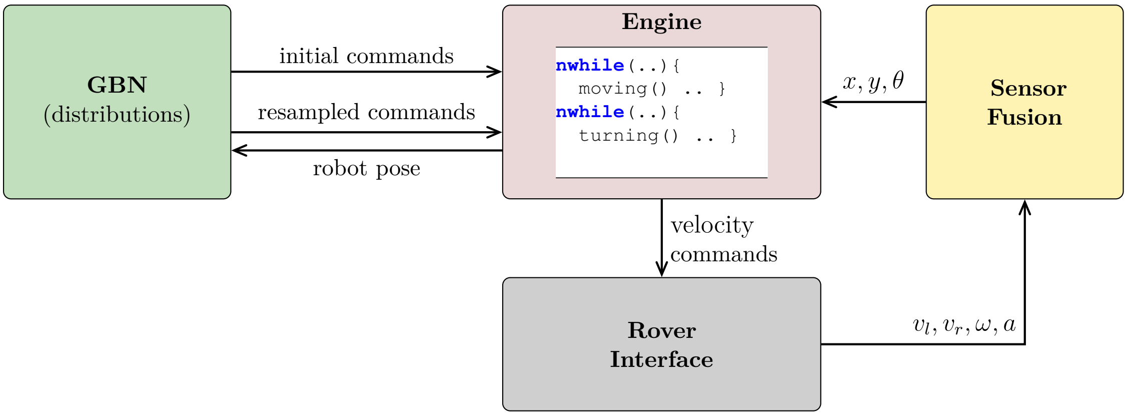


Figure 4.6: Parking system architecture

The Gaussian Bayesian Network Block The goal of the GBN block in Figure 4.6, is to generate motion commands for the Engine to execute. A motion command corresponds to a driving distance or a turning angle.

The distribution of the first move l_1 is independent from any other move and has the form $\mathcal{N}(\mu_1, \sigma_1^2)$. Starting from the second move α_1, each motion depends on the previous one: For motion number n, the distribution has the form $\mathcal{N}(\mu_n - b_{n,n-1} * x_{n-1}^{\text{sampled}}, \sigma_n^2)$. The initial command vector is obtained by sampling from l_1, and each subsequent command vector is obtained by taking into account the previous sample.

As the rover and its environment are uncertain (i.e., sensors are disturbed by noise) we use the pose provided by the sensor fusion unit to update the motion commands. Hence the motion commands are constantly adapted to take into account the actual driven distance (which could be different from the planned one due to the aforementioned uncertainty of the CPS). This allows us to incorporate the results of the sensor fusion algorithm in the updated commands.

The Engine Block During the run we execute a motion command according to the semantics of the `nwhile` loop. In particular, the estimated pose is passed from the Sensor Fusion block to the Engine and compared with the target location, specified as a point on a 2-D plane. Since the rover is affected by noise its path can deviate and never come to the target location. To be able to detect and overcome this problem we estimate the scalar product of two vectors: The first one is the initial target location, and the second one is the current target location. This product becomes negative after passing the goal even on a deviating path. In an `nwhile` statement we check the distance (or angle) and detect if we should process the next command. After executing each command we resample the pose to take into account actual driving distance in subsequent moves.

The Rover Interface Block The block *Rover Interface* implements the drivers for sensors and actuators. The *wheel velocities* are measured by encoders, already supplied within the Pioneer rover. A built-in microcontroller reads the encoders and sends their value to the `Carma Devkit`. Additionally the rover is equipped with an inertial measurement unit (IMU) including an accelerometer, measuring the linear acceleration, and a gyroscope, measuring the angular velocity of the rover. The `Raspberry Pi` samples the accelerometer and gyroscope, and forwards the raw IMU measurements to the `Carma Devkit`. The rover is controlled by the incoming velocity commands.

The Sensor Fusion Block The current pose is observed by sensors, which suffer from uncertainty. Measurements are distorted by noise, e.g., caused by fluctuations of the elements of the electrical circuit of the sensors. The environment may be unpredictable, e.g., the rover may slip over water or ice when parking. To overcome such problems sensor fusion techniques are applied, i.e., several sensors are combined to estimate a more accurate state. A common method is state estimation (also called *filtering*) [Mit07, TBF06]. In this application, an unscented Kalman filter (UKF) [WVdM00] is used. This filter combines the measurements from sensors with a dynamics model describing the relations from the measured variables to the pose of the rover.

We implemented the architecture described above using ROS [QCG+09], which is a meta-operating system that provides common functionality for robotic tasks including process communication, package management, and hardware abstraction. The system components (see Fig. 4.6) are implemented as ROS-nodes, which communicate with each other by passing messages. The proposed framework differs from existing approaches described in Section 3.1 in a way that the GBN block uses information about the current state of the rover provided by a sensor fusion to issue a command which takes into account the environment.

Parameters of the GBN

After the learning phase, we obtain the parameters of the GBN that we use in the program skeleton (Table. 4.1). Since we track the position using the data from the sensor fusion and each movement has the experimentally learned uncertainty, we are resistive to the perturbation of the actual driving distances and angles.

4.4.2 Tap withdrawal circuit of C.elegans

The proposed framework and introduced control structures are general enough to be applied for modelling and simulations of biological systems. In the second case study we illustrate how using NeuralP for simulation of neural activity can facilitate modelling individual synaptic connections, and providing consistent results on average. We choose

–		b_{21}	0.7968	b_{32}	-0.2086	b_{43}	0.5475
σ_1^2	0.0062	σ_2^2	0.0032	σ_3^2	0.0019	σ_4^2	0.022

b_{54}	-0.0045	b_{65}	1.1920	b_{76}	-0.0968		
σ_5^2	0.0008	σ_6^2	0.0178	σ_7^2	0.0013		

Table 4.1: BN variances and coefficient dependences

to model tap withdrawal neural circuit of *Caenorhabditis elegans* (C. Elegans), a model organism, whose neural topology is known and does not change among adult species.

Wicks et al. in [WRR96] presented a neural model of a tap withdrawal circuit (Fig. 4.7). Three mechanosensory neurons (`AVM, PLM, ALM`) can be stimulated with external current I_{stim}. The fourth sensory neuron `PVD` reacts on harsh touch or cold temperatures and is not stimulated in our case. The stimuli propagate through the network (see Fig. 4.7) via currents towards the cells responsible for the locomotion: `PVC` and `AVB` for the forward movement; `AVA` and `AVD` for the backward movement. `DVA` plays the role in maintenance of the overall activity of the circuit.

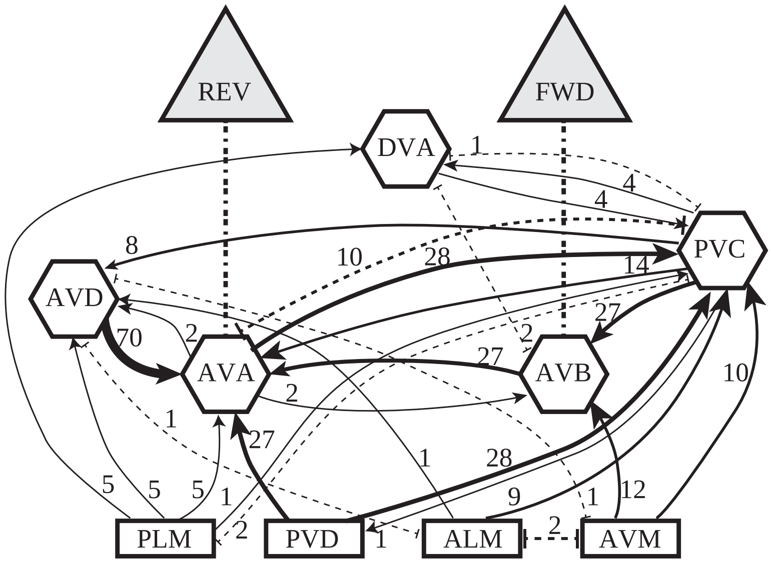


Figure 4.7: Tap withdrawal circuit of C. Elegans

The model by Wicks [WRR96] defines membrane potentials and currents of a neuron i according to equations 4.18 - 4.21:

$$\frac{dV^{(i)}}{dt} = \frac{V_{Leak} - V^{(i)}}{R_m^{(i)} C_m^{(i)}} + \frac{\sum_{j=1}^{N} (I_{syn}^{(ij)} + I_{gap}^{(ij)}) + I_{stim}^{(i)}}{C_m^{(i)}} \tag{4.18}$$

$$I_{gap}^{(ij)} = w_{gap}^{(ij)}\, g_{gap}^{(ij)}\, (V_j - V_i) \tag{4.19}$$

$$I_{syn}^{(ij)} = w_{syn}^{(ij)}\, g_{syn}^{(ij)}(V^{(i)})\, (E^{(ij)} - V^{(j)}) \tag{4.20}$$

$$g_{syn}^{(ij)}(V^{(j)}) = \frac{\bar{g}_{syn}}{1 + e^{K\left(\frac{V^{(j)} - V_{eq_j}}{V_{range}}\right)}} \tag{4.21}$$

Change of a membrane potential $dV^{(i)}$ of the neuron i is affected by all the currents flowing into the neuron, current potential $V^{(i)}$ and leakage potential V_{Leak} of the cell. Resistance $R_m^{(i)}$ and conductance $C_m^{(i)}$ of a membrane of each neuron characterize the strength of a dependence on voltage and current. The neurons have two types of connections: synaptic and gap junction (respectively solid and dashed lines on Fig. 4.7). Gap junction current $I_{gap}^{(ij)}$ from a neuron j to a neuron i is characterized by a difference of membrane potentials, constant conductance $g_{gap}^{(ij)}$, and number of gap junctions $w_{gap}^{(ij)}$. This current characterises instantaneous resistive connection between neurons and comprises a linear combination of inputs. Synaptic current $I_{syn}^{(ij)}$ is of chemical nature, characterized by a weight $w_{syn}^{(ij)}$ or a number of synaptic connections from neurons i to j, conductance $g_{syn}^{(ij)}(V^{(i)})$ and difference of potentials. Synaptic conductance $g_{syn}^{(ij)}(V^{(i)})$ has a sigmoid shape, parametrized by constant K, presynaptic voltage range V_{range}, and equilibrium V_{eq}.

The output of Wicks' model characterizes the direction of movement, either *forward* or *backward* after applying stimulus current to mechanosensory neurons. The continuous model above shows an average behavior of the circuit. In [DS13] the authors claim that biological processes possess inherent stochasticity and should be modeled using appropriate tools. Given the network structure on Fig. 4.7 we rewrote differential equations as a neural program and captured behavior of each synaptic connection (Eq. 4.21) with a `nif` statement: we also claim our method better reflects the reality since a decision for each synaptic connection if it is open or closed is made probabilistically based on a semantics of a `nif` statement and takes into account inherent noise in each neuron. Two simulations produce on average the same results (Table 4.2), but for the case of NeuralP the simulation incorporates a probabilistic choice of each synapse to release neuro-transmitter, which is closer to biophysical description of synapse function.

The listing 4.2 presents the simulation of the tap withdrawal circuit of C. Elegans in a NeuralP framework. For simplicity the computation only for one neuron is shown, and all the neurons from Fig. 4.7 compute in parallel. For each time step we compute currents flowing from one neuron to another and the difference of membrane potentials. Whenever the exact number of operations is required, we exploit the limit case of `nwhile` statement with zero uncertainty. Table 4.2 shows the difference between ODE simulation and average output of a neural program, and time for executing a simulation in MATLAB and C++. The row $*$ in Table 4.2 denotes stimulation of all three input neurons.

Listing 4.2: C. Elegans tap withdrawal simulation as NeuralP

```
nwhile( $t \leq t_{dur}$,  0){
  compute $I_{gap}^{(ij)}$ using Equation 4.19
  nwhile( $k \leq w_{syn}^{(ij)}$,  0){
    nif($V^{(j)} \leq V_{eq}$, $K/V_{range}$){
    $g_{syn}^{(ij)} \leftarrow g_{syn}^{(ij)} + g_{syn}$
    }
  }
  compute $I_{syn}^{(ij)}$ using Equation 4.20
  compute $dV^{(i)}$ using Equation 4.18
  $V^{(i)} \leftarrow V^{(i)} + dV^{(i)}$
  $t \leftarrow t + dt$
}
```

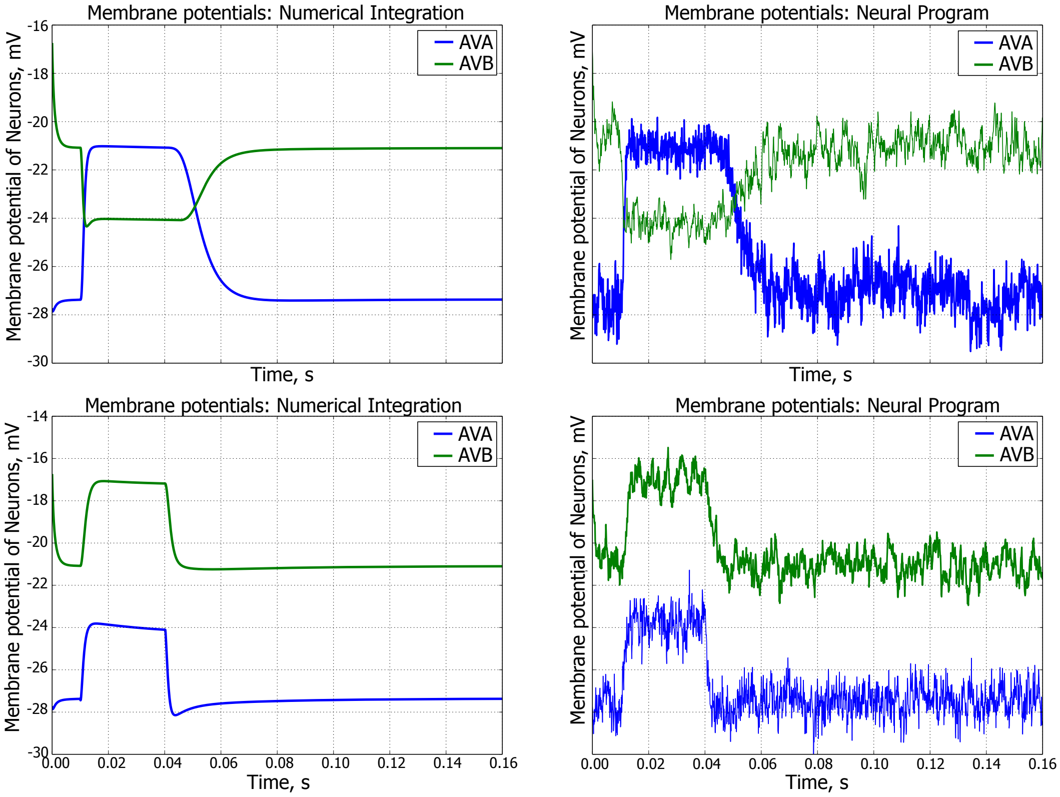


Figure 4.8: Deterministic simulations of potentials of `AVA` and `AVB` neurons (left); stochastic simulations using `nif` condition (right)

NEURON	ODE.OUT	AV.OUT	DIFF.%	t_{MATLAB}	$t_{\text{C++}}$, s
`AVM`	-0.0326	-0.0330	1.35	59.06	0.395
`ALM`	-0.0314	-0.0320	2.00	59.06	0.403
`PLM`	0.0807	0.0811	0.38	59.04	0.418
$\star$	-0.0272	-0.0266	2.25	60.12	0.420

Table 4.2: Comparison of simulation results

4.4.3 Reflections: Parallel Parking as a Neural Circuit

In this section we draw analogy between the two case studies and express an adaptive proportional controller for parallel parking as a neural circuit. We define the following variable mapping between the two case studies: (i) voltages $V^{(i)}$ from the second case study are mapped to the distances x and turning angles θ in the first case study; (ii) synaptic currents $I_{syn}^{(ij)}$ from the neural circuit are mapped to the linear v and angular velocities ω in the first case study; (iii) synaptic conductance $g_{syn}^{(ij)}$ defines a control flow in parallel parking controller; (iv) synaptic potentials $E^{(ij)}$ are mapped to the target locations (thresholds) $l^{(i)}$ and $\alpha^{(i)}$. Neurons have only synaptic connections (see Fig. 4.9).

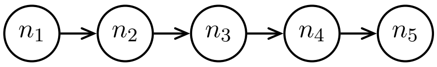


Figure 4.9: Controller for parallel parking as a neural circuit

Linear and angular velocities are defined as follows:

$$\frac{dx^{(1)}}{dt} = w^{(1)}(l^{(1)} - x^{(1)}) \tag{4.22}$$

$$\frac{d\theta^{(i)}}{dt} = w^{(i,i-1)}\ g^{(i,i-1)}(x^{(i)})\ (\alpha^{(i)} - \theta^{(i)}) \tag{4.23}$$

$$\frac{dx^{(i)}}{dt} = w^{(i,i-1)}\ g^{(i,i-1)}\ (\theta^{(i)})(l^{(i)} - x^{(i)}) \tag{4.24}$$

$$g^{(i,i-1)}(\theta^{(i)}) = \text{nif}(\theta^{(i)} > \alpha^{(i)}, \sigma_i^2) \quad 1 \quad \text{else} \quad 0 \tag{4.25}$$

The circuit (see Fig. 4.9) acts as a proportional controller with proportionality constant $w^{(i,i-1)}$: the velocity of the rover is proportional to the difference between current and target locations (i.e. the error). A neuron i executes its corresponding control policy (Eq. 4.22 for the first neuron, Eq. 4.23 and Eq. 4.24 for even and odd neurons respectively). The controller operates as follows: at a starting time point the neuron n_1 is active. Each neuron represents the one motion primitive in parallel parking procedure. First neuron is active until $(x_1 > l_1, \sigma_1^2)$ evaluates to 0 according to `nif` semantics, and then $g^{(21)}$ fires the second neuron. The second neuron stays active until $\text{nif}(\theta_1 > \alpha_1, \sigma_2^2)$ evaluates to 0 and so on. A neuron i is activated by synaptic conductance $g^{(i,i-1)}$ and stays active until the rover reaches the corresponding target location up to a variance σ_i^2.

4.5 Summary

In this chapter we showed how taking inspiration from neural smooth activation functions facilitates CPS controller design in *neural programming framework* (NeuralP). The neural programming constructs and an underlying network facilitate writing robust and adaptive CPS controllers. The key of the framework is: (i) the use of smooth probability distributions in conditional and loop statements, instead of their classic stepwise counterparts; and (ii) the use of Gaussian Bayesian networks for capturing the dependencies among probability distributions.

We validated the utility of NeuralP by developing a CPS controller for a robot that is able to adapt to changing environment. Simulation of the tap-withdrawal response of *C. elegans* as an NeuralP program yields the same result as the differential-equations model and allows one to directly address the stochastic nature of each synaptic connection between two neurons. We also showed analogy between case studies by expressing a controller of the first problem as a neural circuit.

CHAPTER 5

Neural Models for Qualitative Monitoring

In the previous chapter we have identified how to use smooth sigmoidal functions incorporated in programming structures for designing CPS controllers. We developed the approach, where the controllers are defined as a program skeleton and a corresponding GBN, where the parameters of the latter can be learned from good executions.

To build safe, fault-tolerant CPS not only control, but also monitoring the system at runtime is required. In this chapter we tackle research questions **RQ 2** and **RQ 4**, as defined in Chapter 1, and develop a method to use neural models to check qualitatively, at runtime, whether a predefined safety properties are fulfilled.

We use the versatile TrueNorth neuron model, which was proposed in [CMA^{+}13] and reviewed in Section 2.1, as the main computational component for our monitors. In a series of papers [CMA^{+}13, EAA^{+}13, ADR^{+}13], IBM has revealed its novel, brain-inspired TrueNorth hardware architecture. The advantage of the TrueNorth model, is that it (i) incorporates temporal dimension in the function of neurons, (ii) employ operations, that can be efficiently mapped to digital hardware (e.g. FPGA). We believe that brain-inspired architectures are a novel dimension of technological development with a great research potential, and it is of importance to show, how such architectures and models can be used to perform a numerous tasks in CPS infrastructure, including monitoring.

However, neither neuro-synaptic hardware nor its simulation environment are available today for public use. In this work we implemented the spiking TrueNorth neural model from [CMA^{+}13] in C++, and make it available to the research community. We also show how the deterministic part of the model can be used to monitor MTL properties, translated to synthesizable HDL code, and deployed in FPGA.

The universality of the TrueNorth model is particularly appealing: It allows both deterministic and stochastic computation, depending on the neurons' configuration. This chapter is dedicated to show how the TrueNorth model can be applied for runtime monitoring of MTL properties. Having identified how to recognize MTL operators with TrueNorth, we are able to build neural monitors from MTL formulae. This is a first step towards building hardware monitors that provide both qualitative and quantitative information, with regard to a mixed-signal and a STL specification.

The contributions of this chapter can be summarized as follows:

1. We present a view of the MTL runtime-monitoring problem in terms of spiking neurons and their circuits, and provide the complete flow from the C++ implementation of neural circuits to synthesizable in FPGA monitors.
2. We demonstrate the usefulness of the proposed approach on a case study, the launching of a missile from battle ship. We consider this as a proof of concept, that will be used in industrial case study and evaluation in Chapter 8.

The rest of the chapter is organized as follows: Section 5.1 formulates the problem and presents the hardware generation flow. Section 5.2 focuses on applying the TrueNorth model for monitoring MTL specifications. Section 5.3 presents the case study and the experimental results. Section 5.4 gives a summary of the chapter.

5.1 Qualitative Monitoring with the TrueNorth model

The TrueNorth neural model has both deterministic and stochastic modes of operation, in this chapter we consider only its deterministic part. The model extends all the stages of the LIFmodel and operates in purely digital fashion. The reader is referred to Section 2.1 or [CMA$^+$13] for model description. We now formulate the problem and present the top-level view of the solution.

5.1.1 Problem Definition

Let $\mathbb{T}$ denote a discrete finite time interval $[0, r] \cap \mathbb{N}$, $\mathbb{B} \in \{0, 1\}$. A Boolean signal is a function $w : \mathbb{T} \to \mathbb{B}^n$. A TrueNorth neuron $n_i = (\texttt{params}_i, V_i, s_i, l_i)$ is characterised by (i) a set of parameters $\texttt{params}_i = \{\alpha_i, \beta_i, R_i, \lambda_i, \varepsilon_i, \kappa_i, \mathbf{s}_i, M\}$, (ii) its membrane potential V_i, (iii) a binary spike output s_i, and (iv) a label $l_i \in \{\texttt{in}, \texttt{interm}, \texttt{out}\}$ of a neuron. For a neural network $N = \{n_1, \cdots, n_m\}$ of m neurons we define its computation C over a time interval $[0, r]$ in the following way: $C : [0, r] \mapsto V^m \cup S^m$, where $V^m = \{V_1, \cdots, V_m\}$ and $S^m = \{s_1, \cdots, s_m\}$ are membrane potentials and spikes of neurons in the network N respectively. Given a MTL specification φ and a signal w, out task is to devise a monitor that delivers a verdict whether w violates φ. The architecture of the monitor is a neural circuit N of TrueNorth neurons, with the labelled `out` neuron manifesting the formula satisfaction via sequence of spikes. Moreover, the resulted monitor should allow synthesizable in FPGA hardware realization.

We use neurons to recognize sub-formulae of φ: a neural circuit then represents a monitor. At each time step a TrueNorth neuron either spikes or not, which can be seen as outputting a binary signal. Since neurons compute on outputs of parent neurons, we can supply an external signal w into the circuit using special "input" neurons (which have label `in`, `MockTrueNorthNeuron`'s in our implementation) which behave in accordance with w. At each time step the neural monitor accepts w as input and outputs a SPIKE if the specification has been fulfilled. The absence of a SPIKE at the output of the neural monitor corresponds to a specification violation.

In order to build neural monitors for MTL specifications using TrueNorth, we have to be able to express logical and (bounded) temporal operators in terms of the model [CMA+13].

5.1.2 Solution: Top Level View

In this section we describe the flow from getting a natural language specification of a property to a hardware TrueNorth neural monitor in an FPGA (see Fig. 5.1).

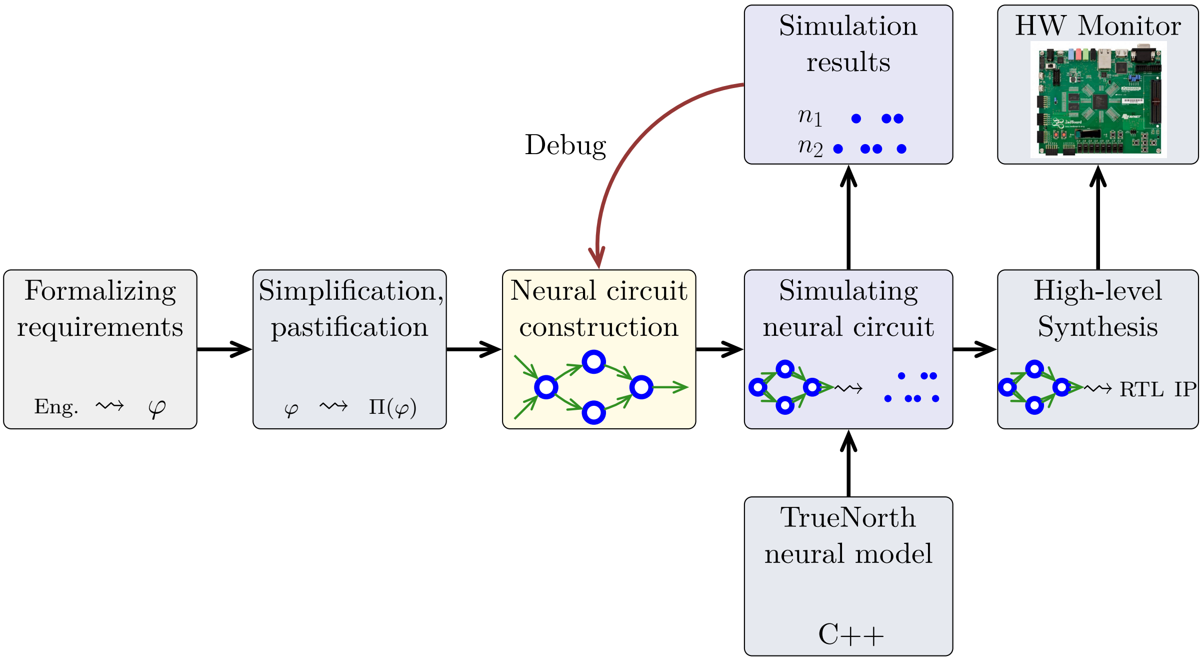


Figure 5.1: Neural Monitor Generation Flow

Formalizing requirements

In the first step natural-language requirements must be converted to an MTL formula to eliminate ambiguity and possible misinterpretations. It is a non-trivial problem how to devise a formula from a textual description and still an open issue [CRST09, VBG06]. To express mutual dependencies and temporal behavior we rely on expert knowledge, who translates requirements to a formal language. Requirements engineering [vL09] is a broad research subject on its own, and is beyond the scope of the thesis. We assume that for a CPS or its sub-parts requirements are already defined. The formalization step is illustrated in the case study in Section 5.3 and elaborated in more detail in Chapter 8.

Pastification, Simplification

As we are able to devise a verdict based on the behavior that has already happened, every bounded future formula must be pastified to obtain an equisatisfiable formula containing only past operators. Duplicate sub-trees in a formulae parse tree are eliminated. This pre-processing step is necessary for hardware realization to make the resulting resource-efficient. We use two-step pastification procedure from [MNP07]: (i) compute temporal depth D of the formula and (ii) convert a bounded-future formula to a past one.

Constructing a neural monitor

We then define a circuit topology based on the formulae from the previous step, and instantiate the parameters of TrueNorth neurons that will correspond to a monitor of a past-MTL[1] specification. We use configuration parameters for logical and temporal operators from Section 5.2 and traverse the parse tree of the formulate replacing each node in the tree with a corresponding neural sub-monitor.

Circuit simulation

We implemented the model described in Section 2.1.1 in C++ and use this implementation to simulate circuits of TrueNorth neurons (available in [sou]). Each neuron is an instance of the `TrueNorthNeuron` class, with pre-defined parameter values and neurons' connections. We then test the circuit on the external input and observe outputs for every neuron, which is useful for debugging and attesting neurons' configurations. The results of the simulation are stored in text files, which are then visualized.

HLD code generation with High-Level Synthesis

Given the circuit from the simulation step and C++ implementation of TrueNorth model, our task in this step is to obtain an FPGA-ready synthesizable representation of a neural monitor. This is the final step of hardware implementation of a neural monitor. We use the HLS from Xilinx [Inc] to convert a C++ description of a neural circuit to a synthesizable HDL code (Verilog or VHDL). Given a C++ function that represents a monitor for a sub-formula of the initial specification, we generate HDL code with HLS according to the following steps: 1) Define a test suite to compare a function that uses hardware-specific (bit-precise) data types with a "golden" function that uses generic software types; 2) Generate HDL with RTL synthesis. 3) Compare the functionality of the synthesized code from 2) with the "golden" function. 4) Export the synthesized

---

[1] We refer to a past-MTL formula as a formula with past-temporal operators only

code as an intellectual property (IP) (available upon request). These IPs can be then imported in development tools (e.g. Vivado or PlanAhead) for bit-stream generation.

5.2 Neural Temporal Testers

In this chapter we show how to use TrueNorth neurons as temporal testers [PZ08a] for evaluation MTL formulae. The logic is interpreted over discrete time, which is a necessary requirement for FPGA realizations. We first give a formal definition of MTL and then show how to build temporal testers for past fragment of it using the TrueNorth model. Using the conversion procedure [MNP07] we are not restricted only to past but are also able to monitor formulae, equisatisfiable to bounded future-MTL specifications, after linear time preprocessing in the size of the formula. The authors in [AH90] showed that MTL is decidable in discrete time.

5.2.1 Metric Temporal Logic

The syntax of an MTL formula φ with past and future operators over a set of boolean variables $P = \{p_1, \cdots, p_m\}$ is defined by the following grammar [MNP06]:

$$\varphi := p \,|\, \neg\varphi \,|\, \varphi_1 \vee \varphi_2 \,|\, \varphi_1 \mathcal{U}_I \varphi_2 \,|\, \varphi_1 \mathcal{S}_I \varphi_2,$$

where $p \in P$, I is $[a, b]$, $a, b \in \mathbb{N}$ and $0 \leq a \leq b$. Other MTL 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 $\boxminus_I \varphi = \neg \diamondminus_I \neg\varphi$.

The semantics of an MTL formula is defined as follows:

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

5.2.2 Parameter synthesis for neural monitors as ILPs

To configure TrueNorth neurons as monitors of MTL specifications, we need to find their parameters $\texttt{params}_i = \{\alpha_i, \beta_i, R_i, \lambda_i, \varepsilon_i, \kappa_i, \mathbf{s}_i, M\}$, (i.e. synaptic weights, leak, thresholds, etc.). For each function (either logical or bounded temporal) our goal is to devise constraints that characterize its behavior: If, for example, for given combination of inputs we require a neuron to output a spike s_j then at the current time step the value of a membrane potential V_j of a neuron j after synaptic and leak integration steps should exceed positive threshold α_j. The configuration task can be seen as a search in parameter space for a point that satisfies all the constraints. Since all the parameters of the TrueNorth model [CMA$^+$13] are either integers or booleans, the configuration problem for a TrueNorth neuron can be stated as an Integer Linear Program (ILP) [Van01] with zero objective function (i.e. finding a feasible solution).

Logical Operators with TrueNorth

To configure neurons for executing logical functions, we need to impose memoryless behavior on the TrueNorth model, which has memory (i.e. membrane potential V_j). This is achieved by forcing a neuron j to reset its internal state V_j at every time point by executing either positive or negative reset steps in the reset mode $\mathbf{0}$ ($\gamma_j = 0$): At each time step a combination of inputs should either exceed α_j (when spike s_j on this particular combination of inputs is required) or stay below β_j (when no spike is required on the input). By assigning $R_j = 0$ we keep $V_j = 0$ at the beginning of every computation. One neuron is required to compute AND, OR, NOT, NAND, NOR and implication.

Example: 2-NAND To illustrate our approach, we configure a TrueNorth neuron to perform a two input NAND operation (see Fig. 5.2). The input neurons n_0, n_1 provide stimulus to the neuron n_2, the parameters of which we need to find in this example.

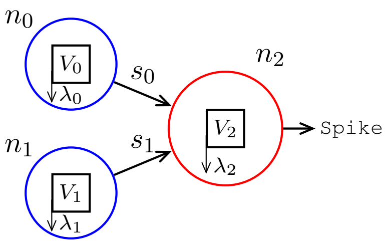


Figure 5.2: Two input TrueNorth circuit

To reproduce the behavior of interest, the output neuron n_2 must spike at every time step unless both n_0 and n_1 are active: in this case n_2 must be inhibited and its membrane potential V_2 must fall below β_2; In all other cases V_2 after leak integration V_2 must exceed α_2. The left-hand side of the inequalities represent the value of V_2 after synaptic and leak integration for different combination of inputs. The ILP is given in Equation 5.1.

$$\min 0 \quad \text{s.t.}$$

$$\begin{array}{cccccc} & & \lambda_2 & \geq & \alpha_2 \\ & +s_1 & +\lambda_2 & \geq & \alpha_2 \\ s_0 & & +\lambda_2 & \geq & \alpha_2 \\ s_0 & +s_1 & +\lambda_2 & < & \beta_2, \end{array} \tag{5.1}$$

where s_0 and s_1 are synaptic weights of neurons n_0 and n_1 respectively; λ_2 is a leak weight of n_2; α_2 and β_2 are positive and negative thresholds of n_2 respectively. This problem can be now solved using a standard ILP solver (e.g. we use `intlinprog` from MATLAB). By similar argument we devise constraints and find parameters of different logical functions (see Table 6.2 for the results for two-input case). Cases with more input and composed logical functions can be also stated as ILPs and solved analogously.

Table 5.1: Neural Parameters of Logical Operators

	s_0	s_1	λ_2	α_2	β_2	γ_2	c_2^λ	ϵ_2	M_2	κ_2
`AND`	9	9	-14	4	-4	0	0	0	0	0
`OR`	9	9	-5							
`NOT`	-4	-	4							
`NOR`	-9	-9	4							
`NAND`	-9	-9	13							
$\rightarrow$	9	-9	-5							

Temporal Operators with TrueNorth

We employ the idea of temporal testers [PZ08a] to construct monitors for MTL specifications in a compositional way. We build temporal testers on top of the TrueNorth model, i.e. use TrueNorth neurons to recognize past-LTL and past-MTL operators by leveraging the internal state V_j and different reset modes.

The *Once* Operator $\diamondminus$ The semantics of the *Once* operator according to [MP92] is as follows: $(w, i) \models \diamondminus p$ iff $(w, k) \models p$ for some $k, 0 \leq k \leq i$. The temporal tester for $\diamondminus$ spikes continuously after p has happened. We need one TrueNorth neuron for this task. Fig. 5.3 shows a circuit where n_1 computes $\diamondminus p$, where p denotes a predicate "n_0 spikes". The constraints that govern behavior of n_1 are as follows:

- To be non-forgetful, leak weight λ_1 must be zero,
- To fire when the first Spike from n_0 arrives, $s_{01} \geq \alpha_1$;
- To continue firing after the first occurrence of Spike on n_0, reset mode **0** with $R_1 > \alpha_1$, (see Table 5.2).

The *Previous* Operator $\ominus$ A neural temporal tester $\psi = \ominus \phi$ needs to postpone a satisfaction of ϕ for one time step. The circuit consists of two neurons (see Fig. 5.3) that compute in an inverse order: At each time step, n_2 computes on input from n_1, and n_1 compute on input neuron n_0, which results in shifting satisfaction of ϕ by one time step.

The Punctual *Once* Operator $\diamondminus_{\{a\}}$ Since $\psi = \diamondminus_{\{a\}} \varphi$ postpones satisfaction of φ over a time steps, we implement this operator as a cascade of previous operators as in [JBG$^+$15]. This implementation requires $2a$ neurons.

$$\diamondminus_{\{a\}} \varphi \quad = \quad \underbrace{\ominus \ominus \dots \ominus}_{a} \varphi$$

The Bounded *Once* Operator $\diamondminus_{[0,a]}$ The temporal tester for $\psi = \diamondminus_{[0,a]} \varphi$ uses six TrueNorth neurons. According to its semantics, the tester should spike when φ is satisfied, and output spikes for a time steps after the last satisfaction of φ. We build a circuit of four neurons that captures last satisfaction of φ: $\neg\varphi \wedge \ominus \varphi$. Its output activates special "bounded once" neuron, which operate in non-reset mode, has one leak weight and input weight from auxiliary circuit of a. Setting R to 1 allows us obtain a concecutive spikes after last satisfaction of φ. The output of the tester is an `OR` neuron, that combines input with output from a "bounded once" neuron.

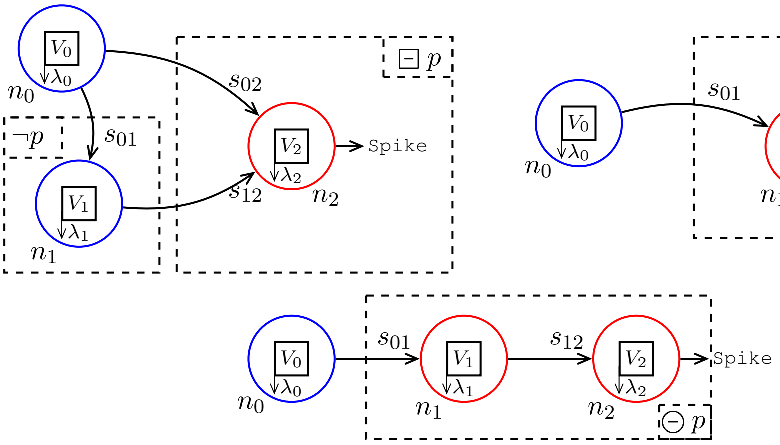


Figure 5.3: Neural Temporal Testers

The *Historically* Operator $\boxminus$ A temporal tester for $\psi = \boxminus \varphi$ takes a neuron n_0 and its an inverse n_1 as inputs (see Fig. 5.3). According to the semantics from [MP92], whenever a Spike from n_1 arrives at the tester, its membrane potential should fall below β_2 and its output never produces a Spike. To configure a neuron as a tester, we find parameters that satisfy the following constraints (see the Equation 5.2 for constraints and possible parameter assignment in Table 5.2):

$$\begin{array}{rrrrcl}
 & +s_{02} & & +\lambda_2 & \geq & \alpha_2 \\
 & & +s_{12} & +\lambda_2 & < & \beta_2 \\
+R_2 & +s_{02} & & +\lambda_2 & \geq & \alpha_2 \\
+R_2 & & +s_{12} & +\lambda_2 & < & \beta_2 \\
-R_2 & +s_{02} & & +\lambda_2 & < & \beta_2 \\
-R_2 & & +s_{12} & +\lambda_2 & < & \beta_2
\end{array} \tag{5.2}$$

The bounded *Historically* Operator $\boxminus_{[0,a]}$ To construct a temporal tester for $\psi = \boxminus_{[0,a]} \phi$ we use the circuit in Fig. 5.3 but configure n_2 differently. We use the reset mode **2** and set the positive threshold $\alpha_2 = a s_{02}$: this allows us to count satisfaction of ϕ over consecutive time steps. To be able to reset the state of n_2 when ϕ gets violated we set the negative threshold to zero, use saturate mode ($\kappa_2 = 1$) and the largest negative weight possible for s_{12} to execute negative reset mode (see Tab. 5.2). To check satisfaction on $[a, b]$ we use the fact that $\boxminus_{[a,b]}\, \phi = \diamondminus_{\{a\}} \boxminus_{[0,b-a]}\, \phi$.

The *Since* Operator $\mathcal{S}$ To build a temporal tester for $\psi = \phi_1 \mathcal{S} \phi_2$ we need four neurons. The definition states that ϕ_2 happened at some time in the past and ϕ_1 held at every time point from the occurrence of ϕ_2 till present. We use an argument from [JBG$^+$15] for constructing the tester: the tester must satisfy $\phi_2(t) \Leftrightarrow \psi(t)$ at the first time step and $\psi(t) \Leftrightarrow (\phi_2(t) \vee (\phi_1(t) \wedge \psi(t-1)))$ starting from the second time step. The tester comprises $\wedge$, $\vee$ and $\ominus$ neurons.

The bounded *Since* Operator $\mathcal{S}_{[0,b]}$ We implement a tester for $\psi = \phi_1 \mathcal{S}_{[0,b]}\, \phi_2$ using a rewriting rule from [JBG$^+$15]: $\phi_1 \mathcal{S}_{[0,b]} \phi_2 = (\phi_1 \mathcal{S} \phi_2) \wedge \diamondminus_{[0,b]}\, \phi_2$. As a combination of two testers, this tester requires 11 neurons.

Table 5.2 summarizes the configuration of TrueNorth neurons as temporal testers and gives parameter assignment. Note, that we consider only deterministic mode in this chapter, hence $M = 0$ for all the testers (omitted from the Table for clarity).

Table 5.2: Neural Temporal Testers

	in_0	in_1	λ_i	α_i	β_i	γ_i	R_i	ϵ_i	κ_i
$\diamondminus$	7	-	0	4	-4	0	5	0	0
$\ominus$	1	-	0	1	0	0	0	0	0
	1	-	0	1	0	0	0	0	0
$\diamondminus_{\{a\}}$	a chain of $\ominus$								
$\diamondminus_{[0,a]}$	combination of $\ominus, \wedge, \neg, \vee$ and a “core” neuron (below)								
	INT_MIN	a	-1	1	0	2	0	0	1
$\boxminus$	0	-18	4	4	-4	0	9	0	0
$\boxminus_{[0,a]}$	1	INT_MIN	0	a	0	2	0	0	1
$\mathcal{S}$	combination of $\wedge, \vee, \ominus$								
$\mathcal{S}_{[0,b]}$	combination of $\wedge$, $\mathcal{S}$, $\diamondminus_{[0,a]}$								

5.3 Case Study and Experimental Results

The purpose of the case study is to show the possibility of applying the proposed in Section 5.1.2 flow for obtaining hardware monitors with acceptable resource utilization. In Chapter 8 we consider application of runtime monitoring for real industrial case studies. In this section we illustrate how to build a hardware neural 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." (see Fig. 5.4).

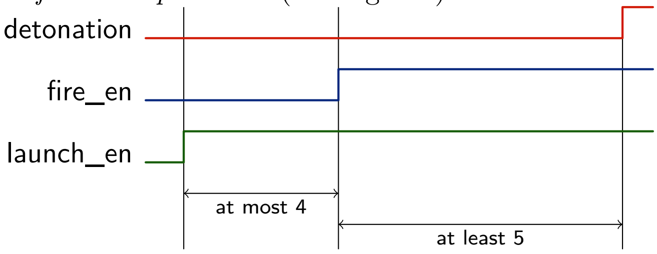


Figure 5.4: Missile timing property specification

The natural-language specification above can be almost directly expressed in MTL: the relation that the rising edge of f must appear within four time steps after the edge of ℓ is by definition bounded eventually; no detonation for five time points is a bounded always of negation of d signal:

$$\uparrow \ell \to \Diamond_{[0;4]} \left(\uparrow f \wedge \Box_{[0;5]} \neg d\right), \tag{5.3}$$

where $\uparrow \varphi$ is a shortcut for the MTL formula $\varphi \wedge \ominus \neg\varphi$ which denotes a rising edge of φ.

We then convert the bounded future MTL formula (Eq. 5.3) to an equisatisfiable past one. After performing a Step 5.1.2 we obtain an equisatisfiable past MTL specification:

$$\diamondminus_{\{9\}} \uparrow \ell \to \diamondminus_{[0,4]} \left(\diamondminus_{\{5\}} \uparrow f \wedge \boxminus_{[0;5]} \neg d\right) \tag{5.4}$$

In the next step we construct a neural circuit which represents a monitor. Each subformula from Eq. 5.4 is converted to a corresponding neural circuit. After composing a neural circuit we simulate it using our C++ implementation of TrueNorth model (Fig. 5.5 shows simulation results).

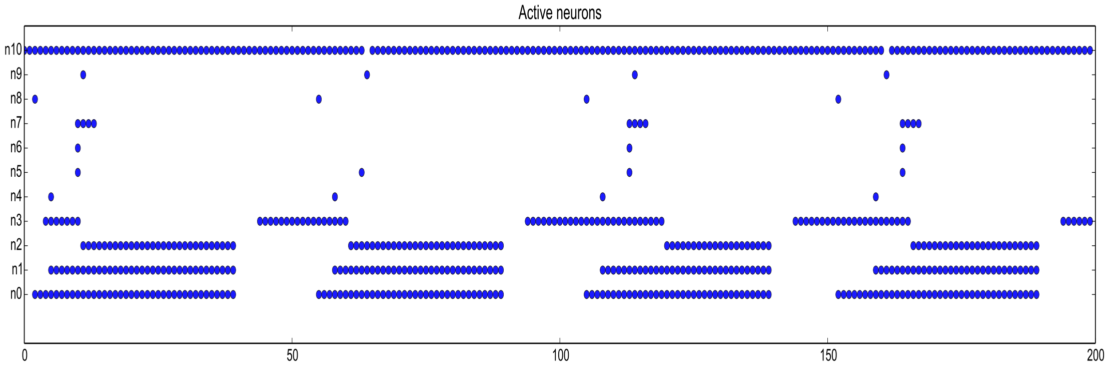


Figure 5.5: Missile monitor: simulation results

We use Zedboard from Xilinx and Vivado System Edition [viv] (University License) for generating neural monitors (see Fig. 5.6). 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 temporal 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. FPGA resources for each operator are listed in Table 5.3 (FF and LUT stand for flip-flop and look-up tables, respectively). Figure 5.6 shows experimental setup.

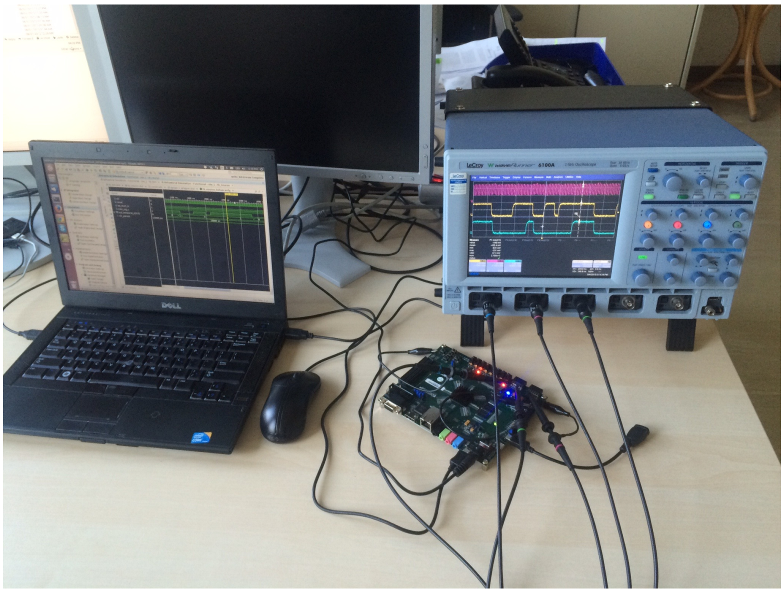


Figure 5.6: Experimental setup (see [osc] for oscillograms)

Table 5.3: Missile monitor: implementation results

Operator	# of neurons	FPGA resources	
		FF	LUT
$\rightarrow$	1	25	82
$\diamondminus_{\{9\}}$	18	443	1623
$\uparrow \ell$	4	98	325
$\diamondminus_{[0,4]}$	6	146	1085
$\wedge$	1	25	82
$\diamondminus_{\{5\}}$	10	246	920
$\uparrow f$	4	98	325
$\boxminus_{[0;5]} \neg d$	2	48	164
Total:	46	1129 (1.1%)	4606 (8.6%)

5.4 Summary

In this chapter we showed how digital spiking neural model, called TrueNorth, can be applied for qualitative monitoring of signals against MTL specifications. We presented the flow to formalize natural language requirements to a formula and showed how to build neural testers for each of logical and temporal components of MTL specifications. We also showed how the validated neural circuit can be synthesized in FPGA and used to monitor requirements in real time. This work demonstrates application of neural models for qualitative monitoring of cyber parts of CPS. We also demonstrated applicability our approach on a case study where we build a neural hardware runtime monitor for a missile launch from natural language requirement.

CHAPTER 6

Neural Models for Quantitative Monitoring

In this chapter we further elaborate on the research question **RQ 2**, and show how to perform computations using neural models. We consider two neural models: biophysically-accurate yet computationally plausible Hodgkin & Huxley [HH52] neural model, extended with the synapse model from [RvR09], and hardware-efficient digital TrueNorth [CMA+13] model; these two models use a spike as a mechanism to communicate between neurons.

Neuroscience and computer engineering are fundamentally different: while, in general, the purpose of a neuroscientist is to understand a nervous system and to develop models capable of explaining its function (from the physical world to models), the purpose of a computer engineer to realize a hardware system that would satisfy initial requirements (from models to the physical world). Despite the differences, from the papers of McCulloch & Pitts [MP43] neural networks are influencing the development of computer hardware, which result in dedicated chip architectures [SMN11, WHTvS14, CMA+13].

Neuromorphic hardware accelerators [DBDRC+15] that co-exist together with the traditional CPU infrastructure enable new functionality of hardware systems, add flexibility to existing designs, retain established design flow, and reduce overall costs when implemented on a commercial off-the-shelf (COTS) general purpose hardware. To build efficient hardware implementation, it is essential to understand how to perform fundamental arithmetic operations, as those set the basis for the higher-level complex processing.

As temporal logic can be viewed as filtering [RBNG16], allowing both qualitative and quantitative semantics, depending on the choice of windowing function, we show how to implement temporal logic monitors for MTL using the TrueNorth model.

The contributions of this chapter can be summarized as follows:

- we show how arithmetic operations can be implemented both using biophysical and digital neural models;
- we elaborate on the role of assumptions on inputs to obtain the correct computation results for both models;
- we present temporal logic monitors based on the TrueNorth model for convolutional semantics of MTL.

The rest of this chapter is organized as follows: Section 6.1 provides a short description of the spiking models under study. Section 6.2 elaborates on performing computations with Hodgkin-Huxley and TrueNorth neural models. Section 6.3 presents the temporal logic monitoring using convolution. Section 6.4 offers concise summary of the presented results.

6.1 Neuron and Synapse Modeling

In this section we recap the biophysical Hodgkin-Huxley neural model and synapse models under study. Description of the TrueNorth model is given in Chapter 2.

6.1.1 The Hodgkin-Huxley neuron model

The model qualitatively describes the dynamics of the membrane potential as a function of activation and deactivation of ionic channels such as sodium and potassium together with the leak channel [HH52], e.g. ODE describe the evolution of the membrane potential:

$$C_m \frac{dV_m}{dt} = -[\bar{g}_K n^4 (V_m - E_K) + \bar{g}_{Na} m^3 h (V_m - E_{Na}) + \bar{g}_l (V_m - E_l)] + I_{in}, \tag{6.1}$$

where C_m and V_m are the membrane capacitance and potential; $\bar{g}_K$, $\bar{g}_{Na}$ and $\bar{g}_l$ are the conductances of the potassium, sodium and leak channels, respectively; E_K, E_{Na} and E_l represent the reversal potential of the channels; n, m and h are voltage-dependent gating variables for the potassium channel activation, sodium channel activation and sodium channel inactivation, respectively. The detailed description is presented in original paper of Hodgkin & Huxley [HH52].

6.1.2 Modeling Synapses

In order to model the current flow between neurons in the Hodgkin-Huxley model (as it accounts for external current stimulus), we implemented three models of synaptic conductance g_{syn} from [RvR09] and assume that $I_{\text{syn}} \propto g_{\text{syn}}$. The first model (*exponential decay*) assumes that ionic channels open instantaneously upon an arrival of a presynaptic action potential and then g_{syn} decays exponentially:

$$g_{\text{syn}}(t) = \bar{g}_{\text{syn}} e^{-(t-t_0)/\tau}. \tag{6.2}$$

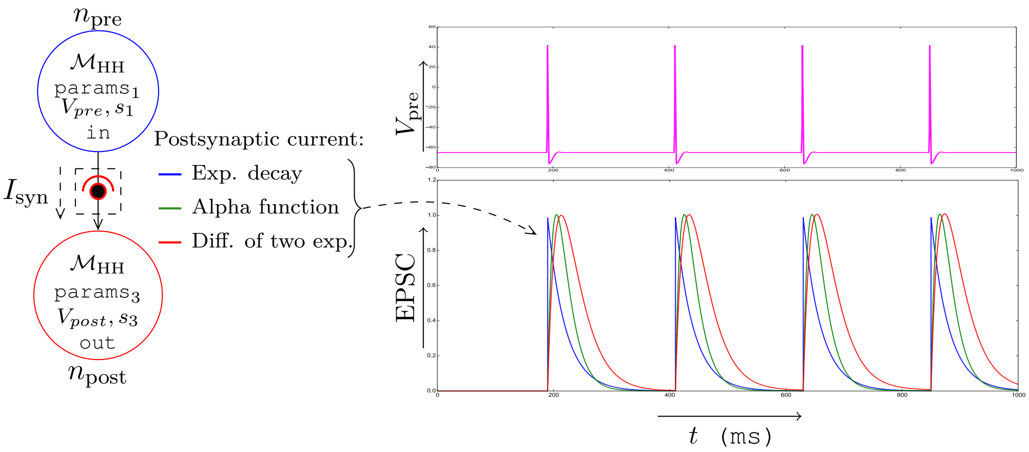


Figure 6.1: Normalized EPSC in response to pre-synaptic action potentials (V_{pre})

The *alpha function* [VVAE94] takes into account that the opening of ionic channels is not instantaneous without introducing additional parameters into the model:

$$g_{\text{syn}}(t) = \bar{g}_{\text{syn}} \frac{t - t_0}{\tau} e^{1-(t-t_0)/\tau}. \tag{6.3}$$

A more comprehensive representation of the dynamics of synaptic conductance can be modeled by the *difference of exponentials* where the rise and decay times are explicitly introduced [RvR09]:

$$g_{\text{syn}}(t) = \bar{g}_{\text{syn}}(e^{-(t-t_0)/\tau_{\text{decay}}} - e^{-(t-t_0)/\tau_{\text{rise}}}). \tag{6.4}$$

We implement the above models of synaptic conductance (Eq.6.2-6.4) and employ them in the design of arithmetic operations using the Hodgkin-Huxley model. Figure 6.1 depicts the normalized excitatory post-synaptic current (EPSC) for synapse models in response to the presynaptic action potential; Tab. 6.1 lists the parameters of the models studied in this work.

Table 6.1: Parameters of neuron and synapse model

Model	**Parameters**
Hodgking-Huxley model	C_m, $\bar{g}_K$, $\bar{g}_{Na}$, $\bar{g}_l$, E_K, E_{Na}, E_l, $\alpha_{\{K,Na\}}$, $\beta_{\{K,Na\}}$
TrueNorth model	$A_j, w_j, s_j, \lambda_j, \epsilon_j, \gamma_j, \alpha_j, \beta_j, \kappa_j$
Exponential Decay	$\bar{g}_{\text{syn}}$, τ
Alpha function	
Double-exp Synapse	$\bar{g}_s^{max}$, τ_{rise}, τ_{decay}

6.2 Computations with Neural Models

As we study computation for both biophysical and digital neural models, we assume that: (i) biophysical model operates over real time and real-value domain (it is simulated with a pre-defined rational-value integration step $\Delta t = \frac{q}{r}$, where $q, r \in \mathbb{N}$), (ii) to be efficiently hardware-realizable, the digital neural model operates over discrete time and finite-value domain. A *trial* is an execution of a neural circuit for a time interval $[0, T]$. Each neuron $n_i = (\mathcal{M}, \texttt{params}_i, V_i, s_i, l_i)$ is characterised by (i) an underlying model $\mathcal{M}$, (ii) a set of parameters $\texttt{params}_i$, (iii) its membrane potential V_i, (iv) a binary spike output s_i, and (v) a label $l_i \in \{\texttt{in}, \texttt{interm}, \texttt{out}\}$ of a neuron. For a neural network $N = \{n_1, \cdots, n_m\}$ of m neurons we define its computation C over a time interval in the following way: $C : [0, T] \mapsto V^m \cup S^m$, where $V^m = \{V_1, \cdots, V_m\}$ and $S^m = \{s_1, \cdots, s_m\}$ are membrane potentials and spikes of neurons in the network N respectively.

A spiking activity of a neuron n_i over a trial is defined as a mapping $[0, T] \mapsto s_i$. We assume that a neuron n_i encodes numbers in a spike-count rate [DA05], and measure spiking activity over a time window $w \subseteq [0, T]$ of the length $\|w\|$:

$$r_i = \frac{\sum_{t \in w} s_i[t]}{\|w\|}. \tag{6.5}$$

The task of computing a function f in a neural network N then can be formulated as follows: for a given neural model $\mathcal{M}$ find number of neurons with labels `interm`, `out` and their corresponding parameters `params` such that $r_{\{\text{out}_1, \cdots \text{out}_n\}} = f(r_{\{\text{in}_1, \cdots, \text{in}_m\}})$.

The circuit topology for the two-argument operations is shown in Fig. 6.2. The TrueNorth model (left) assumes that neurons are connected with discrete weights. For Hodgkin & Huxley model (right) we introduce the synaptic connections between neurons. Input neurons (blue and green) provide spikes with rates r_1 and r_2 to the computing neuron n_3, which outputs the result $f(r_1, r_2)$.

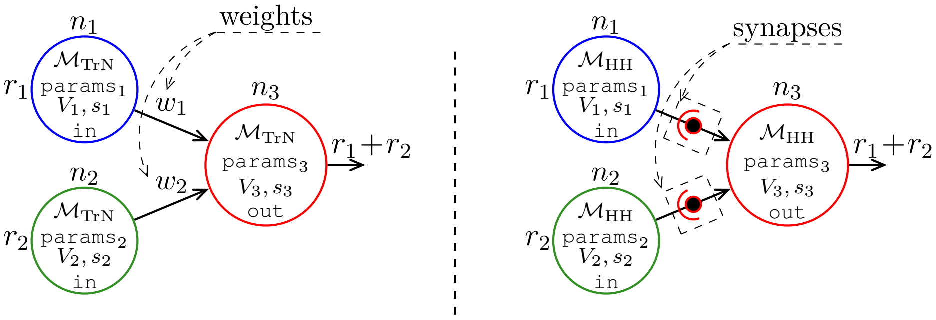


Figure 6.2: Neural models for computations on spike rates

6.2.1 Computing Addition

Addition using the TrueNorth model is realized as follows: we configure the output neuron n_3 in the linear reset mode ($\gamma = 1$), the input weights and the positive threshold are set to one. This allows to: (i) generate a spike in the `out` neuron whenever an action potential is generated by the `in` neurons; (ii) memorize in the membrane V_3 if two spikes happened at the same time instant, and temporally separate output spikes over the adjacent time steps.

Without any assumptions on input, if the membrane potential is empty at the end of the trial, then the result of addition is correct. If, however, the both inputs arrive at the end of the trial, the output neuron may not be able to generate the correct result when the number of the remaining time steps in the trial is less the value of the membrane potential. This can be mitigated by extending the length of the trial for the output neuron, which we aim to avoid to keep the hierarchical composition simple. In this work we assume random arrival of input, which is implemented in the TrueNorth model as the “rate-store” function [CMA$^+$13]. Fig. 6.3 shows the simulation results: blue + green = red (n_1 – blue; n_2 – green; n_3 – red). We performed 1000 simulation trails for 1000 time steps each. The leftmost plot shows the dependence between input and output spike rates for all the trials. Subsequent 1000×1000 plots present the spiking activity of input and output neurons over time during all the trials: a black pixel with coordinates (i, j) denotes a spike in trial i (horizontal axes) at a time step j (vertical axes).

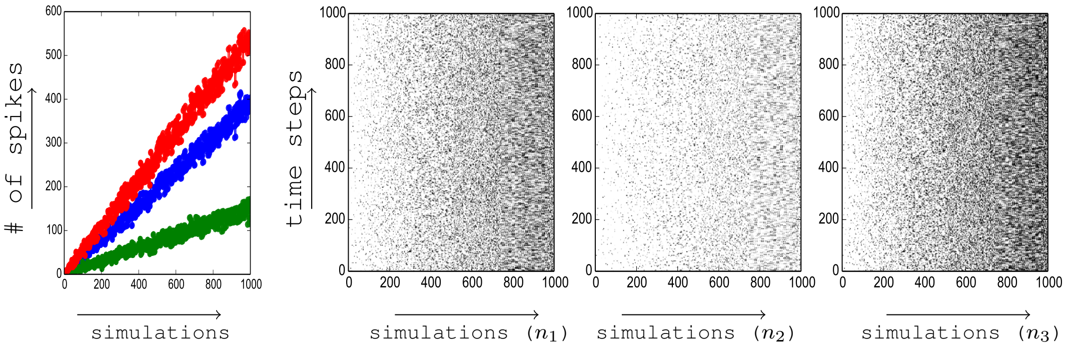


Figure 6.3: Addition in the spike rates using the TrueNorth model

Addition using the Hodgkin-Huxley model crucially depends on the underlying synapse model. Unlike the TrueNorth, the biophysical Hodgkin-Huxley model is not able to memorize the occurrences of two simultaneous spikes from both inputs in the membrane potential V_m. Furthermore, one needs to account for the refractory period, in which no action potential can be initiated. To obtain correct results it is vital to distribute the synaptic current over the time, such that after the refractory period the output neuron still receives enough stimulation. The *alpha function* and the *difference of two exponential* can be used to perform the addition with Hodgkin-Huxley model. Fig. 6.4 shows the simulation result of a trial, where the synapse are modelled as the alpha function: the profile of the pre-synaptic voltage profile for the input neurons (blue and cyan), the superposed synaptic current (green), the post-synaptic voltage profile (red).

The fact that at the end of the trial the membrane potential V_m stabilizes at the resting value for a time $T_{\text{stable}} \sim 10ms$ is necessary but not sufficient requirement for producing the correct results.

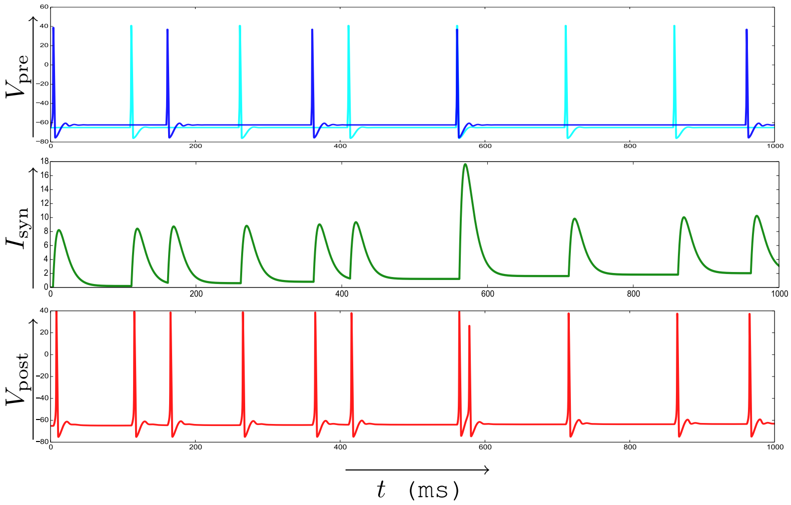


Figure 6.4: Addition of spike rates with the Hodgkin-Huxley model

6.2.2 Computing Constant Multiplication

Constant multiplication using the TrueNorth model is implemented as follows: for each occurrence of the input spike, the output neuron generates C spikes, hence the strength of the connection (i.e. its weight) is proportional to C. The output neuron n_3 is set to the non-reset mode (i.e. $\gamma = 2$) to be able to store all the spikes seen so far. The negative leak λ and the saturate flag κ ensure that the membrane potential will converge to zero in the absence of input. In the case of constant factor division, we need to output one spike for each C spikes seen so far. To do so, we set a positive threshold α proportional to C and weight s_0 to one. We also set the leak to zero and linear reset mode ($\gamma = 2$); see Fig. 6.5 for the simulation results.

Constant multiplication using the Hodgkin-Huxley model can only be performed if the synaptic current and the input spike rate satisfy the following requirements: (i) the length of the synaptic current pulse must be proportional to the multiplication constant C; (ii) the input arrival rate is low enough to allow synaptic current attenuate to its resting value before the arrival of the next spike from the pre-synaptic neuron. To satisfy the first requirement it is necessary to control both the amplitude and the width of the synaptic current, as the "difference of two exponentials" allows to adjust both rise and decay times of the synaptic current, this model shows the best results. Fig. 6.6 shows the simulation trial of performing the multiplication of the input rate by four: the profile of the pre- and post-synaptic membrane potentials (magenta and green, respectively), and the total synaptic current (red).

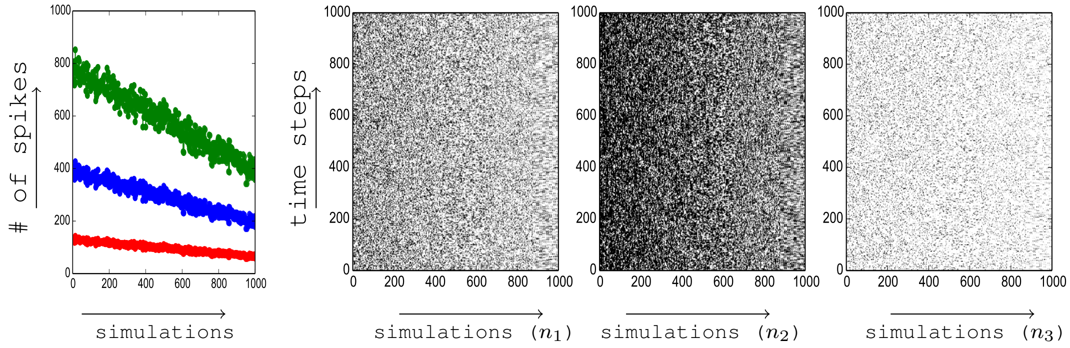


Figure 6.5: Constant multiplication and division of the spike rates with the TrueNorth model: $2 \cdot$ blue = green, $\frac{1}{3} \cdot$ blue = red (n_1 – blue; n_2 – green; n_3 – red)

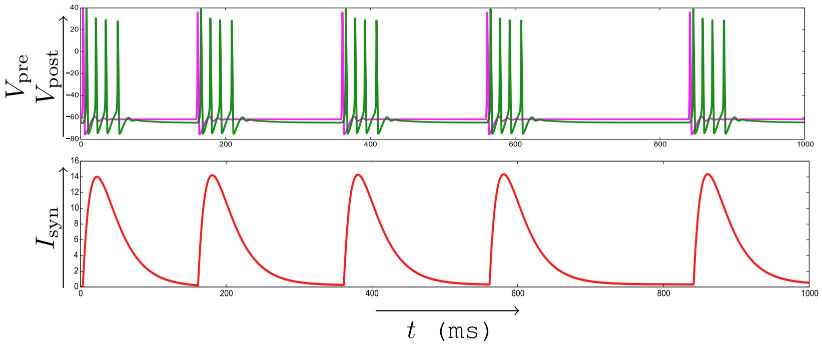


Figure 6.6: Constant multiplication of the spike rates with the Hodgkin-Huxley model

6.2.3 Computing Subtraction and min/max

Subtraction using the TrueNorth model is realized analogously to addition: the subtrahend though receives the weight of -1. Such implementation is inherently sensitive to the input timing: if the spikes from the subtrahend neuron happen before the spikes of the minuend neuron, the circuit computes $max(0, r_1 - r_2)$, i.e. if the actual difference is negative, no spikes are outputted.

Conversely, if at a time step t_i the output neuron receives the spike from the minuend, it needs to compute the running result and the correct way would be also to generate an action potential, although a spike from the subtrahend after an arbitrary silence interval would make the running result incorrect until the next spike from the minuend. The necessary and sufficient condition to ensure the correctness of the result is $V = 0$ at the end of the trial. Fig. 6.7 shows the simulation results for the randomized input.

Subtraction using the Hodgkin-Huxley model can only be performed when the following assumptions on the inputs are met: since the model does not have a mechanism to memorize the occurrences of spikes from the input neurons, all action potentials of the

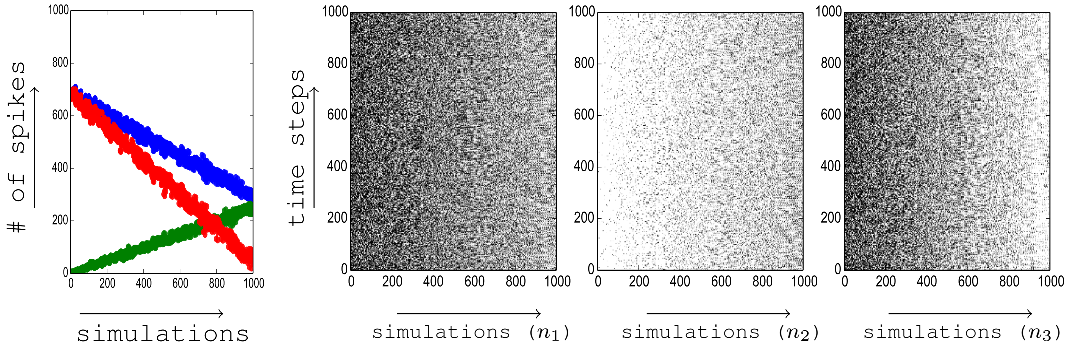


Figure 6.7: Subtraction in the spike rates with the TrueNorth model: blue − green = red (n_1 − blue; n_2 − green; n_3 − red).

subtrahend neuron should coincide (up to the small time difference) with the action potential of the minuend neuron.

Minimum/Maximum using the TrueNorth model is based on the fact, that the subtraction actually computes $max(0, r_1 - r_2)$. We now can construct the minimum and maximum operators compositionally as follows: $min(r_1, r_2) = r_1 - max(0, r_1 - r_2)$, and $max(r_1, r_2) = r_2 + max(0, r_1 - r_2)$. Fig. 6.8 shows the simulation results of performing these computations: the neuron n_3, which generates an offset of 200 spikes per trial on average, is added to separate the results from the inputs. Table 6.2 shows parameters of the TrueNorth model to perform the presented arithmetic operations.

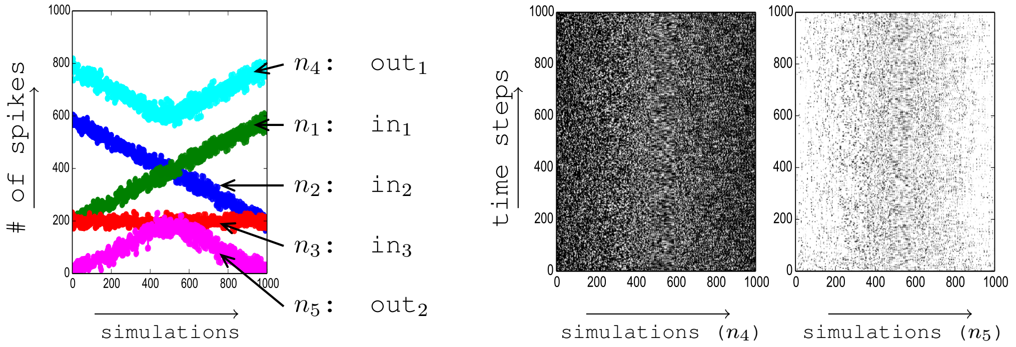


Figure 6.8: Computing min/max using the TrueNorth model: $r_4 = max(r_1, r_2) + r_3$, and $r_5 = min(r_1, r_2) - r_3$

6.3 Neural filters as temporal logic monitors

The task of multiplying two spike rates $f_0 \cdot f_1$ is the most challenging one among arithmetic computations with the TrueNorth model. We use the fact from the convolution

Table 6.2: Parameters of the TrueNorth Model

	s_0	s_1	λ_2	α_2	β_2	γ_2	ϵ_2	M_2	κ_2
$f_0 + f_1$	1	1	0	1	0	1	0	0	0
$f_0 - f_1$	1	-1	0	1	0	1	0	0	0
$f_0 \cdot C$	C+1	-	-1	1	0	2	0	0	1
f_0 / C	1	-	0	C	0	1	0	0	0

theorem [Lyo11] to tackle this problem: performing convolution of inputs in time domain allows us to obtain multiplication in the frequency domain.

Fig. 6.9 shows the circuit of TrueNorth neurons for performing circular convolution. Inputs u and v of a trial of the length T are supplied in parallel to the convolution circuit. Neurons for computing pairwise product (which terms of TrueNorth spikes correspond to a logical `AND`) are shown on the red background. Output neurons (purple background) perform summation and store in membrane potential V_j the result of a convolution in a time step j.

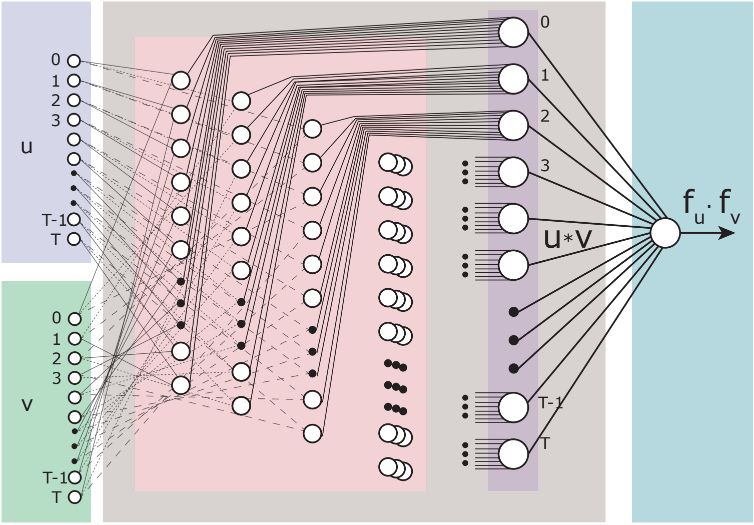


Figure 6.9: Circular Convolution using TrueNorth: the Circuit

To use the circuit from Fig. 6.9 for computing pairwise product of input signals that are observed over time, we implement additional delay circuitry with TrueNorth to store the signal for the period T.

The circuit from Fig. 6.9 gives the exact product of two numbers over a period T, without any assumptions about inputs (i.e. spikes can appear anywhere in T, therefore we buffer the signal for one period and supply it in parallel). The price we pay for exact value of the product is the size of the circuit, which requires in general $O(T^2)$ neurons.

6.3.1 Using TrueNorth to monitor MTL Temporal Operators

In [RBNG16] it is proven that convolution can be used to compute the satisfaction of an MTL formula with respect to a signal, and the semantics of past temporal operators can be defined as follows:

$$\begin{aligned}
[\![x, \diamondminus_I \varphi]\!][i] &= \textstyle\sum_{j \in \mathbb{T}} [\![x, \varphi]\!][j] \cdot w^-_I[i-j] \\
[\![x, \boxminus_I \varphi]\!][i] &= \min_{j \in i-I} [\![x, \varphi]\!][j] \\
[\![x, \varphi \mathcal{S}_I \psi]\!][i] &= \tfrac{1}{|I|} \textstyle\sum_{j \in I} ([\![x, \boxminus_{[1,j-1]} \varphi]\!][i] \cdot [\![x, \diamondminus_{\{j\}} \psi]\!][i])
\end{aligned}$$

where $w^-_{[a,b]}$ is a discrete-time window defined as a sum of the δ Kronecker function:

$$w^-_{[a,b]} = \frac{1}{b-a} \sum_{i=a}^{b} \delta(s-i) \tag{6.6}$$

Interpreting $\times$ and $+$ as min and max allows to use convolution to compute whether the signal satisfies an MTL specification φ. If the assumption holds that the spiking activity at the end of the trial T is given, we can reuse the circuit from Fig. 6.9 with a minor modification: we reconfigure the sum neurons (on the purple background) for computing max (equivalent to the logical `OR` in the quantitative case).

The Bounded Once Operator $\diamondminus_{[a,b]}$

To compute a satisfaction signal of the formula $\diamondminus_{[a,b]} p$, we perform a convolution of p with a window $w^-_{[a,b]}$, which is derived from the interval $[a,b]$ (Eq. 6.6). We supply the signal to the input u of the circuit from Fig. 6.9. We then use use an interval $[a,b]$ of the bounded once operator to form a window $w^-_{[a,b]}$ as follows: neurons in v with indices $i \in [a,b]$ must spike, others not. Fig. 6.10 shows an input signal, a window and the output of the circuit for two examples.

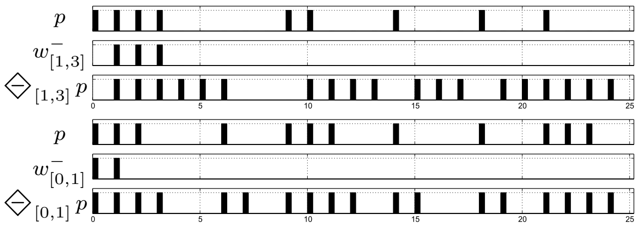


Figure 6.10: Computing $\diamondminus_{[a,b]}$ using TrueNorth

The Bounded Historically Operator $\boxminus_{[a,b]}$

Given a signal w of a length T represented with SPIKES and a bounded historically formula $\boxminus_{[a,b]} p$, to compute the satisfaction signal with the circuit from Fig. 6.9 we proceed as follows:

(i) supply the signal to the input u;

(ii) construct a window $w^{-}_{[a,b]}$ (Eq. 6.6) in which a neuron i spikes if $i \in [a, b]$. This window then is supplied to v;

(iii) deactivate circular connections from Fig. 6.9 to obtain standard (non-circular) convolution;

(iv) reconfigure summation neurons (on a purple background in Fig. 6.9) to output SPIKE if and only if the number of spikes in the input is greater of equal to $b - a$. This is done by increasing the positive threshold α to $b - a$ while keeping weights value at one in the normal reset mode ($\gamma = 0$).

Although we are not computing convolution in its pure sense, (a minimum in a moving window according to the semantics from [RBNG16]), the versatility of the TrueNorth model allows to reuse the same circuit with different configuration of neurons to accomplish this task (see Fig. 6.11 for simulation results).

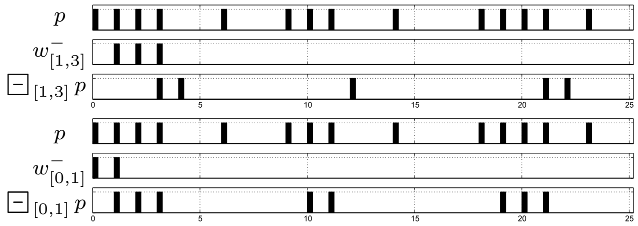


Figure 6.11: Computing $\boxminus_{[a,b]}$ using TrueNorth

The Bounded Since Operator $\mathcal{S}_{[a,b]}$

The satisfaction of the MTL formula $\varphi = p\,\mathcal{S}_{[a,b]}q$ according to its semantics from [RBNG16] is defined as a union of pairwise products of bounded historically and bounded once operators. For each $j \in [a, b]$ we compute $\boxminus_{[1,j-1]}\, p$ (see Sec. 6.3.1) and $\diamondminus_{\{j\}}\, q$ (Sec. 6.3.1). Using T neurons we perform a pairwise `AND` between $\boxminus_{[1,j-1]}\, p$ and $\diamondminus_{\{j\}}\, q$ thus obtain the result for a time step j in the interval of interest. We obtain the final result by performing a sum over all j computed before. Figure 6.12 shows the simulation results of computing $p\,\mathcal{S}_{[a,b]}q$ for two different signals and intervals.

6.4 Summary

In this chapter we showed how inherently different models of spiking neurons that come from neuroscience and computer engineering can be configured to perform the computations on the spike rates. The correctness of the computational results for both models depends on the operation being performed and the spike profile of the inputs. According to the results, the assumption on inputs for obtaining the correct computation results with the TrueNorth model are less restrictive then for the Hodgkin-Huxley model.

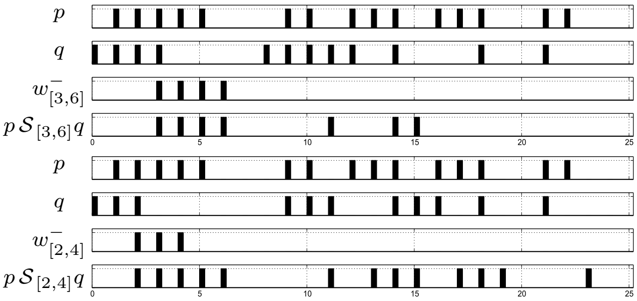


Figure 6.12: Computing $p\,\mathcal{S}_{[a,b]}\,q$ using TrueNorth

The model of the synaptic transmission is crucial to perform the computation and obtain correct results using the biophysical neural model.

We then developed an approach how to implement circular convolution using the TrueNorth neurons, and then use the obtained circuit to monitor MTL specifications. We used convolution of an original signal and a windowing function, defined from the formula to obtain the satisfaction signal.

CHAPTER 7

Runtime Monitoring in Automotive Electronic Development

This chapter elaborates on applied results of the thesis. We consider research questions **RQ 3** and **RQ 4**, and discuss the applications of runtime monitoring in the industrial design practice. The chapter aims to show how automotive electronic development can benefit from applying semi-formal techniques, as runtime monitoring, and gives two use cases which forster monitor reuse from early design prototype to post-silicon verification.

Verification & Validation of complex mixed-signal integrated circuit products in industrial practice accounts for 60-70% of project development time. It is known that simulation, which is the dominant method for pre-silicon verification, does not scale due to immense computing requirements. The increasing trend to overcome the simulation bottleneck is to complement it with the emulation based approach: the designed system is replaced by an early prototype implemented using FPGA allowing long-term/stress testing and whole-range parameter variations, which is usually limited to small examples with simulation-based verification [BBNS15, BBN13, Don10, ALFS11]. Design emulation techniques also allow one to approximate an analog component with discretized behavioral model, therefore enabling end-to-end testing early on. However, verification techniques used with the emulation-based approach involve manual tasks, making them error-prone and time consuming. To improve the verification techniques in this scenario, we propose combining assertion-based runtime verification with design emulation. In this framework we employ STL [MN13a] as specification language to express the desired requirements.

Electronic components and software systems in the automotive industry are increasingly prominent: They already encompass up to 35% of the costs of a car and they will continue to expand [KS14]. The compliance with the stricter safety standards (e.g. ISO

26262[iso11]), the increase in the number of functions that the electronic systems of a vehicle must fulfill (ADAS [OKT14], X-by-wire [San13]), the tight time-to-market schedule, and the enlarged system complexity challenge the automotive electronic industry as never before. To overcome these challenges, chip manufacturers strive for solutions that help capture errors at all stages of the development cycle.

Sensors build up the front-end between the analog world and the digital electronic systems. They provide the required level of safety and comfort while driving: E.g., they detect the rotation angle of a steering wheel, the position of pedals, the pressure for launching airbags [AG16], the distance to surrounding objects (high-speed radar sensor), air pressure in tires (tire pressure monitor sensor, TPMS), etc. This chapter is dedicated to illustrate how runtime verification [Leu12], a light-weight state-of-the-art technique for checking compliance between a specification and a system at runtime, can be applied during sensor chip development. We consider a sensor that measures a magnetic field, from which the angle of the steering wheel can be calculated in an electronic power-steering application (Fig. 7.1: two magnetic sensors measure the rotation angle of the steering wheel. The rotation angle is then sent to the electronic control unit (ECU). Based on the speed and the sensor data, the ECU activates the motor. The lower the speed, the more the motor is activated. This affects the ease of rotating the steering wheel). Measuring angular information is also required in electronic throttle control (Fig. 7.2) where the sensor is used to detect the position of the pedal.

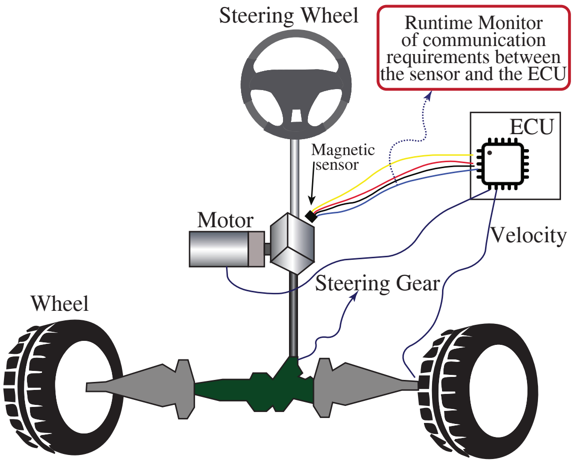

Steering Wheel
Runtime Monitor of communication requirements between the sensor and the ECU
ECU
Magnetic sensor
Motor
Velocity
Steering Gear
Wheel

Figure 7.1: Electronic power steering application

According to the state-of-practice, verification and validation (lab evaluation) of such sensor interfaces become challenging for the following reasons:

- Verification for the sensor interfaces has to cover real-time mixed signal domains;
- Failure during the reception, decoding and processing of sensor data in the system can lead to unexpected or false events which might put human safety in danger;

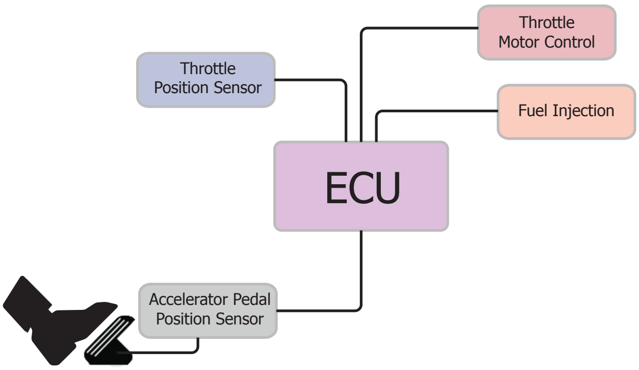


Figure 7.2: Electronic Throttle Control for Engine Management

- Most of the functionality of sensor interfaces can only be verified at the system level of the chip and at the system application level. Using classical mixed-signal simulation approach becomes a bottleneck;
- Many verification scenarios of the sensor interfaces such as long-term verification run with checking of millions sensor data frames are not suitable using computer-based simulation as well as manual sensor data evaluation/checking.

The contributions of this chapter can be summarized as follows:

1. We showcased runtime verification techniques in automotive sensor chip design and provide two use cases: Simulation and lab measurement.
2. In the first use case, the runtime monitors which are formalized and generated from product requirements are embedded in the test bench of a chip concept simulation. The implementation's correctness with respect to the requirements is then possible to monitor during simulation runtime.
3. In the second use case, the runtime monitors are reused and synthesized into an FPGA hardware for monitoring implementation correctness in the lab.

We now elaborate on how runtime monitoring of STL requirements can be applied for both checking the conformance of a chip model and for testing the chip later on against the formalized requirements.

Fig. 7.3 shows the flow of the runtime-monitors generation for the two use cases. In a first step, we formalize time-invariant product requirements, and obtain a set of temporal logic formulas. We use bounded-time STL as a specification language, due to its ability to handle analog-mixed-signal properties. The formulae to be hardware-synthesizable need to be converted to a specific form (containing only past temporal operators i.e. pastified) and simplified (eliminate duplicate sub-trees to save the hardware resources). Hence, in this step we produce an equisatisfiable past STL formula [MNP07], which will be used as a formal specification for the use cases.

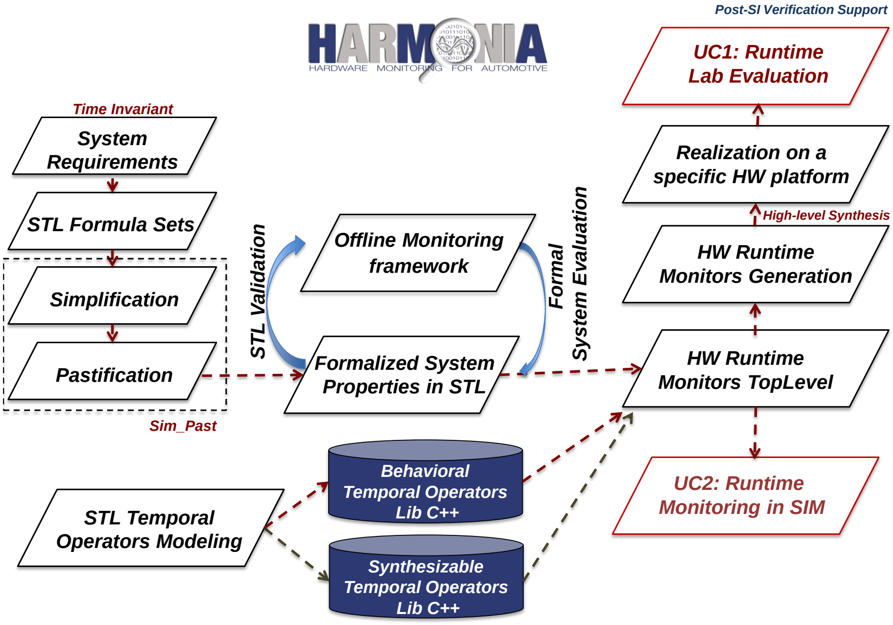

HARMONIA
HARDWARE MONITORING FOR AUTOMOTIVE
Post-SI Verification Support
UC1: Runtime Lab Evaluation
Time Invariant
System Requirements
Realization on a specific HW platform
High-level Synthesis
STL Formula Sets
Offline Monitoring framework
Formal System Evaluation
STL Validation
HW Runtime Monitors Generation
Simplification
Pastification
Formalized System Properties in STL
HW Runtime Monitors TopLevel
Sim_Past
Behavioral Temporal Operators Lib C++
UC2: Runtime Monitoring in SIM
STL Temporal Operators Modeling
Synthesizable Temporal Operators Lib C++

Figure 7.3: Runtime Monitoring Generation Flow

7.1 Use Case 1: Runtime Monitors in Simulation

The first use case, *Runtime Monitors in Simulation*, checks the implementation correctness of the developing electronic product. The use case draws inspiration from the offline monitoring framework [NN14]. However, simulation traces are simultaneously generated and checked against the product's requirements during the simulation runtime. Monitors are embedded in the toplevel test bench and are being simulated together with the DUT. This allows one to run the chip model and the monitors at once and observe whether the chip model satisfies its formalized specification for different test cases (that correspond to modelling various environmental conditions, power supply quality or fault injections). This application allows design teams at an early stage assess whether the sensor prototype adheres to the formal requirements.

We use the SystemC implementation of the STL temporal-operators ("behavior" Fig. 7.3). This allows us to simulate both chip design prototype and monitor within the SystemC simulation kernel, speed up the implementation by using the facilities of the SystemC and C++ libraries.

7.2 Use Case 2: Runtime Monitors for Lab Evaluation

At a later phase of the product manufacturing, the chip is taped-out in the so-called engineering samples. These samples still need to be verified in the lab environment.

The key concept of use case 2 is to synthesize monitors from use case 1 into FPGAs, to be used as an extended lab-equipment support for lab evaluation activities. This is especially valuable for the scenarios in which errors can be seen only after a certain test time. For example, hardware runtime errors, or scenarios including large amounts of data exchange between sensor and ECU. The aim at this stage is to guarantee that the implementation satisfies the requirements, under its operational condition, and sometimes under a stress condition.

In this use case runtime monitors are synthesized in an FPGA and run in parallel with the test hardware to keep up with the real-time sensor-ECU data exchange requirements. The C++ code ("synthesizable" in Fig. 7.3) is supplied to High-Level Synthesis [Inc] to generate RTL that can be put in FPGA. To be able to obtain efficient hardware implementation the code must use hardware precise data types, must not dynamically allocate memory. This approach allows to reuse monitors across product design phases.

7.3 Case Study: Automotive Sensor Interface

We implemented the use cases described in Sections 7.1 and 7.2 to show the applicability of the proposed approach both in simulation and in hardware for a chip model and an engineering sample of the magnetic sensor used in electronic power steering (Fig. 7.1).

In this section we show how two fundamental electrical requirements, which define a shape of a synchronization pulse and a sensor response of Peripheral Sensor Interface 5 (PSI5), can be checked in a lab environment using the proposed approach. The ECU sends synchronization pulse to the sensor via the voltage line. The sensor produces the reply by modulating the current. We monitor both, the voltage from the ECU and the current from the sensor: raise and fall time of these pulses must not exceed t_{rise}. These requirements can be written in the past-STL as follows:

$$\begin{array}{ll} \text{rise\_req:} & \texttt{enter}(\text{high}) \rightarrow \text{trans}\,\mathcal{S}_{[0,t_{\text{rise}}]}\texttt{exit}(\text{low}) \\ \text{fall\_req:} & \texttt{enter}(\text{low}) \rightarrow \text{trans}\,\mathcal{S}_{[0,t_{\text{rise}}]}\texttt{exit}(\text{high}), \end{array}$$

where $\texttt{enter}\varphi$ and $\texttt{exit}\varphi$ are syntactic sugar for $\ominus \neg\varphi \wedge \varphi$ and $\ominus\, \varphi \wedge \neg\varphi$.

Fig. 7.4 shows the simulation setup and the result of a run of the chip model, which illustrates the proposed application of runtime monitors during the chip concept design phase. We simulate both the SystemC model of the chip and the runtime monitor, obtaining the monitoring results at the end of the simulation run. A magnetic field - Signal 1 in Fig. 7.4 - is an input for the sensor. Then the field values are sampled by an Analog-to-digital converter (ADC) and passed through a filter for internal processing in a sensor, Signals 2-4. After powering the chip and a passed stabilization time, the ECU sends synchronization pulses to the sensor (Signal 5, Fig. 7.4). In synchronization mode, each synchronization pulse sent by the ECU is responded by modulated sensor frames: Signal 6. Signals 7-12 in Fig. 7.4 are intermediate outputs of the sub-formula of the specification "rise_req" ("fall_req" is omitted from the picture for conciseness).

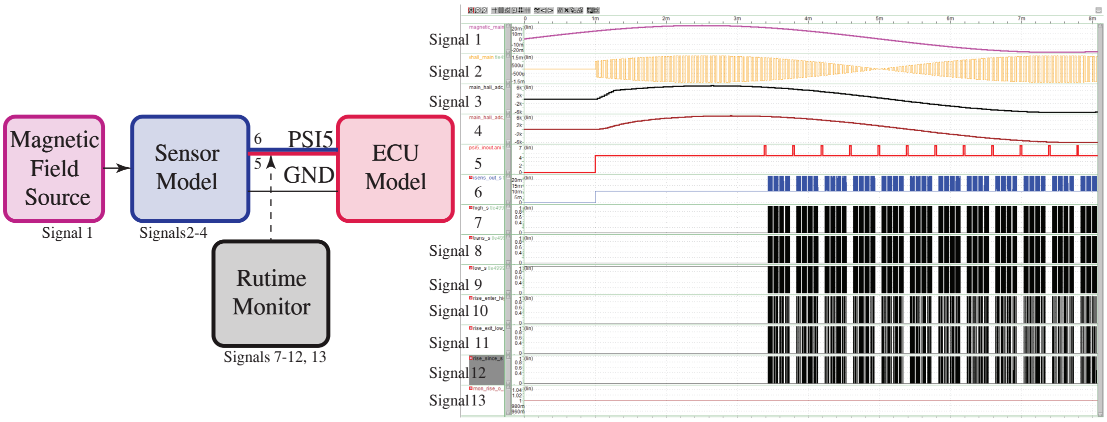


Figure 7.4: The runtime monitor in simulation: setup and results

Signal 13 is the output of the monitor, which, in this case, indicates that the requirement has been met.

To demonstrate the second use case, we generated the runtime monitors in FPGA and check the test chip. Fig. 7.5 illustrates the lab setup: To emulate the ECU we use a signal generator that sends synchronization pulses to the test chip. The sensor replies with data packets, which are handled by an Analog Front-End (AFE). In the FPGA the transmission line between the sensor and the ECU is modeled (Fig. 7.1) to facilitate an evaluation of various system integration scenarios. We generate hardware monitors using Vivado HLS and integrate them to the output of the transmission line, where we check the same requirements as in the chip simulation (i.e. “req_rise”). We use the Xilinx debug core to observe the communication between the sensor and the ECU and the outputs of our monitors on the ChipScope (Fig. 7.6).

7.4 Summary

This chapter elaborated on *the two use cases of runtime monitoring in the automotive electronic development*, and demonstrated their applicability in industrial context by checking the communication interface requirements of a steering-wheel magnetic sensor. We showed that runtime monitors can be included in a chip-concept simulation and monitors can be later reused for hardware-monitoring in-the-lab after applying HLS. Runtime monitoring in automotive electronic industry can be used as a tool to speed up the verification process and should be considered as an additional mechanism to capture runtime bugs which could be challenging to catch by classical in-the-lab approaches.

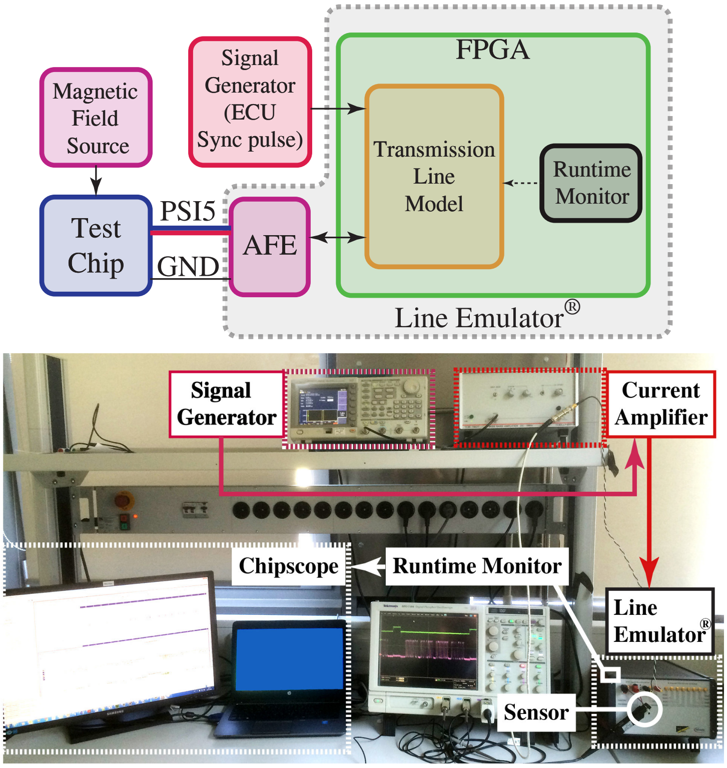
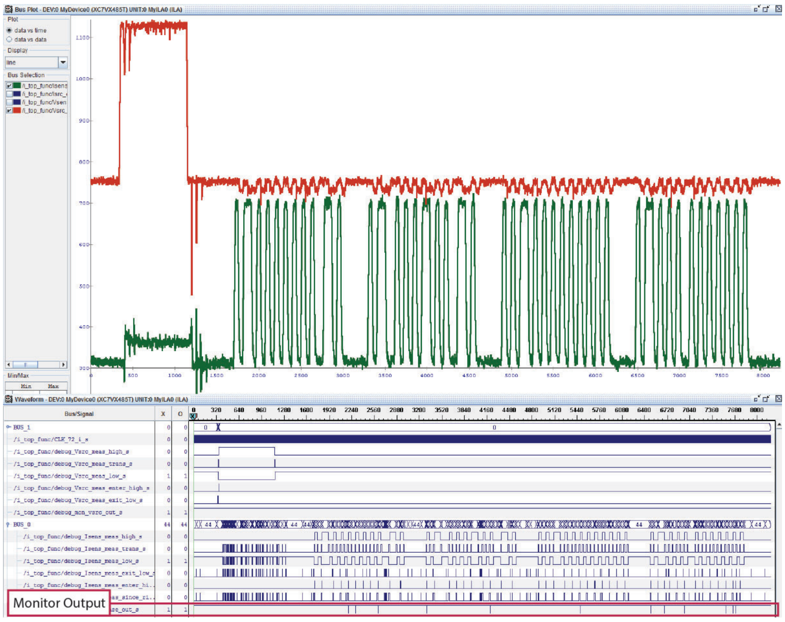


Figure 7.5: Runtime Monitor in Hardware: Lab Setup



Bus Plot - DEV:0 MyDevice0 (XC7VX485T) UNIT:0 MyILA0 (ILA)

Plot
● data vs time
○ data vs data

Display
line

Bus Selection
☑ 🟩 /i_top_func/Isens
☐ 🟦 /i_top_func/Isrc_
☐ 🟦 /i_top_func/Vsen
☑ 🟥 /i_top_func/Vsrc_

Waveform - DEV:0 MyDevice0 (XC7VX485T) UNIT:0 MyILA0 (ILA)

Bus/Signal	X	O
BUS_1	0	0
/i_top_func/CLK_72_i_s	0	0
/i_top_func/debug_Vsrc_meas_high_s	0	0
/i_top_func/debug_Vsrc_meas_trans_s	0	0
/i_top_func/debug_Vsrc_meas_low_s	1	1
/i_top_func/debug_Vsrc_meas_enter_high_s	0	0
/i_top_func/debug_Vsrc_meas_exit_low_s	0	0
/i_top_func/debug_mon_vsrc_out_s	1	1
BUS_0	44	44
/i_top_func/debug_Isens_meas_high_s	0	0
/i_top_func/debug_Isens_meas_trans_s	0	0
/i_top_func/debug_Isens_meas_low_s	1	1
/i_top_func/debug_Isens_meas_exit_low_s	0	0
/i_top_func/debug_Isens_meas_enter_hi..	0	0
as_since_ri..	0	0
se_out_s	1	1

Figure 7.6: Runtime Monitor in Hardware: Chip Scope Results

CHAPTER 8

Industrial Case studies and Evaluation

Previous chapter described the context of applying runtime monitoring in automotive electronic development, focusing on use cases at a system-level. In this chapter we elaborate in detail on runtime monitoring for two industrial protocols: the Single-Edge Nibble Transmission (SENT) and the Short PWM Code (SPC). We formalized and then runtime-monitored the subset of electrical and timing requirements of these protocols both in simulation and hardware. The protocols under study are mainly used in automotive applications, for instance, in an electronic power steering (EPS), or an electronic braking system (EBS). In these applications sensors transfer data about rotation of a steering wheel or position of a braking pedal, respectively; hence ensuring the correct information transfer and runtime error detection is of utter importance.

The current industry practice relies on hard-crafted checkers, that lack diagnostics information and do not runtime-check the signals on the electrical level. Existing tools for offline trace verification (e.g. the AMT [NM07]) are not directly applicable in this context, due to the excessive size of the resulting traces: e.g. if one records an analog signal, sampled at 70MHz, for an hour of runtime in an array of 16-bit integers, the trace will result in 504Gb of data. Moreover, it is also often the case that a long-term test takes several days of real-time execution. In order to be able to speed up the checking process and to produce the monitoring results during the execution of the system, we translate high-level specifications into monitors implemented in FPGA and run them in parallel with the system under investigation. We developed an approach that allows to observe the monitoring results in real time, track requirements to implementation, and report violation and debugging information for the higher level analysis.

To ease acceptance of runtime monitoring in industrial practice, we developed a graphical user interface (GUI) for substituting parameters in formalized STL specifications. We also compared two formalisms: STL and TRE to formalize the requirements of interest.

Main contributions of this chapter can be summarized as follows:

1. We developed a framework for generating monitors with recovery from a class of high-level specifications;
2. We formalized the electrical and timing requirements of the SENT and SPC;
3. We evaluated our framework on the real-world case study, demonstrating the synergy between formal methods and industrial practice in a real-world setting.

We use the definition of STL from Chapter 2 and introduce two useful macros in our notation, which capture the change in evaluation of a boolean component of w: for $p \in P$, **`enter`**$(p) = \circleddash \neg p \wedge p$ and **`exit`**$(p) = \circleddash p \wedge \neg p$. The standard semantics of the future operators, i.e. $\varphi_1 \mathcal{U}_I \varphi_2$, $\diamondsuit_I \varphi$, $\square_I \varphi$ is defined s.t., the satisfaction of the formulae at the time step i depends on events that happen in the future, namely at $(i + I) \cap \mathbb{T}$, which makes monitoring of these specifications acausal. To overcome such limitation, our hardware monitors comprise only past-temporal operators, and we use a procedure from [MNP07] to convert a formula with future operators to an equi-satisfiable past one.

As the goal of the case study is to produce runtime monitors in digital hardware (FPGA), the monitors operate on a finite representation of originally real-valued signals (ADC is used for quantization and sampling of continuously evolving voltage). For this purpose we interpret STL over discrete time and finite-valued domain. Let w be a multi-dimensional signal of a finite length, $w : [0, d] \mapsto P^n \cup X^m$, where $d \in \mathbb{N}$ is a duration of the signal; $P^n = \{p_1, \cdots, p_n\}$ and $X^m = \{x_1, \cdots, x_m\}$ are boolean (digital) and finite-domain (analog) variables respectively. Analog variables X^m are interpreted over a domain $\mathbb{D} = [0, \gamma] \subseteq \mathbb{N}$, where $\gamma = 2^r - 1, r \in \mathbb{N}$ is defined by a resolution of an ADC. The projection of the signal w to a component $e \in P \cup X$ is denoted by $\pi_e(w)$.

In the following section we briefly recap the TRE, the formalism that we use to obtain alternative formulation of electrical and timing requirements of the protocols.

8.1 Timed Regular Expressions

Timed regular expressions (TRE) [ACM02] allow to pattern-match a specification over a signal. As the authors in [FMNU15] mentioned, the fundamental difference between STL and TREs comes from a fact that the satisfaction of an STL formula is computed for a time point, while the match of a TRE results in a time interval. In this work we adapt the definition of TREs from [FMNU15] with an assumption of interpreting TREs over discrete time. To adhere to the definition from [FMNU15] and to allow negation in TREs, we make the following assumption: for every boolean variable $p_j \in P^n$ we admit a definition of a complementary variable $p_j^{\neg}$ with an opposite value of p_j (to which we refer as $\neg p_j$). Every analog variable $x_j \in X^m$ is allowed to be used in TREs only in the form of $x_j \sim c$, where $\sim \in \{< . \leq\}$ and $c \in \mathbb{D}$. With every $x_j \sim c$ we associate the boolean satisfaction variable $p_{x_j \sim c}$; we then analogously define $p^{\neg}_{x_j \sim c}$ and refer to it as $\neg(x_j \sim c)$.

A timed regular expression ψ is defined according to the following syntax [FMNU15]:

$$\psi := \epsilon \mid q \mid \psi_1 \cdot \psi_2 \mid \psi_1 \cup \psi_2 \mid \psi_1 \cap \psi_2 \mid \psi^* \mid \langle \psi \rangle_I$$

where q is of the form p, $\neg p$, $x \sim c$ or $\neg(x \sim c)$; I is a time interval $[a, b] \subseteq \mathbb{N}$.

For improved readability, we will refer to discrete time instance $i \cdot T$, where T is discrete time step, simply as i. The semantics of timed regular expression φ with respect to discrete signal w and time instances $i \leq i'$ is given in terms of satisfaction relation $(w, i, i') \models \varphi$:

$$\begin{array}{lll}
(w,i,i') \models \epsilon & \leftrightarrow & i = i' \\
(w,i,i') \models q & \leftrightarrow & i \leq i' \text{ and } \forall i'' \text{ s.t. } i \leq i'' < i', \pi_p(w)[i''] = 1 \\
(w,i,i') \models \varphi_1 \cdot \varphi_2 & \leftrightarrow & \exists i'' \text{ s.t. } i \leq i'' < i', (w,i,i'') \models \varphi_1 \text{ and } (w,i'',i') \models \varphi_2 \\
(w,i,i') \models \varphi_1 \cup \varphi_2 & \leftrightarrow & (w,i,i') \models \varphi_1 \text{ or } (w,i,i') \models \varphi_2 \\
(w,i,i') \models \varphi_1 \cap \varphi_2 & \leftrightarrow & (w,i,i') \models \varphi_1 \text{ and } (w,i,i') \models \varphi_2 \\
(w,i,i') \models \varphi^* & \leftrightarrow & (w,i,i') \models \epsilon \text{ or } (w,i,i') \models \varphi \cdot \varphi^* \\
(w,i,i') \models \langle \varphi \rangle_I & \leftrightarrow & i' - i \in I \text{ and } (w,i,i') \models \ \varphi
\end{array}$$

We reuse the notation $\{a\}$ for intervals of the form $[a, a]$. We introduce the following macros for describing transitions of a boolean signal: **`enter`** $(p) = \langle \neg p \rangle_{\{1\}} \cdot \langle p \rangle_{\{1\}}$ and **`exit`** $(p) = \langle p \rangle_{\{1\}} \cdot \langle \neg p \rangle_{\{1\}}$. We also use a superscript with a TRE to denote a number of concatenations of this TRE (e.g. if ψ is a TRE, then ψ^3 stands for $\psi \cdot \psi \cdot \psi$). Finally, we use ψ^+ as syntactic sugar for $\psi \cdot \psi^*$.

8.2 Formalization of the SENT and SPC Protocols

In this section we introduce the communication protocols under study: the Single Edge Nibble Transmission Protocol (SENT), and the Short PWM Code (SPC), and then formalize a subset of their electrical and timing requirements.

8.2.1 Single Edge Nibble Transmission Protocol

The SENT protocol is an industry standard (SAE J2716 [Int16]) for transmitting data between a sensor and a controller. SENT communication is unidirectional from a sensor to a controller; the information is partitioned into frames with the structure shown in Fig. 8.1. The transmitted data is split in four-bit data chunks, so-called nibbles, which encode the data in their length. Each nibble has the shape depicted in Fig. 8.2, where the length of the 'H' region determines the transmitted value (from 0 to 15). In the case study we build runtime monitors for magnetic sensors based on Hall effect, which transfer angular information encoded in the three data nibbles D1-D3.

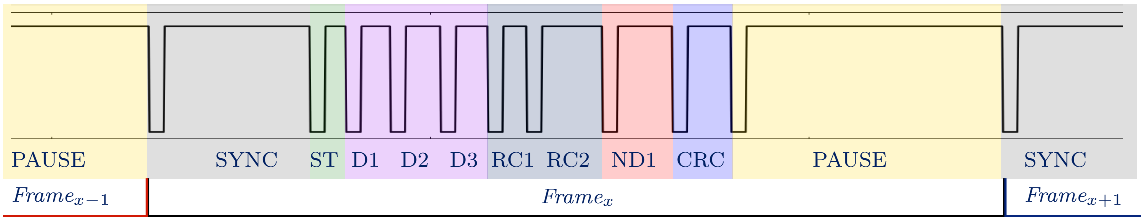


Figure 8.1: A SENT frame starts with a synchronisation pulse (SYNC), followed by a status nibble (ST), data nibbles (D1, D2, D3), rolling counters (RC1, RC2), bit inverse of D1 (ND1), cyclic redundancy check (CRC), and finishes with an optional pause.

The SAE J2716 standard admits several frame configurations (e.g. the number of data nibbles may vary). SENT devices are configured prior to operation, and the configuration does not change on-the-fly; we take this into account and also assume that the frame structure is static and cannot change at runtime.

Fig. 8.2 shows a SENT nibble and graphically depicts the requirements to be checked. Table 8.1 presents in natural language a subset of electrical and timing requirements of the SENT protocol.

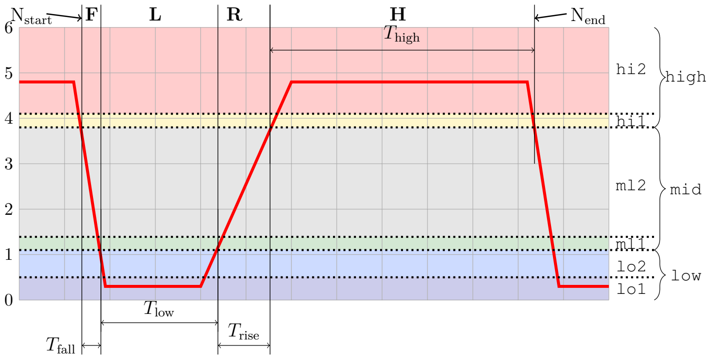


Figure 8.2: SENT nibble pulse: A pulse starts (N_{start}) with a falling edge F, followed by a low region L, followed by a rising edge R, followed by a high region H.

8.2.2 Short PWM Code

The SPC is an extension of the SENT protocol that allows bi-directional communication between a sensor and a controller. An SPC master (an ECU) initiates data transmission with a trigger pulse and a sensor responds with a SENT data packet. This allows to share the physical line between an ECU and up to four sensors, which transmit request and response as a modulated voltage. The SPC trigger pulse can either be of a variable or a fixed length, and the length of the low region of the pulse (t_{mlow}) identifies sensor id.

Table 8.1: SENT Requirements in natural language

	Electrical Interface Requirements	
1	The fall time from V_1 to V_2 must be no longer than T_{fall} μs	F
2	The rise time from V_2 to V_1 must be no longer than T_{rise} μs	R
3	The signal stabilization time below low threshold V_1 or above high threshold V_2 must be at least T_{stable} μs	$\texttt{ST}_{\texttt{low}}$,L $\texttt{ST}_{\texttt{high}}$
	Transmission Properties of Synchronization & Nibble Pulses	
4	The synchronization pulse shall have a nominal period of 56 clock ticks.	SYNC
5	Five clock ticks of the synchronization pulse shall be driven low.	L
6	All remaining clock ticks of the calibration / synchronization pulse shall be driven high.	SYNC, $\texttt{H}_{\texttt{sync}}$
7	Five clock ticks of the nibble pulse shall be driven low.	L
8	All remaining clock ticks of the nibble pulse shall be driven high.	NIBBLE, $\texttt{H}_{\texttt{nibble}}$
9	The minimum pulse period of the nibble pulse shall be 12 clock ticks.	NIBBLE, $\texttt{H}_{\texttt{nibble}}$
10	The maximum pulse period of the nibble pulse shall be 27 clock ticks.	NIBBLE, $\texttt{H}_{\texttt{nibble}}$

Figure 8.3 shows the SPC frames for the bus mode, and figure 8.4 shows the SPC trigger pulse and its requirements. Table 8.2 summarizes configuration of the SPC modes: we also assume that the configuration is static and cannot change at runtime.

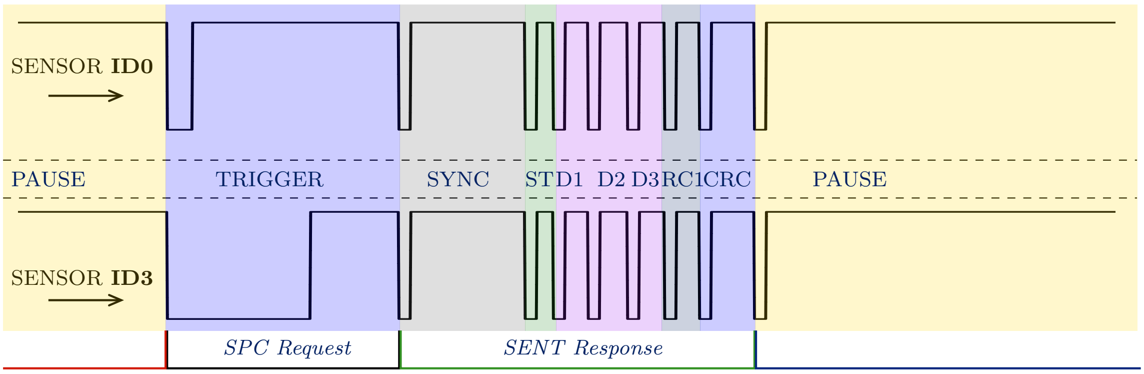


Figure 8.3: SPC bus mode: Specification signal for two sensors: ID0 and ID3 – communication starts with the trigger pulse, which is followed by a response from a sensor

To be able to correctly decode sensor data, a controller needs to receive a signal that satisfies electrical and timing requirements of the SENT protocol. For a sensor to respond

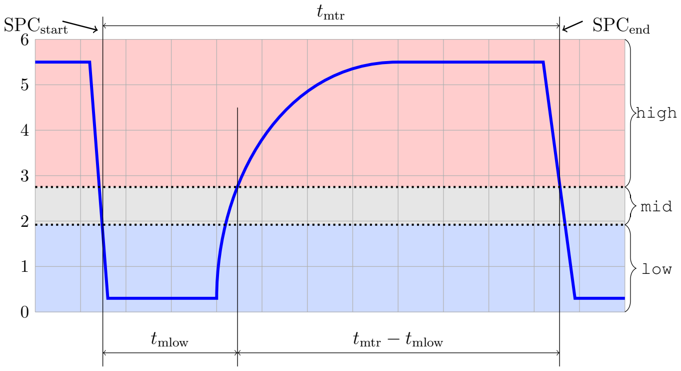


Figure 8.4: SPC trigger pulse: A pulse of duration t_{mtr} starts with a low duration t_{mlow}, which encodes the sensor id, followed by a high region

to a request, the trigger pulse needs to satisfy the requirements for the SPC trigger pulse for a corresponding sensor ID. We now state these requirements formally, both in STLand TRE and elaborate on checking the frame correctness.

8.2.3 Formalization of the SENT requirements in STL

Electrical Interface Requirements specify the duration of the slopes, as well as the minimum stable time of the SENT signal. The STL formulae (Eq. 8.1-8.4) capture the temporal order in which the signal should cross voltage regions from Fig. 8.2. `F` and `R` (Eq. 8.1, 8.2) are the formal representations of falling and rising time requirements (Tab. 8.1,Req.1,2). The signal stabilization requirement (Tab. 8.1,Req.3) is mapped to two STL formulae (Eq. 8.3, 8.4) that deal separately with both thresholds. The STL formulae are written using past temporal operators: in this type of formulation a consequent should have happened before an antecedent (i.e. the form "whenever at a time step i φ holds, ψ should have held at $(i - I) \cap \mathbb{T}$").

$$\texttt{F}= \textbf{enter}\texttt{(low)} \rightarrow \texttt{mid}\,\mathcal{S}_{[0,T_{\texttt{fall}}]}\ \textbf{exit}\texttt{(high)} \tag{8.1}$$

$$\texttt{R}= \textbf{enter}\texttt{(high)} \rightarrow \texttt{mid}\,\mathcal{S}_{[0,T_{\texttt{rise}}]}\ \textbf{exit}\texttt{(low)} \tag{8.2}$$

$$\texttt{ST}_{\texttt{low}} = \textbf{exit}\texttt{(low)} \rightarrow \boxminus_{[0,T_{stable}]}\ \texttt{low} \tag{8.3}$$

$$\texttt{ST}_{\texttt{high}} = \textbf{exit}\texttt{(high)} \rightarrow \boxminus_{[0,T_{stable}]}\ \texttt{high} \tag{8.4}$$

Transmission Properties of Synchronization & Nibble Pulses. The synchronization and the nibble pulse requirements (Tab.8.1, 4-6 and 7-10 respectively) describe the timing properties these pulses should adhere to. A synchronization pulse has a pre-defined length and is considered as the start of a SENT frame. The shape of synchronization and nibble pulses is to be checked as well (see Fig.8.2).

To verify the form of the synchronization, nibble, and pause pulses, we split each pulse in regions `F`, `L`, `R`, `H` (see Fig 8.2), check requirements for the corresponding region and the

Table 8.2: SPC Requirements in natural language

	Common Case Physical & Timing Requirements	
1	Rising and falling thresholds $V^{(th)}_{\text{rising}} = 0.5V_{dd}$ and $V^{(th)}_{\text{falling}} = 0.35V_{dd}$	
2	SPC unit time (UT): $1.5 \leq$ UT ≤ 3.0 μs, configurable in steps of $0.5\mu s$	
3	Sensor ID0 low time t_{mlow} in range from 9 to 12 UT	
4	Sensor ID1 low time t_{mlow} in range from 19 to 23 UT	
5	Sensor ID2 low time t_{mlow} in range from 35.5 to 40.5 UT	
6	Sensor ID3 low time t_{mlow} in range from 61.5 to 67.5 UT	
Config 1: **single** sensor with **fixed** length of the trigger pulse		
7	The total trigger time t_{mtr} is 90 UT	
8	Default sensor ID is 0 with t_{mlow} in range from 9 to 12 UT	
Config 2: **single** sensor with **variable** length of the trigger pulse		
9	The total trigger time $t_{mtr} = t_{mlow} + 12$ UT	
10	Default sensor ID is 0 with t_{mlow} in range from 9 to 12 UT	
Config 3: **multiple** sensor with **fixed** length of the trigger pulse		
11	The total trigger time t_{mtr} is 90 UT	
12	From two to four sensors on the bus with IDs from 0 to 3	
13	The low time t_{mlow} for each sensor corresponds to requirements 3-6	
Config 4: **multiple** sensor with **variable** length of the trigger pulse		
14	The total trigger time $t_{mtr} = t_{mlow} + 12$ UT	
15	From two to four sensors on the bus with IDs from 0 to 3	
16	The low time t_{mlow} for each sensor corresponds to requirements 3-6	

temporal precedence of the regions. The total length of the pulses and the length of the low region `L` are given in "clock ticks" (Tab. 8.1, 4-5, 7, 9-10), which are generated by a sensor's internal clock. The "clock tick" is also called unit time (UT), which is in the range between 1.5 and 3.0 μs with steps of 0.5 μs (see Req. 2 in Tab. 8.2). Let us denote $\delta = (T_{\text{rise}} + T_{\text{fall}})$, then the allowed durations of the `H` region for the nibble pulse and synchronization pulse are $[7\text{tick} - \delta, 22\text{tick} - \delta]$ and $(51\text{tick} - \delta)$, respectively. Similarly, the length of the `H` region of the pause pulse is: $[7\text{tick} - \delta, 122\text{tick} - \delta]$.

Requirements for `L` and `H` regions can be written directly in past-STL:

$$\mathtt{L} = \mathbf{exit}(\mathtt{low}) \rightarrow \boxminus_{[0,5\mathtt{ticks}]}\ \mathtt{low} \tag{8.5}$$

$$\mathtt{H}_{\mathtt{sync}} = \mathbf{exit}(\mathtt{high}) \rightarrow \mathtt{high}\,\mathcal{S}_{\{51\mathtt{tick}-\delta\}}\ \mathbf{enter}(\mathtt{high}) \tag{8.6}$$

$$\mathtt{H}_{\mathtt{nibble}} = \mathbf{exit}(\mathtt{high}) \rightarrow \mathtt{high}\,\mathcal{S}_{[7\mathtt{tick}-\delta,22\mathtt{tick}-\delta]}\ \mathbf{enter}(\mathtt{high}) \tag{8.7}$$

$$\mathtt{H}_{\mathtt{pause}} = \mathbf{exit}(\mathtt{high}) \rightarrow \mathtt{high}\,\mathcal{S}_{[7\mathtt{tick}-\delta,122\mathtt{tick}-\delta]}\ \mathbf{enter}(\mathtt{high}) \tag{8.8}$$

The general way of capturing precedence relation in STL is by using the bounded until operator $\mathcal{U}_I$. As the authors in [JBG+15] show, the hardware implementation of $\mathcal{U}_I$ is not scalable w.r.t. operator time bounds. In order to overcome this issue, we avoid using nested $\mathcal{U}_I$ operators in the formulation, and reformulate the properties. Each `SYNC`, `NIBBLE`, and `PAUSE` patterns of the SENT protocol are the requirements `F`, `L`, `R`, and the corresponding $\texttt{H}_{\{\texttt{sync|nibble|pause}\}}$ requirement put in a sequence. In order to attain efficient hardware implementation, we (i) re-state assertions from $\varphi \rightarrow \psi$ to $\psi \wedge \varphi$, to capture the events when the corresponding requirement has been satisfied; (ii) we then define precedence relation with following macro:

$$\varphi_1 \texttt{before}_{[t_1,t_2]} \varphi_2 = \varphi_2 \wedge \boxminus_{[0,t_1]} \neg\varphi_1 \wedge \diamondminus_{[t_1,t_2]} \varphi_1. \tag{8.9}$$

This allows to use hardware-cheap bounded historically $\boxminus_{[0,t_1]}$ and bounded once $\diamondminus_{[t_1,t_2]}$ operators and significantly reduce hardware resources.

The requirement for `NIBBLE` is then defined as follows (STL formulae for `SYNC` and `PAUSE` are constructed analogously):

$$\begin{aligned}\texttt{NIBBLE} = {} & (\texttt{F}\wedge\textbf{enter}\texttt{(low)})\ \texttt{before}_{[t_1,t_2]}\ (\texttt{L}\wedge\textbf{exit}\texttt{(low)})\ \texttt{before}_{[t_3,t_4]}\\ & (\texttt{R}\wedge\textbf{enter}\texttt{(high)})\ \texttt{before}_{[t_5,t_6]}\ (\texttt{H}_{\texttt{nibble}}\wedge\textbf{exit}\texttt{(high)})\end{aligned}$$

The top-level `FRAME` requirement captures precedence relation between `SYNC`, `NIBBLE`s, and the `PAUSE`, and manifests that requirements for the frame has been fulfilled:

$$\begin{aligned}\texttt{FRAME} = \texttt{SYNC}\ & \texttt{before}_{[t_7,t_8]}\ \texttt{ST}\ \ \texttt{before}_{[t_9,t_{10}]}\ \texttt{D1}\ \ \texttt{before}_{[t_9,t_{10}]}\ \texttt{D2}\ \ \texttt{before}_{[t_9,t_{10}]}\ \texttt{D3}\\ & \texttt{before}_{[t_9,t_{10}]}\ \texttt{RC1}\ \texttt{before}_{[t_9,t_{10}]}\ \texttt{RC2}\ \texttt{before}_{[t_9,t_{10}]}\ \texttt{ND1}\ \texttt{before}_{[t_9,t_{10}]}\ \texttt{CRC}\\ & \texttt{before}_{[t_9,t_{10}]}\ \texttt{PAUSE}\end{aligned}$$

The monitor construction is compositional: each sub-formula produces a satisfaction signal that is accepted at an upper level of hierarchy – a frame correctness is reported only when all the lower-level requirements for all the frame components (`SYNC`, `NIBBLE`s, `PAUSE`) are met. Time bounds $t_1 - t_{10}$ are derived from Equations 8.1-8.8.

8.2.4 Formalization of the SPC requirements in STL

According to the Tab. 8.2 SPC protocol supports four different static configurations, which define timing requirements for the SPC pulse from Fig. 8.4. To capture these requirements, we first formalize the low time requirement T_{mlow}:

$$\texttt{T}_{\texttt{mlow}} = \textbf{exit}\texttt{(mid)}\wedge\ \texttt{high}\wedge\ \texttt{(low}\vee\ \texttt{mid)}\,\mathcal{S}_{[t_{mlow,L},t_{mlow,H}]}\ \textbf{enter}\texttt{(low)}, \tag{8.10}$$

where $t_{mlow,L}, t_{mlow,H}$ are the limits from Table 8.2 (e.g. 9 and 12 UT for the ID0). The formula 8.10 is satisfied whenever the t_{mlow} for the corresponding sensor ID is met. We can now use this requirement to check the total time of the trigger pulse:

$$\texttt{T}_{\texttt{mtr}} = \textbf{exit}\texttt{(high)}\wedge\ \texttt{mid}\wedge\ \texttt{high}\,\mathcal{S}_{[t_{tmtr-tmlow,H},t_{tmtr-mlow,L}]}\texttt{T}_{\texttt{mlow}} \tag{8.11}$$

For each configuration from Tab. 8.2 we check the requirements 8.10 and 8.11 with the time bounds corresponding to the sensor ID.

8.2.5 Formalization of the SENT requirements in TRE

Although it is possible to formulate TREs in an STL-like style and express the same intent: e.g. the requirements $\texttt{F}^{\dagger}$ and $\texttt{R}^{\dagger}$ (Eq. 8.13 and 8.15) match falling and rising time intervals of the signal; using the syntax features of the TRE and composing the requirements hierarchically allows to obtain a concise formalization for the properties of interest. `F` and `R` regions (Eq. 8.12-8.14) are defined as follows:

$$\texttt{F}= \langle\texttt{mid}\rangle_{[0,T_{\texttt{fall}}]} \tag{8.12}$$

$$\texttt{F}^{\dagger}= \textbf{exit}(\texttt{high})\cdot\langle\texttt{mid}\rangle_{[0,T_{\texttt{fall}}]}\cdot\textbf{enter}(\texttt{low}) \tag{8.13}$$

$$\texttt{R}= \langle\texttt{mid}\rangle_{[0,T_{\texttt{rise}}]} \tag{8.14}$$

$$\texttt{R}^{\dagger}= \textbf{exit}(\texttt{low})\cdot\langle\texttt{mid}\rangle_{[0,T_{\texttt{rise}}]}\cdot\textbf{enter}(\texttt{high}) \tag{8.15}$$

The `L` TRE (Eq. 8.16) combines the requirements 3 and 5 from Table 8.1. The `H` TRE (Eq. 8.17) will match when the requirement 3 is fulfilled. The two are the necessary building blocks for checking the shape of pulses:

$$\texttt{L}= \langle\texttt{low}\rangle_{[T_{\texttt{stable}},5\texttt{tick}]} \tag{8.16}$$

$$\texttt{H}= \langle\texttt{high}\rangle_{[T_{\texttt{stable},123\texttt{tick})}} \tag{8.17}$$

We are now able to define the TRE for synchronization, nibble, and pause pulses as a concatenation of regions, restricting the length of the pulses with appropriate time bounds. The `SYNC` TRE (Eq. 8.18) will match only when the requirements 1-6 (Tab. 8.1) are met. The sensor signal will match the `NIBBLE` TRE (Eq. 8.19) if the requirements 1-3, 7-10 are fulfilled. The pause pulse requirements are captured by Eq. 8.20):

$$\texttt{SYNC} = \langle\texttt{F}\cdot\texttt{L}\cdot\texttt{R}\cdot\texttt{H}\rangle_{\{56\texttt{tick}\}} \tag{8.18}$$

$$\texttt{NIBBLE} = \langle\texttt{F}\cdot\texttt{L}\cdot\texttt{R}\cdot\texttt{H}\rangle_{[12\texttt{tick},27\texttt{tick}]} \tag{8.19}$$

$$\texttt{PAUSE} = \langle\texttt{F}\cdot\texttt{L}\cdot\texttt{R}\cdot\texttt{H}\rangle_{[12\texttt{tick},127\texttt{tick}]} \tag{8.20}$$

The frame and protocol requirements in TRE are formulated as follows:

$$\texttt{SENT\_FRAME} = \texttt{SYNC}\cdot\texttt{NIBBLE}^{8}\cdot\texttt{PAUSE} \tag{8.21}$$

$$\texttt{SENT\_PROTOCOL} = (\texttt{SENT\_FRAME})^{+} \tag{8.22}$$

8.2.6 Formalization of the SPC requirements in TRE

The requirements for the SPC trigger pulse are formalized in Eq. 8.23 and 8.24. The corresponding TREs match sequences of events; each requirement formalizes the temporal order in which the signal should cross voltage regions from Fig. 8.4. For a bus mode, a monitor with the time bounds corresponding to the sensor ID will be constructed:

$$\mathtt{T}_{mlow} = \langle \mathtt{low} \cdot \mathtt{mid} \rangle_{[t_{mlow,L}, t_{mlow,H}]} \tag{8.23}$$

$$\mathtt{T}_{mtr} = \langle \mathtt{T}_{mlow} \cdot \mathtt{high} \rangle_{\{t_{mtr}\}} \tag{8.24}$$

8.3 Runtime Monitoring with Recovery

A runtime monitor typically partitions the execution traces in those that either satisfy or violate system's specification, possibly providing a quantitative metric of satisfaction (violation). However, for data-driven applications, such as serial protocols, test executions may last for hours and it is required to continue monitoring even after detecting errors, and not stop after the first violation, marking a trace as invalid, since even some amount of errors might be foreseen by protocol designers and should be tolerated. Similarly to compilers, a monitor in such a case must be able to recover after observing a violation, collect the encountered errors, and report them to the user.

For a class of serial protocols, the asynchronous serial protocols (e.g. SENT [Int16], RS-232 [Axe07], DMX512 [R2008], etc.), we propose a procedure to construct monitors with *error recovery*. To apply monitoring with recovery, the protocol must fulfil the following requirement: the devices communicate over a single line, where synchronization symbol, control and payload data, respectively, are multiplexed in time. As control signals are absent, the devices rely on the synchronization symbol to successfully capture the beginning of a useful portion of a frame.

By creating runtime monitors with recovery, we are able to: (i) Continue monitoring after detecting violations; (ii) Collect the errors and report them with their violation type.

8.3.1 TRE Monitors with Recovery

In the case of asynchronous serial protocols, the devices communicate with sequences that form certain patterns over time; the communication is cyclic, where the data is split in subsequently following frames. These protocols admit a formalization in TREs: A frame begins with a unique synchronization pattern (`START`), followed by n `PAYLOAD` patterns, and ends with a `STOP` pattern. The asynchronous serial protocol is then defined as a sequence of frames:

$$\mathtt{ASYNC\_SERIAL\_PROTOCOL} = \mathtt{FRAME}^{+}, \tag{8.25}$$

$$\mathtt{FRAME} = \mathtt{START} \cdot \mathtt{PAYLOAD}^{n} \cdot \mathtt{STOP}. \tag{8.26}$$

The above expression exactly generalizes the TRE formalization of the SENT protocol from Section 8.2.5 (Eq. 8.21 and 8.22). It is important to mention that the Kleene star

(*) operator should not be used in the specification of `START`, `PAYLOAD` and `STOP` in TREs, as these patterns are finite sequences of symbols; we use the Kleene star operator only at the top TRE (i.e. Eq. 8.25).

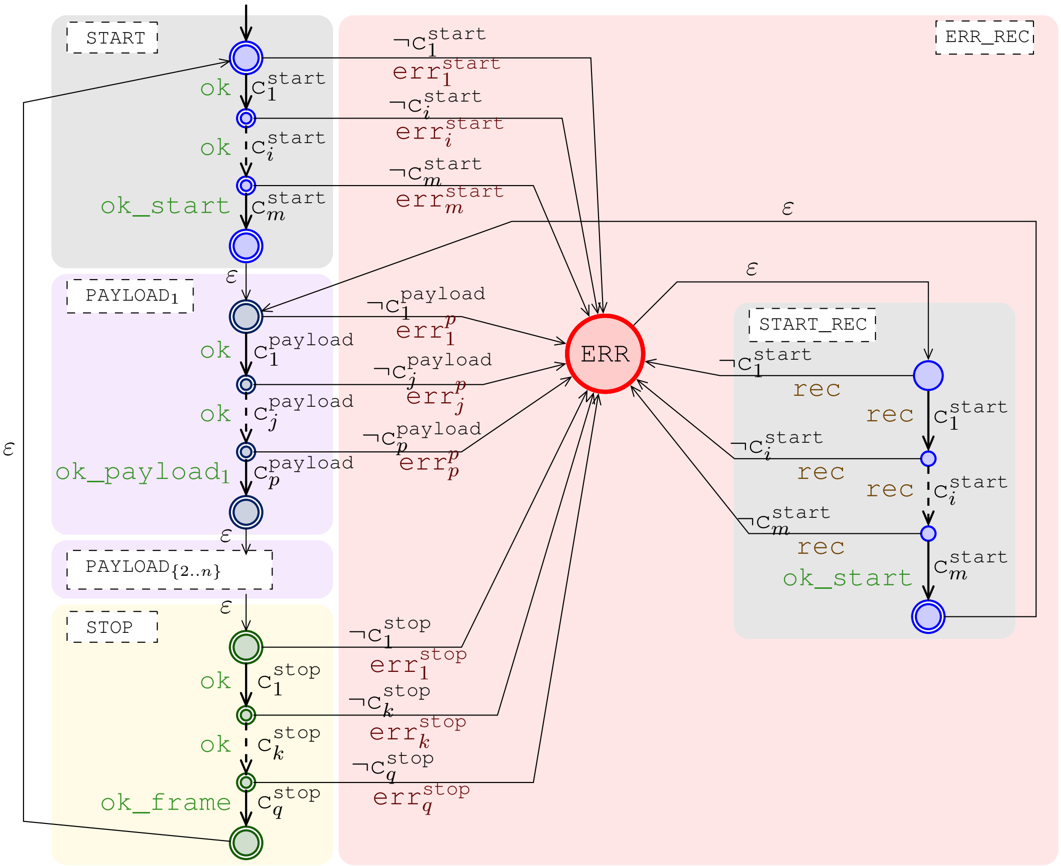


Figure 8.5: Monitoring an asynchronous serial protocol with recovery[1]

The sketch of construction procedure for a monitor with recovery is shown in Fig. 8.5. For each of the `START`, `PAYLOAD`, and `STOP` patterns, we construct the corresponding automata with discrete-time clocks $\mathcal{A}_{\texttt{start}}$, $\mathcal{A}_{\texttt{payload}}$, and $\mathcal{A}_{\texttt{stop}}$, respectively. We also create an additional copy of $\mathcal{A}_{\texttt{start}}$, called $\mathcal{A}_{\texttt{rec}}$, which enables the runtime monitor to recover from an error. In this work we take an optimistic approach, and use a weak interpretation of regular expression over finite traces. In case when a trace ends and only a prefix of the regular expression is matched, we decide to accept the input sequence. Therefore all the states in $\mathcal{A}_{\texttt{start}}$, $\mathcal{A}_{\texttt{payload}}$, and $\mathcal{A}_{\texttt{stop}}$ are accepting. The automaton-construction procedure from a given TRE, is adopted from [ACM02] to the discrete interpretation of time. The state transitions are protected by a set C of symbolic transition guards C, where $C = \{c_1^{\texttt{start}}, \ldots, c_m^{\texttt{start}}, c_1^{\texttt{payload}}, \ldots, c_p^{\texttt{payload}}, c_1^{\texttt{stop}}, \ldots, c_q^{\texttt{stop}}\}$.

---

[1] For clarity of the presentation, we keep ε-transitions in the Figure 8.5; these transitions are removed in implementation though keeping the monitor deterministic.

For each $c_i \in C$ we associate a complementary transition $\neg c_i$ to the global error state. The error state silently transitions to the starting state of the recovery automaton $\mathcal{A}_{\texttt{rec}}$ which consumes garbage symbols until a correct synchronization symbol is observed. The correct `START` pattern is a necessary pre-requisite for a monitor to analyze subsequent frames, and for the decoder to analyze the transferred data: as long as the synchronization symbol of the next frame is not received, $\mathcal{A}_{\texttt{rec}}$ goes back to the error state.

We introduce a diagnostic variable `out`, defined over a finite set of symbolic values: $\{$`ok`, `ok_start`, `ok_payload`$_{1,\ldots,N}$, `ok_frame`, `rec`, `err`$_{1,\ldots,m}$ $\}$. The values have the following meaning: `ok`: the trace has been correct so far; `ok_start`: the starting synchronization symbol has been matched; `ok_payload`$_i$: the i^{th} payload symbol has been matched; `ok_frame`: the frame has met all the requirements; `rec`: the monitor is in the recovery state; `err`$_i$: the specification is violated by an error of type i.

We then transform $\mathcal{A}_{\texttt{start}}$, $\mathcal{A}_{\texttt{payload}}$, $\mathcal{A}_{\texttt{stop}}$ and $\mathcal{A}_{\texttt{rec}}$ to transducers $\mathcal{A}'_{\texttt{start}}$, $\mathcal{A}'_{\texttt{payload}}$, $\mathcal{A}'_{\texttt{stop}}$ and $\mathcal{A}'_{\texttt{rec}}$ as follows: (i) For each transition in $\mathcal{A}_{\texttt{i}}$, we output `ok` value; (ii) For each transition leading to a sink state, we output appropriate `ok_{start|payload|frame}` value; (iii) For each transition guarded by $\neg c_i$ we output `err`$_i$; (iv) For each recovery automaton transition, except the synchronization symbol matching transition, we associate `rec` value. The transition in $\mathcal{A}'_{\texttt{rec}}$ which matches synchronization symbol outputs `ok_start` (see Fig. 8.5). For the top-level expression `FRAME`, we create the automaton $\mathcal{A}_{\texttt{frame}}$ by concatenating the $\mathcal{A}_{\texttt{start}}$, $\mathcal{A}_{\texttt{payload}}$, and $\mathcal{A}_{\texttt{stop}}$ with ε transitions. This way the user is capable to receive the information about the number of frames that meet the specification, as well as errors and their type.

8.3.2 STL Monitors with Recovery

The STL monitors are transducers (temporal testers [PZ08a]) by construction and are composed hierarchically to output the satisfaction signal of the top-level requirement. The sketch of construction procedure for monitoring with recovery is as follows: (i) we first formalize the `START`, `PAYLOAD`, and `STOP` patterns in STL; (ii) we then change the semantic meaning of STL assertions from (1) $\varphi \to \psi$ to (2) $\varphi \wedge \psi$: in the first formulation the transducer outputs '1' even if the requirement has never been checked, and '0' when the requirement has been violated (e.g. the `F` requirement from Sec. 8.2.4 is fulfilled even the line stays always at '1'); the second case the transducer manifests with the signal the precise time stamp when the requirement has been satisfied (i.e. outputting '1' when the correct falling edge occurred); (iii) for each requirement we identify a set of possible violations and assign an error code `err`$_i$ to each violation type. Each violation is guarded by an STL assertion $\varphi \wedge \neg\psi \wedge v_i$, where v_i identifies a violation type (e.g. `mid` $\mathcal{S}_{[T_{\text{fall}+1},\infty)}$ `exit(high)` is a v_i clause to capture the violation of the type "too slow falling time" for the STL assertion `F` from Sec. 8.2.4).

Finally we check the temporal precedence of the `START`, n `PAYLOAD` sequences and the `STOP` pattern with the `before`$_{[t_1,t_2]}$ macro defined in Sec. 8.2.4. Using temporal testers allows to monitor all the requirements in parallel, and extending with violation clauses v_i provides the necessary debugging information.

8.4 Runtime Monitoring of SENT and SPC protocols

This section describes building runtime monitors in FPGA and evaluating the results in industrial environment. A general overview of the framework is followed by implementation and evaluation details. Appendix A gives details of performing runtime monitoring.

8.4.1 From Requirements to Hardware Monitors

Fig. 8.6 summarizes the process of creating runtime monitors; the proposed framework is not limited to the SENT, or the SPC and can be applied for other protocols as well.

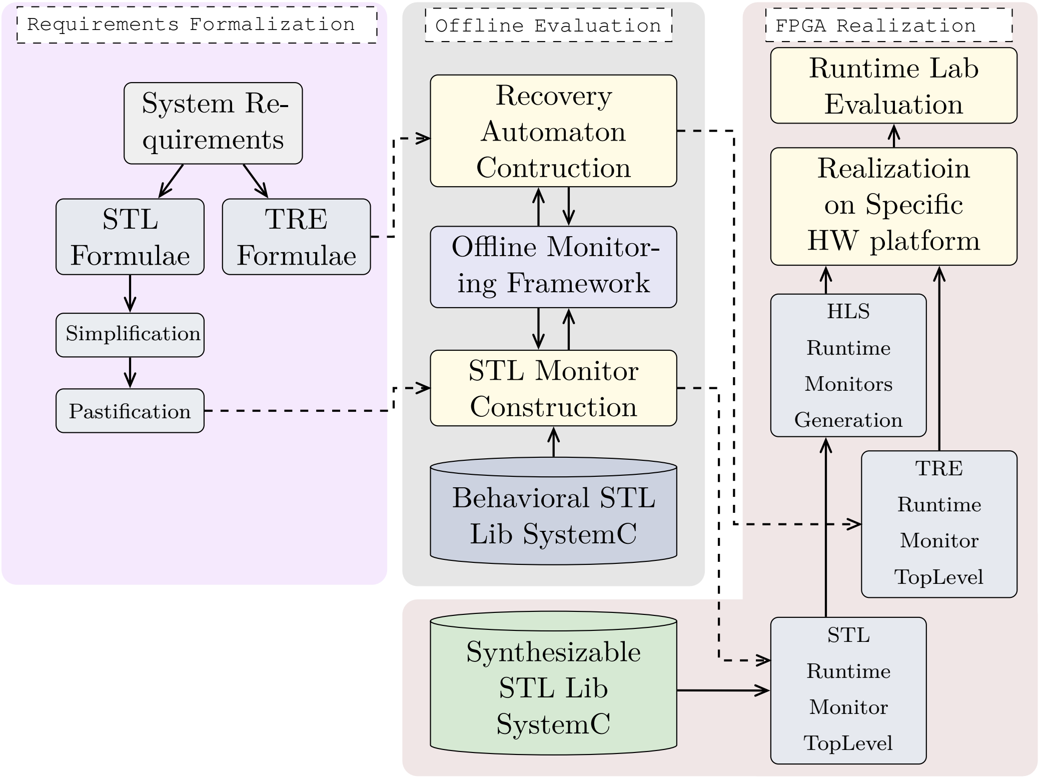

Requirements Formalization
System Requirements
STL Formulae
TRE Formulae
Simplification
Pastification
Offline Evaluation
Recovery Automaton Contruction
Offline Monitoring Framework
STL Monitor Construction
Behavioral STL Lib SystemC
FPGA Realization
Runtime Lab Evaluation
Realizatioin on Specific HW platform
HLS Runtime Monitors Generation
TRE Runtime Monitor TopLevel
Synthesizable STL Lib SystemC
STL Runtime Monitor TopLevel

Figure 8.6: Monitor Generation

Requirements Formalization. The initial step for creating runtime monitors is to obtain formal representation of the system requirements. Formal semantics allows to eliminate ambiguities in interpretations and precisely define what is to be monitored. In order to evaluate the power of different formalisms, and to eliminate “single source of truth" from the system we used two specification languages: STL and TRE. This phase results in a set of formulae (STL & TRE) which describe natural-language requirements.

For STL requirements we admit an automated pre-processing step (see Fig. 8.6) to obtain formulae that allow efficient hardware realization: on the parse tree of the formula we (i) eliminate duplicate sub-trees (Simplification); (ii) apply a recursive procedure from [MNP07] to convert bounded future STL temporal operators to an equi-satisfiable past operators, resulting in a causal formula with the past temporal operators only (Pastification). The second step is achieved by (i) calculating the temporal depth $\mathcal{D}$ of the formula; (ii) re-writing a formula with past operators which results in postponing a monitoring verdict by $\mathcal{D}$. For electrical and timing requirements some of the timing parameters are given in UT, at this step the time bounds in monitor clock are calculated.

Offline Evaluation. In this phase we evaluate monitors offline on short trace fragments, previously recorded from an oscilloscope or an ADC via the Chipscope [Inc16] in order to speed-up debugging and identify implementation bottlenecks.

The monitors for STL formulae are built compositionally from the formula parse tree [PZ08a]. With each node of the STL parse tree, which represents either a temporal or a logical operator, we associate a transducer $\mathcal{T}$ which takes as inputs satisfaction signals of its child nodes and outputs the satisfaction signal for the corresponding operator. The satisfaction signal of the root node produces output of the monitor. *Behavioral STL Lib SystemC* (see Fig. 8.6) is a SystemC implementation of STL transducers, which are used to obtain a monitor. We use SystemC simulation kernel to run the monitor on the pre-recorded traces.

The runtime monitors for the TRE requirements are also implemented in hierarchical fashion: the $\mathcal{A}'_{\texttt{sync}}$, $\mathcal{A}'_{\texttt{nibble}}$, and $\mathcal{A}'_{\texttt{stop}}$ transducers are combined in the top-level recovery automaton described in Section 8.3.1. We use Vivado Behavioural Simulation to evaluate VHDL code of the top-level $\mathcal{A}'_{\texttt{frame}}$ transducer.

Runtime Monitoring in FPGA During this final phase the monitors are synthesized in a digital reconfigurable hardware and evaluated in the lab environment. After the off-line phase we obtain the validated monitors for STL and TRE, which follow different paths of hardware implementation.

In case of STL monitoring, we use High-Level Synthesis [Inc] (HLS) to generate HDL code for monitors written in SystemC. During the HLS step, the SystemC monitors are transformed to an equivalent synthesizable VHDL or Verilog. We use an alternative implementation of transducers (*Synthesizable STL Lib SystemC*, Fig. 8.6), which is suitable for HDL code generation. *Behavioral* and *Synthesizable* implementations are functionally equivalent, but HLS imposes constraints on the SystemC code to be hardware-synthesizeable. Keeping *behavioral* and *synthesizable* versions allows quick prototyping using all `C++` features and then produce a hardware-optimized *synthesized* version.

Since transducers $\mathcal{A}'_{\texttt{sync}}$, $\mathcal{A}'_{\texttt{nibble}}$, $\mathcal{A}'_{\texttt{stop}}$, and $\mathcal{A}'_{\texttt{frame}}$ in the TRE approach are implemented in VHDL, we directly use Vivado Synthesis, Logic & Power Optimization, Place & Route tools to obtain a bitstream for FPGA programming.

8.4.2 FPGA Implementation

We implemented runtime monitors for the SENT protocol in Xilinx Virtex 7 FPGA. The monitors are embedded in the *Line Emulizer* hardware (see Fig. 8.7), which combines an AFE capable to interface various sensors with a high-performance Virtex 7 FPGA. This hardware also models a physical transmission line with adjustable parameters between a sensor and an ECU.

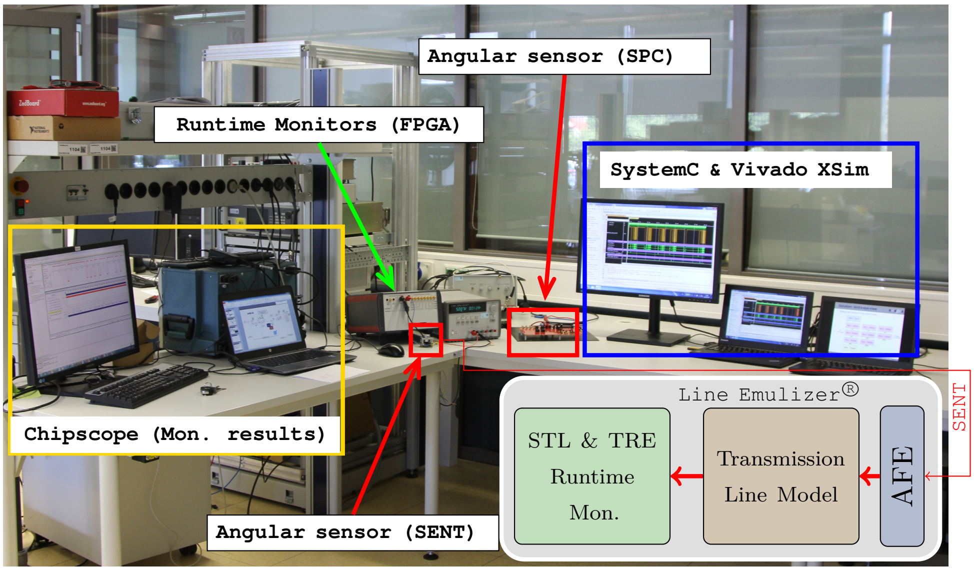


Figure 8.7: Runtime Monitoring of the SENT: Hardware Setup

The signal from the SENT sensor (see Fig. 8.7) comes to the *Line Emulizer*, where it is passed through the AFE and sampled with a high-speed ADC, which results in its finite value representation. During operation in a car, a sensor and an ECU are placed in different locations, hence the sensor signal is affected by a transmission line. To take into account the effects of physical wires, the sensor signal is passed through a digital model of a transmission line. We attach the STL and TRE runtime monitors at the end of the transmission line model (see Fig. 8.7), to be able to report specification conformance at the receiver side, which is important for proper signal decoding.

The STL and TRE monitors observe at 70 MHz the sensor signal affected by the physical line, calculate verdicts at every clock cycle (i.e. 70 million times per second), and output the result to the user via the Chipscope (Fig. 8.7). We performed experiments with different models of the line, and conclude that the appropriate line parameters are critical for ensuring the specification compliance. The sensor signal passed through a line with a higher capacitance violates the specification, since the falling and rising times are not met, which can be directly observed from the monitor.

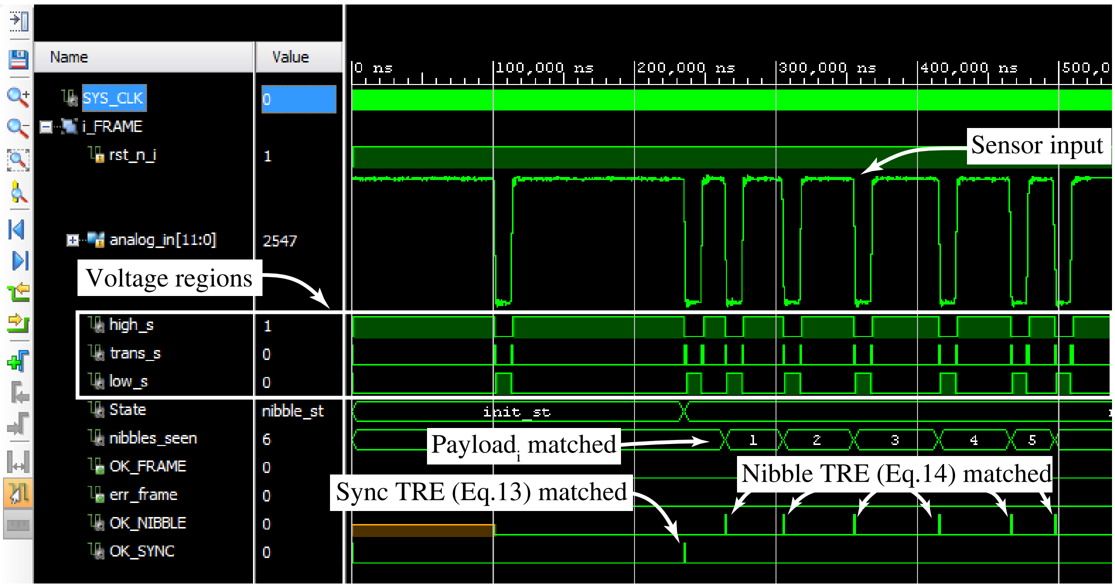


Figure 8.8: Runtime TRE monitoring: Vivado functional simulation

Table 8.3 reports the estimated FPGA hardware resources (flip-flops, FF & look-up tables, LUT), and the estimated maximum clock period of the runtime monitors. For each HLS-generated monitor we also present its generation time and peak memory usage during HDL-code generation. The monitors in HLS are constructed in a hierarchic fashion, hence the `FRAME` monitor (see Tab. 8.3) subsumes monitors for other requirements and results the highest hardware footprint. The last row of the Tab. 8.3 reports the total hardware resources consumed by the top-level TRE monitor: the direct hardware implementation results in an order of magnitude lower footprint.

Fig. 8.8 shows a result of offline evaluation for TRE requirements. The original SENT signal is observed by the monitor, which outputs `OK_NIBBLE`, `OK_SYNC` and the corresponding `ERR` signals. The figure depicts a nominal case, where all the requirements are met. Runtime Monitoring of the SPC protocol is illustrated in Fig. 8.9, which captures nominal and violation scenario of a missing frame. In this case an ECU sends a trigger pulse, but sensor does not respond back with a data packet. Runtime monitor is capable to capture this rare event, and then statistics over a time period can be collected.

Runtime monitoring of the SENT signal against STL requirements is shown in the Fig. 8.10. For this test case the optional pause pulse was deactivated, hence the correct frame is manifested after observing eight correct nibbles (signals `OK_NIBBLE, OK_SYNC, OK_FRAME`). The `OK_NIBBLE` signal is asserted when the corresponding precedence between the requirements `F`, `L`, `R`, and `H` is met. The output of the monitors `F`, `L`, `R`, and

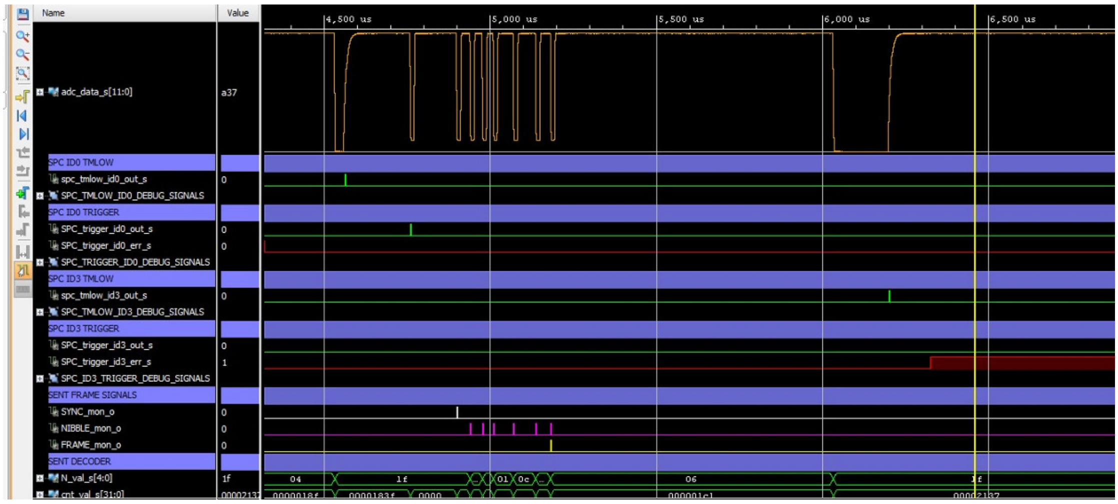


Figure 8.9: Runtime monitoring of SPC protocol

Table 8.3: STL Monitors Generation: FPGA & HLS resources

Requirement		FF	LUT	Clock	HLS: Time	HLS: Memory
F	HLS	61	118	5.81ns	114.203s	225MB
L		53	85	4.24ns	96.490s	159MB
R		61	113	5.81ns	109.784s	224MB
H_{nibble}		125	249	5.81ns	175.716s	225MB
H_{sync}		28	407	5.81ns	253.507s	224MB
H_{pause}		73	98	4.24ns	162.637s	212MB
NIBBLE		435	1123	7.7ns	394.671s	611MB
SYNC		207	1062	7.7ns	723.690s	605MB
PAUSE		217	710	7.7ns	206.767s	317MB
FRAME		1198	4322	7.7ns	1675.52s	1.39GB
FRAME	TRE	68	350	4.5ns	-	-

H, and the corresponding sub-formulae are presented in the lower part of the Fig. 8.10.

8.5 Summary

The case study focuses on assessing STL and TRE for formalizing requirements of the SENT and SPC protocol and obtaining hardware monitors for these requirements. We showed the application flow of runtime monitoring for industrial-standard protocols from formalization of electrical and timing requirements to generating hardware monitors.

The hardware resource consumption in Tab. 8.3 shows that (i) both approaches can be easily mapped to state-of-the-art FPGAs, (ii) STL-based monitors consume an order of magnitude more resources than the TRE monitors. Obtaining hardware monitors based on STL Synthesizable-SystemC library requires an intermediate transformation using HLS, which comes at price of increased hardware footprint.

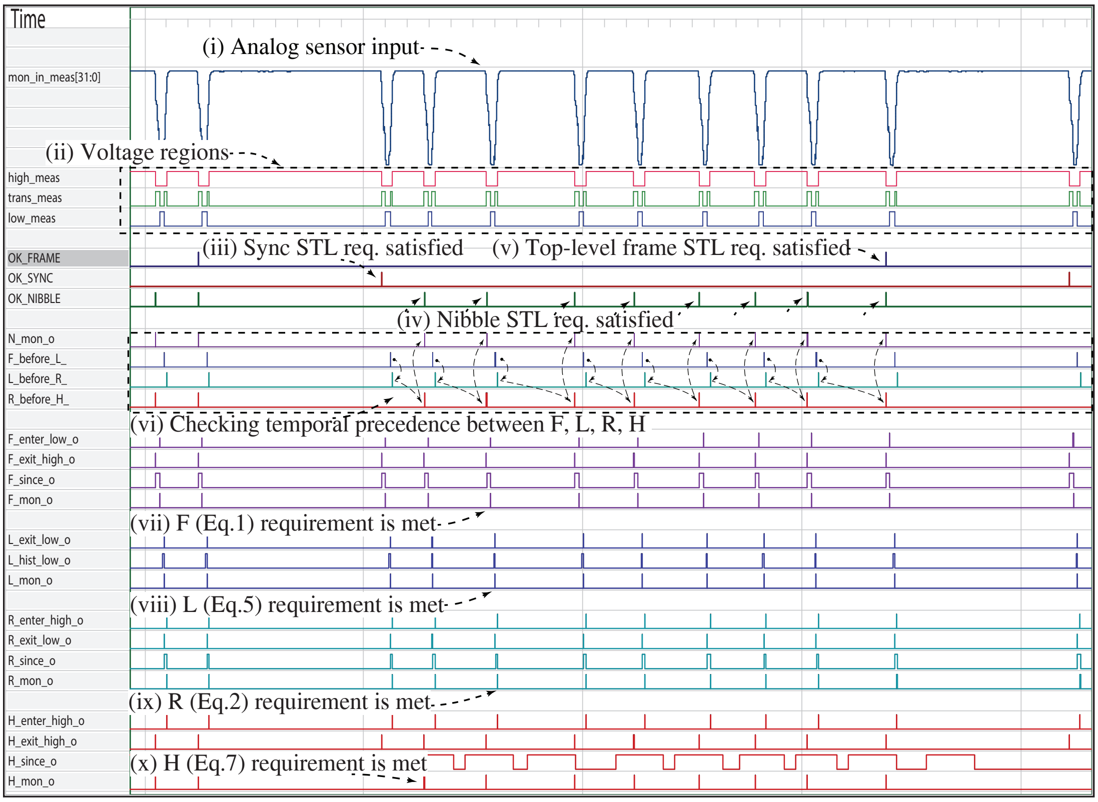


Figure 8.10: Runtime monitoring of the STL requirements

Besides low-level hardware monitoring, which both of the approaches facilitate, SystemC STL monitors can be re-used to check SystemC models. Trace verification in this setting happens during the runtime of the simulation kernel and the monitoring results are obtained at the end of the run. The re-usability of HLS-based monitors though comes at price of FPGA resource consumption.

We found both formalisms applicable for the SENT requirements formalization. TREs allow natural formulation of requirements that are concerned with repetitive sequences of groups of symbols, while formalizing precedence constraints with STL requires in general additional effort to be hardware-efficient.

CHAPTER 9

Conclusions and Future Work

The chapter summarizes the results and concludes the thesis. Section 9.1 outlines the research results and presented solutions. Section 9.2 gives critical reflections, discusses the limitation and possible improvements of the work. Section 9.3 aligns the research questions, defined in Section 1.3 with the outcomes of the thesis. Section 9.4 presents directions for future research, that can be performed using the results of the thesis.

9.1 Summary of Contributions

Challenges in CPS require development of novel approaches in design, analysis, and maintenance of such systems. Architectures based on neural models are getting more traction both in research and industry, allowing to solve complex problems [Amp10]. In the thesis we addressed how neural models can be used in CPS control and monitoring, as well as developed a flow to runtime monitor industrial protocols in FPGA hardware.

First, we considered how uncertainty in making decisions in CPS can be incorporated in the controller and put in correspondence with neural circuits. This allows updating the parameters of neural models from successful traces and adapting using controllable variance, which changes during the run of the system.

Second, we showed how TrueNorth spiking neural model, which allows efficient hardware implementation, can be used to monitor temporal logic specifications. We started with qualitative approach, and then showed how the model can be used for computations of arithmetic functions over the spike rates, and introduced a way compute quantitative semantics with the TrueNorth model using circular convolution.

Third, we developed a flow to runtime monitor requirements of industrial protocols in FPGA. To speed up the testing process and enable long-term tests, we implemented a procedure to create HDL monitors using HLS from formal requirement representation, and showed its applicability for SENT and SPC protocols. We also formalized the electrical and timing requirements of the aforementioned protocols.

9.2 Critical Reflections

9.2.1 Reflections on neural models for monitoring

The authors describe a dedicated chip in [CMA$^+$13] for synthesizing TrueNorth neurons directly in hardware, which is not available at the moment. Using FPGA although allows to create neural monitors, should be compared with the dedicated TrueNorth chip as the programmable logic of the Zynq chip uses general purpose logic cells.

Rate encoding when performing arithmetic operations and quantitative monitoring illustrates the trade-off between time delay and a number of TrueNorth neurons. Performing operations in over spike rates results in a more compact circuit. Yet for the cases, when the input is available reading and processing it in parallel can be faster, which comes at price of an increased hardware footprint.

9.2.2 Reflections on runtime monitoring in electronic development

The case study presented in Chapter 8 received high evaluation score from Infineon, where it has been evaluated by a concept design engineer and a verification engineer (the results of a user study after the evaluation are summarized in Appendix 2). The work during the case study though revealed several open issues, which are described below.

On the conceptual level there is a gap between textual and formal requirements, which means that on one hand, it is the state-of-the-art practice to have textual requirements as a binding contract between the parties during product design, and, on the other hand having unambiguous formal representation requires significant expert knowledge, and often poses difficulties to electrical engineers without additional training when first time exposed to the formal specification languages. The ways to mitigate the problem are: (i) applying parametric STL [ADMN12], where the core functionality is formalized by an expert, and the parameters for particular protocol, timing and voltage levels are substituted by an engineer. This option, which would lead the most rapid acceptance, though requires an expert to formalize each protocol separately. (ii) providing formal training for verification engineers in the long run, (iii) developing techniques in specification mining to extract common patterns from typical requirements of the sensor products, and provide common building blocks in a form of a assertions library.

On the implementation level, the limitations of the HLS code generation allows only partial re-use between the synthesizable and behavioral code. All the parameters of the monitor should be known at compile time, to be able to generate the hardware using HLS. The fact that the HDL code for the monitor should be generated first, and then integrated in the hardware design, which needs to be re-synthesized each time the monitor code changes is also seen as a limitation. To tackle this problem and shorten hardware synthesis cycles, exploration of partial reconfiguration of FPGA parts is required.

9.3 Research Questions Revisited

RQ 1: *How to design controllers for CPS using neural models?*

This research question has been addressed in Chapter 4. We showed that neural models can be put in correspondence with the programming structures for CPS controller design. The parameters of the neural controller then can be updated incrementally, based on results of the experiments. Each action, taken by a controller, is characterized by an variance σ which quantifies uncertainty of making decisions.

RQ 2: *How to perform qualitative and quantitative monitoring of temporal properties of CPS at runtime using neural models?*

We have addressed this question in Chapters 5 and 6, where we showed how to apply digital spiking neural model for qualitative monitoring of temporal logic specifications. Configuring parameters of the TrueNorth neuron allows to obtain temporal testers, which are then combined hierarchically to obtain a monitor for the specification of interest. In Chapter 5 we interpreted neural spikes as events on the time axis. In the continuation of this work in Chapter 6 we considered encoding in spike rates, which allows representing quantitative data in a simulation trial. Based on this representation we showed how to compute arithmetic functions over spike rates and use circular convolution to perform quantitative monitoring of temporal logic formulae.

RQ 3: *How to speed up the testing process in automotive electronic development?*

This research question has been addressed in Chapter 7. Since current industry practice relies on hand-written manual checkers and data post-processing, we proposed to monitor formally defined properties of the DUT during runtime. This, on one hand, removes ambiguities from the specifications, and, on the other hand, eliminates costly post-processing step. We showed how runtime monitoring can be applied both during chip concept design and post-silicon verification phases. During chip concept design, monitors are attached to the SystemC model of the sensor prototype and provide results during simulation runtime; during post-silicon verification the monitors are synthesized in FPGA hardware and attached to the engineering sample. This enables to perform long-term tests, and also to reuse the monitors between verification phases.

RQ 4: *What are the necessary steps to build efficient runtime monitors in hardware that are applicable in the industrial state-of-the-art practice?*

This research question has been addressed in Chapter 8, where we showed how to runtime monitor the SENT and SPC protocols and apply results in industrial lab environment. We started with electrical and timing requirements of the protocols, formalized these in STL and perform necessary transformation steps to obtain formula in a way that can be efficiently mapped to FPGA. We then used pre-recorded sensor data to test the monitors and include this data in a test bench to generate runtime monitor using HLS. After this step we obtained the HDL code that is included in the hardware which models transmission medium between sensor and an ECU. For the results to be applicable in the industrial context, we addressed "inhibitors of adopting the tools": (i) we embedded monitor generation in the tool chain that is used at Infineon (see Appendix 1), connected monitor generation with the design process of the electronic sensor product, and demonstrated industrial-strength evidence on the SENT and SPC protocols.

9.4 Future Work

In the thesis we have presented approaches for control, and runtime monitoring of CPS. We also showed how runtime monitoring can be applied in automotive electronic development for checking conformance of requirements of industrial protocols. The presented results uncover a range of research directions can be addressed in future work:

- Based on Chapter 4 and [CFFW16] where the authors applied neural models to construct an LQR controller for a flapping robot, we see as an extension of this work to develop a neural model predictive controller for a mobile robot;
- We are interested to extend the work in Chapters 5 and 6 to develop computational procedure for evaluating other quantitative semantics of STL and how different spike encoding strategies (e.g. "time to the first spike", "phase" [GK02]) can be applied to facilitate the computation;
- In the context of Chapters 7 and 8, and over the course of development the industrial case study, we have identified difficulties in in converting natural language requirements for automotive sensor products to their formal representation. The future work needs to facilitate this conversion, with the possible improvements on specification mining and identifying relevant patterns in the specification. Another direction for future work is related to works of [SKK$^+$13] where a user graphically specifies a property and it is then converted to a temporal logic formula;
- Based on the research on applying runtime verification in automotive electronic development, a project proposal has been submitted to continue this work and develop methods to bridge the gap between natural language and formal requirements.

Glossary

C. Elegans – *Caenorhabditis elegans*, a nematode, whose adult individual has exactly 302 neural cells. The absence of variability in the neural structure, relatively low number of cells made *C. elegans* a perfect model organism in gerontology, developmental, evolutional and neuro- biology, and, recently, in formal verification [IDFF$^+$15]. xi, 21, 22, 32, 45, 46

reactive system – According to “Reactive Systems: Modelling, Specification and Verification” by Luca Aceto, Anna Ingolfsdottir, Kim G. Larsen, and Jirí Srba, reactive system is an inherently non-terminating system, which behavoir cannot be described with a function from inputs to outputs. 2

sciences of the artificial – Herbert A. Simon in “The Sciences of the Artificial” opposes “natural sciences” which “are concerned with how things are” with the “design sciences” which are “concerned with how things ought to be, with devising artifacts to attain goal”. 2

Tier 1 – Hardware and Software components providers, which provide their outcomes directly to the OEM. 26

Tier 2 – Hardware and Software components providers, which do not provide their outcomes directly to the OEM, but to the Tier 1 suppliers, who in turn integrate the components from Tier 2 and provide systems to the OEM. 5, 26

Acronyms

ADC Analog-to-digital converter. 79, 84, 96, 97

AFE Analog Front-End. 80, 97

AI Artificial Intelligence. 21, 31

ASIL Automotive Safety Integrity Level. 26

AXI Advanced eXtensible Interface. 25

BN Bayesian network. 33, 34

COTS commercial off-the-shelf. 63

CPD conditional probability distribution. 33, 39

CPS Cyber-Physical Systems. 1–8, 11, 14, 16, 21, 22, 28, 31, 32, 34, 37, 38, 41, 43, 49, 51, 54, 62, 101, 103, 104

DSI3 3rd generation Distributed System Interface. 28

DSL Domain Specific Language. 27

DUT Design Under Test. 26, 78, 103

EBS electronic braking system. 83

ECU Electronic Control Unit. 26, 79, 80, 86, 97, 98, 104

EPS electronic power steering. 83

FPGA Field-Programmable Gate Array. 14, 25–27, 51–55, 61, 62, 75, 79, 80, 83, 84, 95, 97–103

GBN Gaussian Bayesian network. 21, 22, 32–34, 38–41, 43, 44, 51

GD Gaussian distribution. 33, 35, 36, 38

GS Gaussian smoothing. 22

GUI graphical user interface. 83

HDL Hardware Description Language. 27, 51, 54, 96, 98, 101, 102, 104

HLS High-Level Synthesis. 27, 54, 60, 80, 96, 98–102, 104

HMM Hidden Markov Model. 23, 25

ILP Integer Linear Program. 56

IMU inertial measurement unit. 44

IP intellectual property. 55

LIF leaky-integrate-and-fire. 24, 52

LTL Linear Temporal Logic. 16, 23, 27, 57

MGD multivariate Gaussian distribution. 33, 39

MTL Metric Temporal Logic. 16, 23, 25, 27, 51–55, 57, 60–64, 74

NeuralP Neural Programming Constructs. 32, 34, 44, 46, 49

ODE ordinary differential equation. 12, 13, 46, 64

OEM Original Equipment Manufacturer. 18, 26

PD probability density. 35

PDF Probability Density Function. 22, 33–35

PSI5 Peripheral Sensor Interface 5. 79

PSL Property Specification Language. 23, 26–28

ROS Robot Operating System. 21, 42, 44

RS requirements specification. 4, 5

RTL Register-Transfer Level. 27, 54, 79

RV random variable. 33–36, 38

SENT Single Edge Nibble Transmission. 7, 8, 16, 21, 27, 83–88, 90, 92, 95, 97–99, 102–104, 131, 150

SPC Short PWM Code. 7, 8, 16, 21, 27, 83–88, 90, 91, 95, 98, 99, 102–104, 131, 150

STL Signal Temporal Logic. 17, 26, 28, 52, 75, 77–79, 83, 84, 88–91, 94, 96–100, 102–104

TRE Timed Regular Expressions. 26, 27, 84, 88, 91–93, 96–100

UAV Unmanned Aerial Vehicle. 27

UGD univariate Gaussian distribution. 33

UKF unscented Kalman filter. 44

UT unit time. 89, 90, 96

V & V Verification and Validation. 18, 19

Bibliography

[AC16] Bogdan Aman and Gabriel Ciobanu. Modelling and verification of weighted spiking neural systems. *Theoretical Computer Science*, 623:92 – 102, 2016.

[ACM02] Eugene Asarin, Paul Caspi, and Oded Maler. Timed regular expressions. *J. ACM*, 49(2):172–206, 2002.

[ACN10] Pieter Abbeel, Adam Coates, and Andrew Y. Ng. Autonomous helicopter aerobatics through apprenticeship learning. *Int. J. Rob. Res.*, 29(13):1608–1639, 2010.

[Ade] MobileRobots Pioneer 3-AT (P3AT) research robot platform. http://www.mobilerobots.com/ResearchRobots/P3AT.aspx *(Accessed 19.04.2017)*.

[Adh13] Pooja Adhikari. *A Domain Specific Language Based Approach for Generating Deadlock-free Parallel Load Scheduling Protocols for Distributed Systems*. PhD thesis, Mississippi State, MS, USA, 2013. AAI3558892.

[ADMN12] Eugene Asarin, Alexandre Donzé, Oded Maler, and Dejan Nickovic. *Parametric Identification of Temporal Properties*, pages 147–160. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[ADR+13] Arnon Amir, Pallab Datta, William P. Risk, Andrew S. Cassidy, Jeffrey A. Kusnitz, Steve K. Esser, Er Andreopoulos, Theodore M. Wong, Myron Flickner, Rodrigo Alvarez-icaza, Emmett Mcquinn, Ben Shaw, Norm Pass, and Dharmendra S. Modha. Cognitive Computing Programming Paradigm: A Corelet Language for Composing Networks of Neurosynaptic Cores. In *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2013.

[AG16] Infineon Technologies AG. *Sensing the world: Sensor solutions for automotive, industrial and consumer applications*. Infineon Technologies AG, 2016.

[AH90] R. Alur and T.A. Henzinger. Real-time logics: complexity and expressiveness. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 390–401, Jun 1990.

[AILS07] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification.* Cambridge University Press, New York, NY, USA, 2007.

[ALFS11] Y. S. R. Annapureddy, C. Liu, G. E. Fainekos, and S. Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *Proc. of TACAS 2011: the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *LNCS*, pages 254–257. Springer, 2011.

[AMGC02] M.S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp. A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking. *IEEE Transactions on Signal Processing*, 50(2):174–188, 2002.

[Amp10] Nicholas Ampazis. *Large Scale Problem Solving with Neural Networks: The Netflix Prize Case*, pages 429–434. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[AP93] Panos J. Antsaklis and Kevin M. Passino, editors. *An Introduction to Intelligent and Autonomous Control.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[AW94] Karl Johan Astrom and Bjorn Wittenmark. *Adaptive Control.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994.

[Axe07] Jan Axelson. *Serial Port Complete: COM Ports, USB Virtual COM Ports, and Ports for Embedded Systems; 2nd ed.* Lakeview Research, Madison, WI, 2007.

[BBHM09] S. Bhatti, E. Brady, K. Hammond, and J. McKinna. Domain specific languages (dsls) for network protocols (position paper). In *2009 29th IEEE International Conference on Distributed Computing Systems Workshops*, pages 208–213, June 2009.

[BBN13] Ezio Bartocci, Luca Bortolussi, and Laura Nenzi. A temporal logic approach to modular design of synthetic biological circuits. In *Proc. of CMSB 2013: the 11th International Conference on Computational Methods in Systems Biology*, volume 8130 of *LNCS*, pages 164–177. Springer, 2013.

[BBNS15] Ezio Bartocci, Luca Bortolussi, Laura Nenzi, and Guido Sanguinetti. System design of stochastic models using robustness of temporal properties. *Theor. Comput. Sci.*, 587:3–25, 2015.

[BD13] Alan Burns and Robert Davis. *Mixed Criticality Systems - A Review.* University of York, Tech. Rep, 2013.

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[BDL$^+$11] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing UPPAAL over 15 years. *Softw., Pract. Exper.*, 41(2):133–142, 2011.

[Ben02] Janine M. Benyus. *Biomimicry: Innovation Inspired by Nature.* Harper Perennial, New York, USA, 2002.

[BG12] M. Broy and E. Geisberger. Cyber-physical systems, driving force for innovation in mobility, health, energy and production. *Acatech: The National Academy Of Science and Engineering*, 2012.

[BGK$^+$02] Johan Bengtsson, W.O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Automated verification of an audio-control protocol using uppaal. *The Journal of Logic and Algebraic Programming*, 52:163 – 181, 2002.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.

[BKKS12] Manfred Broy, Helmut Krcmar, Sascha Kirstan, and Bernhard Schätz. What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry? In *Emerging Technologies for the Evolution and Maintenance of Software Models*, pages 310–337, 2012.

[BLMA$^+$05] 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.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011.

[Bre15] Romain Brette. What is the most realistic single-compartment model of spike initiation? *PLOS Computational Biology*, 11(4):1–13, 04 2015.

[BZ06] M. Boule and Z. Zilic. Efficient automata-based assertion-checker synthesis of psl properties. In *2006 IEEE International High Level Design Validation and Test Workshop*, pages 69–76, 2006.

[CES13] K. Claessen, N. Een, and B. Sterin. A circuit approach to LTL model checking. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 53–60, Oct 2013.

[CFFW16] Taylor S. Clawson, Silvia Ferrari, Sawyer B. Fuller, and Robert J. Wood. Spiking neural network (SNN) control of a flapping insect-scale robot. In *55th IEEE Conference on Decision and Control, CDC 2016, Las Vegas, NV, USA, December 12-14, 2016*, pages 3381–3388, 2016.

[CLC$^+$03] Jose M Carmena, Mikhail A. Lebedev, Roy E. Crist, Joseph E. O'Doherty, David M. Santucci, Dragan F. Dimitrov, Parag G. Patil, Craig S. Henriquez, and Miguel A.L. Nicolelis. Learning to control a brain–machine interface for reaching and grasping by primates. *PLOS Biology*, 1(2), 10 2003.

[CMA$^+$13] Andrew S. Cassidy, Paul Merolla, John V. Arthur, Steve K. Esser, Bryan Jackson, Rodrigo Alvarez-icaza, Pallab Datta, Jun Sawada, Theodore M. Wong, Vitaly Feldman, Arnon Amir, Daniel Ben dayan Rubin, Emmett Mcquinn, William P. Risk, and Dharmendra S. Modha. Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores. In *in International Joint Conference on Neural Networks (IJCNN). IEEE*, 2013.

[CMG17] Giovanni Ciatto, Elisabetta De Maria, and Cinzia Di Giusto. Modeling third generation neural networks as timed automata and verifying their behavior through temporal logic. In *[Research Report] Universit'e Cote d'Azur*, pages 1 – 68, CNRS, I3S, France, 2017. <hal-01473941>.

[CMS12] D. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649, 2012.

[CRL99] S. Chandra, B. Richards, and J. R. Larus. Teapot: a domain-specific language for writing cache coherence protocols. *IEEE Transactions on Software Engineering*, 25(3):317–333, May 1999.

[CRST09] 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.

[CSL] Swarat Chaudhuri and Armando Solar-Lezama. Smooth Interpretation: Presentation Slides. http://people.csail.mit.edu/asolar/Talks/PLDI2010Final.pptx *(Accessed 14.03.2015).*

[CSL10] Swarat Chaudhuri and Armando Solar-Lezama. Smooth interpretation. In *PLDI*, pages 279–291, 2010.

[CSL11] Swarat Chaudhuri and Armando Solar-Lezama. Smoothing a program soundly and robustly. In *CAV*, pages 277–292, 2011.

[DA05] Peter Dayan and L. F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems.* The MIT Press, 2005.

[DBDRC+15] Zidong Du, Daniel D. Ben-Dayan Rubin, Yunji Chen, Liqiang He, Tianshi Chen, Lei Zhang, Chengyong Wu, and Olivier Temam. Neuromorphic accelerators: A comparison between neuroscience and machine-learning approaches. In *In Proc. of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 494–507, New York, NY, USA, 2015. ACM.

[DGG+05a] A. Dahan, D. Geist, L. Gluhovsky, D. Pidan, G. Shapir, Y. Wolfsthal, L. Benalycherif, R. Kamidem, and Y. Lahbib. Combining system level modeling with assertion based verification. In *Quality of Electronic Design, 2005. ISQED 2005. Sixth International Symposium on*, pages 310–315, March 2005.

[DGG+05b] Anat Dahan, Daniel Geist, Leonid Gluhovsky, Dmitry Pidan, Gil Shapir, Yaron Wolfsthal, Lyes Benalycherif, Romain Kamdem, and Younes Lahbib. Combining system level modeling with assertion based verification. In *6th International Symposium on Quality of Electronic Design (ISQED) 21-23 March 2005, San Jose, CA, USA*, pages 310–315, 2005.

[Dij70] Edsger W. Dijkstra. *Notes On Structured Programming*, pages 0–88. Technological University Eindhoven, The Netherlands, Department of Mathematics, T.H.-Report (EWD249), 70-WSK-03, 1970.

[DMB+12] Alexandre Donze, Oded Maler, Ezio Bartocci, Dejan Nickovic, Radu Grosu, and Scott Smolka. On Temporal Logic and Signal Processing. In *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, pages 92–106. Springer Berlin Heidelberg, 2012.

[DMSS11] Guillaume Drion, Laurent Massotte, Rodolphe Sepulchre, and Vincent Seutin. How modeling can reconcile apparently discrepant experimental results: The case of pacemaking in dopaminergic neurons. *PLoS Computational Biology*, 7(5), 2011.

[Doi07] Norman Doidge. *The brain that changes itself: stories of personal triumph from the frontiers of brain science.* Viking, New York, USA, 2007.

[Don10] Alexandre Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *Proc. of CAV 2010: the 22nd International Conference on Computer Aided Verification*, volume 6174 of *LNCS*, pages 167–170. Springer Berlin, 2010.

[DP06] Yang Dan and Mu-Ming Poo. Spike timing-dependent plasticity: From synapse to perception. *Physiological Reviews*, 86(3):1033–1048, 2006.

[DPM$^+$11] Alex Dranovsky, Alyssa M. Picchini, Tiffany Moadel, Alexander C. Sisti, Atsushi Yamada, Shioko Kimura, E. David Leonardo, and Rene Hen. Experience dictates stem cell fate in the adult hippocampus. *Neuron*, 70(5):908 – 923, 2011.

[DS13] Susanne Ditlevsen and Adeline Samson. *Introduction to stochastic models in biology*, pages 3–34. Lecture Notes in Mathematics. Springer, 2013. 2013; 1.

[DSO13] Krishnaji Desai, Kenneth S. Stevens, and John O'Leary. Symbolic verification of timed asynchronous hardware protocols. In *IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2013, Natal, Brazil, August 5-7, 2013*, pages 147–152, 2013.

[DWPM11] Utsav Drolia, Zhenyan Wang, Yash Pant, and Rahul Mangharam. Autoplug: An automotive test-bed for electronic controller unit testing and verification. In *14th International IEEE Conference on Intelligent Transportation Systems, ITSC 2011, Washington, DC, USA, October 5-7, 2011*, pages 1187–1192, 2011.

[DY14] Li Deng and Dong Yu. Deep learning: Methods and applications. *Found. Trends Signal Process.*, 7(3&#8211;4):197–387, 2014.

[EAA$^+$13] Steven K. Esser, Alexander Andreopoulos, Rathinakumar Appuswamy, Pallab Datta, Davis Barch, Arnon Amir, John V. Arthur, Andrew Cassidy, Myron Flickner, Paul Merolla, Shyamal Chandra, Nicola Basilico, Stefano Carpin, Tom Zimmerman, Frank Zee, Rodrigo Alvarez-Icaza, Jeffrey A. Kusnitz, Theodore M. Wong, William P. Risk, Emmett McQuinn, Tapan K. Nayak, Raghavendra Singh, and Dharmendra S. Modha. Cognitive computing systems: Algorithms and applications for networks of neurosynaptic cores. In *IJCNN*, pages 1–10. IEEE, 2013.

[EBC$^+$10] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *J. Mach. Learn. Res.*, 11:625–660, March 2010.

[Eis07] Cindy Eisner. PSL for Runtime Verification: Theory and Practice. In *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*, pages 1–8, 2007.

[FMFR11] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. Runtime enforcement monitors: composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3):223–262, 2011.

[FMNU15] Thomas Ferrère, Oded Maler, Dejan Ničković, and Dogan Ulus. *Measuring with Timed Patterns*, pages 322–337. Springer International Publishing, Cham, 2015.

[FSUY12] Georgios E. Fainekos, Sriram Sankaranarayanan, Koichi Ueda, and Hakan Yazarel. Verification of automotive control applications using S-TaLiRo. In *American Control Conference, ACC 2012, Montreal, QC, Canada*, pages 3567–3572, 2012.

[FZ12] Yliès Falcone and Lenore D. Zuck. *Runtime Verification: The Application Perspective*, pages 284–291. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[GBL04] EL Graas, EA Brown, and Robert H Lee. An fpga-based approach to high-speed simulation of conductance-based neuron models. *Neuroinformatics*, 2(4):417–435, 2004.

[GD07] Marc-Oliver Gewaltig and Markus Diesmann. Nest (neural simulation tool). *Scholarpedia*, 2(4):1430, 2007.

[GEB15] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A neural algorithm of artistic style. *CoRR*, abs/1508.06576, 2015.

[GHNR14] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *International Conference on Software Engineering (ICSE Future of Software Engineering)*. IEEE, May 2014.

[GK02] Wulfram Gerstner and Werner M. Kistler. *Spiking Neuron Models: Single Neurons, Populations, Plasticity.* Cambridge University Press, 2002.

[GKNP14] Wulfram Gerstner, Werner M. Kistler, Richard Naud, and Liam Paninski. *Neuronal Dynamics: From Single Neurons to Networks and Models of Cognition.* Cambridge University Press, New York, NY, USA, 2014.

[GrMH13] A. Graves, A. r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, May 2013.

[GRS14] Johannes Geist, Kristin Y. Rozier, and Johann Schumann. Runtime Observer Pairs and Bayesian Network Reasoners On-board FPGAs: Flight-Certifiable System Health Management for Embedded Systems. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Proc. of Runtime Verification: 5th International Conference, RV 2014*, pages 215–230. Springer International Publishing, 2014.

[GS85] G. Grimmett and D. Stirzaker. *Probability and random processes.* Oxford science publications. Clarendon Press, 1985.

[GS12] Sebastian Große and Wolfgang Schröder. *Deflection-based flow field sensors — examples and requirements*, pages 393–403. Springer Vienna, Vienna, 2012.

[GT11] Changdong Gu and Jiangping Tu. One-step fabrication of nanostructured ni film with lotus effect from deep eutectic solvent. *Langmuir*, 27(16):10132–10140, 2011.

[HG95] David Heckerman and Dan Geiger. Learning bayesian networks: A unification for discrete and gaussian domains. In *UAI*, pages 274–284, 1995.

[HH52] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500–544, 1952.

[Hin93] Michael Hines. Neuron a program for simulation of nerve equations. *Neural Systems: Analysis and Modeling*, pages 127–136, 1993.

[HMF14] Donal Heffernan, Ciaran MacNamee, and Padraig Fogarty. Runtime verification monitoring for automotive embedded systems using the ISO 26262 functional safety standard as a guide for the definition of the monitored properties. *IET Software*, 8(5):193–203, 2014.

[Hof80] Miguel García Hoffmann. Hardware implementation of communication protocols: A formal approach. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, ISCA '80, pages 253–263, La Baule, USA, 1980. ACM.

[HQB$^+$09] Xue Han, Xiaofeng Qian, Jacob G. Bernstein, Hui hui Zhou, Giovanni Talei Franzesi, Patrick Stern, Roderick T. Bronson, Ann M. Graybiel, Robert Desimone, and Edward S. Boyden. Millisecond-timescale optical control of neural dynamics in the nonhuman primate brain. *Neuron*, 62(2):191 – 198, 2009.

[IDFF+15] Md. Ariful Islam, Richard De Francisco, Chuchu Fan, Radu Grosu, Sayan Mitra, and Scott A. Smolka. *Model Checking Tap Withdrawal in C. Elegans*, pages 195–210. Springer International Publishing, Cham, 2015.

[ILBH+11] Giacomo Indiveri, Bernabé Linares-Barranco, Tara Julia Hamilton, André Van Schaik, Ralph Etienne-Cummings, Tobi Delbruck, Shih-Chii Liu, Piotr Dudek, Philipp Hafliger, Sylvie Renaud, et al. Neuromorphic silicon neuron circuits. *Frontiers in neuroscience*, 5:73, 2011.

[IMDACPR+12] M.-A Ibarra-Manzano, J.-H. De-Anda-Cuellar, C.-A Perez-Ramirez, O.-I Vera-Almanza, F.-J. Mendoza-Galindo, M.-A Carbajal-Guillen, and D.-L. Almanza-Ojeda. Intelligent algorithm for parallel self-parking assist of a mobile robot. In *Electronics, Robotics and Automotive Mechanics Conference (CERMA), 2012 IEEE Ninth*, pages 37–41, Nov 2012.

[Inc] Xilinx Inc. Vivado High-Level Synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html *(Accessed 25.09.2016)*.

[Inc16] Xilinx Inc. Vivado Design Suite Tutorial, Programming and Debugging. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_2/ug936-vivado-tutorial-programming-debugging.pdf`, 2016. [Online; Accessed 12-October-2016].

[Inf17] Semiconductor and System Solutions - Infineon Technologies, 2017. http://www.infineon.com/ *(Accessed 14.03.2017)*.

[Int16] SAE International. SENT - Single Edge Nibble Transmission for Automotive Applications, J2716, Standard. `http://standards.sae.org/j2716_201001/`, 2016. [Online; Accessed 3-October-2016].

[iso11] *ISO 26262: "Road vehicles – Functional safety"*. International Organization for Standardization (ISO), 2011.

[Izh03] Eugene M. Izhikevich. Simple model of spiking neurons. *IEEE Trans. Neural Networks*, pages 1569–1572, 2003.

[Izh04] Eugene M. Izhikevich. Which model to use for cortical spiking neurons. *IEEE Transactions on Neural Networks*, 15:1063–1070, 2004.

[JBG+15] Stefan Jaksic, Ezio Bartocci, Radu Grosu, Reinhard Kloibhofer, Thang Nguyen, and Dejan Nickovic. From Signal Temporal Logic to FPGA Monitors. In *Proc. of 13. ACM/IEEE International Conference on Formal Methods and Models for Codesign*, pages 218–227, 2015.

[JKN10] Kevin D. Jones, Victor Konrad, and Dejan Nickovic. Analog property checkers: a DDR2 case study. *Formal Methods in System Design*, 36(2):114–130, 2010.

[JMB17] Fatemeh Negin Javaheri, Katell Morin-Allory, and Dominique Borrione. Synthesis of regular expressions revisited: From PSL seres to hardware. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 36(5):869–882, 2017.

[JRCK10] Hongbo Jia, Nathalie L. Rochefort, Xiaowei Chen, and Arthur Konnerth. Dendritic organization of sensory input to cortical neurons in vivo. *Nature*, 464(7293):1307–1312, 4 2010.

[JS99] K. Jiang and L.D. Seneviratne. A sensor guided autonomous parking system for nonholonomic mobile robots. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 1, pages 311–316 vol.1, 1999.

[KBS$^+$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.

[KD05] M. Khoshnejad and K. Demirli. Autonomous parallel parking of a car-like mobile robot by a neuro-fuzzy behavior-based controller. In *Fuzzy Information Processing Society, 2005. NAFIPS 2005. Annual Meeting of the North American*, pages 814–819, June 2005.

[KF09] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning.* The MIT Press, 2009.

[KF12] Daniel Kästner and Christian Ferdinand. Static verification of non-functional software requirements in the iso-26262. In *Automotive - Safety and Security*, pages 39–53, 2012.

[KFL15] Ali Kassem, Yliès Falcone, and Pascal Lafourcade. Monitoring electronic exams. In *Runtime Verification - 6th International Conference, RV 2015 Vienna, Austria, September 22-25, 2015. Proceedings*, pages 118–135, 2015.

[KK17] Benjamin Lucien Kaminski and Joost-Pieter Katoen. A weakest pre-expectation semantics for mixed-sign expectations. *Collected Abstracts of the 2nd Workshop on Probabilistic Programming Semantics (PPS 2017)*, January 2017.

[KKMO16] Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. Weakest precondition reasoning for expected run-times of probabilistic programs. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 364–389, 2016.

[Kop11] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Springer Publishing Company, Incorporated, 2nd edition, 2011.

[KS14] Marko Kolbe and Jonathan Schoo. *Industry Overview: The Automotive Electronics Industry in Germany.* Germany Trade and Invest, 2014.

[LBD+90] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, R. E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Handwritten digit recognition with a back-propagation network. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 396–404. Morgan-Kaufmann, 1990.

[Leu12] Martin Leucker. Teaching Runtime Verification. In Sarfraz Khurshid and Koushik Sen, editors, *Proc. of Runtime Verification: Second International Conference, RV 2011*, pages 34–48. Springer Berlin Heidelberg, 2012.

[Lli99] Rodolfo R. Llinas. *The squid giant synapse : a model for chemical transmission.* Oxford University Press, New York, 1999.

[LMBA06] Francisco López-Muñoz, Jesús Boya, and Cecilio Alamo. Neuron theory, the cornerstone of neuroscience, on the centenary of the Nobel Prize award to Santiago Ramón y Cajal . *Brain Research Bulletin*, 70(4–6):391 – 405, 2006.

[Lon89] Derek Long. A review of temporal logics. *The Knowledge Engineering Review*, 4(2):141–162, 1989.

[Lyo11] Richard G. Lyons. *Understanding Digital Signal Processing 3rd Edition.* Prentice Hall, Upper Saddle River, NJ, USA, 3rd edition, 2011.

[LYW+10] T.S. Li, Ying-Chieh Yeh, Jyun-Da Wu, Ming-Ying Hsiao, and Chih-Yang Chen. Multifunctional Intelligent Autonomous Parking Controllers for Carlike Mobile Robots. *Industrial Electronics, IEEE Transactions on*, 57(5):1687–1700, May 2010.

[Mal16] Oded Maler. *Some Thoughts on Runtime Verification*, pages 3–14. Springer International Publishing, 2016.

[MH15] Ramin M. Hasani. Design of cmos silicon neurons for noise assisted computation in spiking neural networks. *Politesi Digital Library of PhD and Post Graduate Theses, Politecnico di Milano*, 2015.

[Mit07] H.B. Mitchell. *Multi-Sensor Data Fusion - An Introduction.* Springer, Berlin, Heidelberg, New York, 2007.

[MN13a] Oded Maler and Dejan Nickovic. Monitoring properties of analog and mixed-signal circuits. *International Journal on Software Tools for Technology Transfer*, 15(3):247–268, 2013.

[MN13b] Oded Maler and Dejan Ničković. Monitoring properties of analog and mixed-signal circuits. *International Journal on Software Tools for Technology Transfer*, 15(3):247–268, 2013.

[MNP06] Oded Maler, Dejan Nickovic, and Amir Pnueli. From MITL to Timed Automata. In Eugene Asarin and Patricia Bouyer, editors, *Proc. of Formal Modeling and Analysis of Timed Systems*, volume 4202 of *Lecture Notes in Computer Science*, pages 274–289. Springer Berlin Heidelberg, 2006.

[MNP07] Oded Maler, Dejan Nickovic, and Amir Pnueli. On Synthesizing Controllers from Bounded-Response Properties. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 95–107. Springer Berlin Heidelberg, 2007.

[MNP08] Oded Maler, Dejan Nickovic, and Amir Pnueli. Checking Temporal Properties of Discrete, Timed and Continuous Behaviors. In *Pillars of Computer Science*, volume 4800 of *Lecture Notes in Computer Science*, pages 475–505. Springer Berlin Heidelberg, 2008.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification.* Springer, 1992.

[MSH$^+$13] L. Märtin, M. Schatalov, M. Hagner, U. Goltz, and O. Maibaum. A methodology for model-based development and automated verification of software for aerospace systems. In *2013 IEEE Aerospace Conference*, pages 1–19, March 2013.

[NBHT14] T. Nguyen, A. Basa, D. Hammerschmidt, and Dittfeld T. Advanced Mixed-Signal Emulation for Complex Automotive ICs. In *AIRBAG Conference*, pages 1–8, 2014.

[NBN$^+$16] Thang Nguyen, Ezio Bartocci, Dejan Nickovic, Radu Grosu, Stefan Jaksic, and Konstantin Selyunin. The HARMONIA project: Hardware monitoring for automotive systems-of-systems. In *Proc. of Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Corfu, Greece, October 10-14*, pages 371–379, 2016.

[Nea03] Richard E. Neapolitan. *Learning Bayesian Networks.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

[neu] Parking Videos. http://youtu.be/xNOj_ARSEYs?list=PLP5Gx6r7gK2cxjKv0K2V5fBedovfo8_3y *(Accessed 12.03.2015).*

[NM07] Dejan Nickovic and Oded Maler. Amt: A property-based monitoring tool for analog systems. In Jean-François Raskin and P.S. Thiagarajan, editors, *Formal Modeling and Analysis of Timed Systems*, volume 4763 of *Lecture Notes in Computer Science*, pages 304–319. Springer Berlin Heidelberg, 2007.

[NN14] Thang Nguyen and Dejan Nickovic. Assertion-Based Monitoring in Practice – Checking Correctness of an Automotive Sensor Interface. In *Proc. of Formal Methods for Industrial Critical Systems*, pages 16–32. Springer, 2014.

[NN16] Thang Nguyen and Dejan Nickovic. Assertion-based monitoring in practice - checking correctness of an automotive sensor interface. *Sci. Comput. Program.*, 118:40–59, 2016.

[NP10] Dejan Nickovic and Nir Piterman. From MTL to Deterministic Timed Automata. In Krishnendu Chatterjee and Thomas A. Henzinger, editors, *Proc. of Formal Modeling and Analysis of Timed Systems*, volume 6246 of *Lecture Notes in Computer Science*, pages 152–167. Springer Berlin Heidelberg, 2010.

[NW14] Thang Nguyen and Stuart N. Wooters. FPGA-Based Development for Sophisticated Automotive Embedded Safety Critical System. In *SAE International Journal of Passenger Cars - Electronic and Electrical Systems*, pages 125–132, 2014.

[NW15] Werner Nachtigall and Alfred Wisser. *Bionics by Examples.* Springer International Publishing, 2015.

[OKT14] R. Okuda, Y. Kajiwara, and K. Terashima. A survey of technical trend of adas and autonomous driving. In *Proc. of International Symposium on VLSI Design, Automation and Test (VLSI-DAT) 2014*, pages 1–4, April 2014.

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

[OW08] Joël Ouaknine and James Worrell. Some Recent Results in Metric Temporal Logic. In *Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems*, FORMATS '08, pages 1–13. Springer-Verlag, 2008.

[Par85] David Lorge Parnas. Software aspects of strategic defense systems. *Commun. ACM*, 28(12):1326–1335, December 1985.

[Per14] Jeffrey M. Perkel. Mapping Neural Connections. *BioTechniques*, 57(5):230–236, 2014.

[PFJ$^+$13] Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Landry Nguena Timo. *Runtime Enforcement of Timed Properties*, pages 229–244. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[PGZ$^+$14] Can Pan, Jian Guo, Longfei Zhu, Jianqi Shi, Huibiao Zhu, and Xinyun Zhou. Modeling and verification of can bus with application layer using uppaal. *Electronic Notes in Theoretical Computer Science*, 309:31 – 49, 2014.

[PWD$^+$12] Robert Preissl, Theodore M. Wong, Pallab Datta, Myron Flickner, Raghavendra Singh, Steven K. Esser, William P. Risk, Horst D. Simon, and Dharmendra S. Modha. Compass: A scalable simulator for an architecture for cognitive computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 54:1–54:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[PY97] Kevin M. Passino and Stephen Yurkovich. *Fuzzy Control.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[PZ08a] A. Pnueli and A. Zaks. On the Merits of Temporal Testers. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 172–195. Springer Berlin Heidelberg, 2008.

[PZ08b] Amir Pnueli and Aleksandr Zaks. On the merits of temporal testers. In *25 Years of Model Checking - History, Achievements, Perspectives*, pages 172–195, 2008.

[QCG+09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully B. Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[R2008] ANSI E1.11-2008 R2013. Entertainment Technology – USITT DMX512-A – Asynchronous Serial Digital Data Transmission Standard for Controlling Lighting Equipment and Accessories . `http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+E1.11-2008+(R2013)`, 2008. [Online; Accessed 2-October-2016].

[RBNG16] Alena Rodionova, Ezio Bartocci, Dejan Nickovic, and Radu Grosu. Temporal Logic as Filtering. In *19th ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2016, Vienna, Austria, April 12-14, 2016, Proceedings*, pages 11–20, 2016.

[RBNP08] Marc Raibert, Kevin Blankespoor, Gabriel Nelson, and Rob Playter. Bigdog, the rough-terrain quadruped robot. *IFAC Proceedings Volumes*, 41(2):10822 – 10825, 2008. 17th IFAC World Congress.

[RFB12] Thomas Reinbacher, Matthias Függer, and Jörg Brauer. Real-time runtime verification on chip. In *Runtime Verification, Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, Revised Selected Papers*, pages 110–125, 2012.

[RFB14] Thomas Reinbacher, Matthias Függer, and Jörg Brauer. Runtime verification of embedded real-time systems. *Formal Methods in System Design*, 44(3):203–239, 2014.

[RN10] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, 3rd edition, 2010.

[RNPH05] G. Rodriguez-Navas, J. Proenza, and H. Hansson. Using uppaal to model and verify a clock synchronization protocol for the controller area network. In *2005 IEEE Conference on Emerging Technologies and Factory Automation*, volume 2, pages 8 pp.–502, Sept 2005.

[Ros32] Charles Rosen. The Origin of the Conception of the Nervous Impulse. *Canadian Medical Association Journal*, 27(1):66 – 70, 1932.

[RRS14] Thomas Reinbacher, Kristin Yvonne Rozier, and Johann Schumann. Temporal-logic based runtime observer pairs for system health management of real-time systems. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems - 20th Int. Conf., (TACAS), Grenoble, France*, pages 357–372, 2014.

[RSB+14] Rakesh Rana, Miroslaw Staron, Christian Berger, Jörgen Hansson, Martin Nilsson, and Fredrik Törner. *Early Verification and Validation According to ISO 26262 by Combining Fault Injection and Mutation Testing*, pages 164–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[RvR09] Arnd Roth and Mark CW van Rossum. Modeling synapses. *Computational modeling methods for neuroscientists*, 6:139–160, 2009.

[San13] M. Sans. X-by-wire park assistance for electric city cars. In *Proc. of World Electric Vehicle Symposium and Exhibition (EVS27), 2013*, pages 1–9, Nov 2013.

[SBS+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.

[SGC+12] N. Scicluna, E. Gatt, O. Casha, I Grech, and J. Micallef. Fpga-based autonomous parking of a car-like robot using fuzzy logic control. In *Electronics, Circuits and Systems (ICECS), 2012 19th IEEE International Conference on*, pages 229–232, Dec 2012.

[She15] Gordon M Shepherd. *Foundations of the Neuron Doctrine.* Oxford University Press, New York, NY, 25th Anniversary edition, 2015.

[SHR+17] Konstantin Selyunin, Ramin M. Hasani, Denise Ratasich, Ezio Bartocci, and Radu Grosu. Computing with biophysical and hardware-efficient neural models. In *Submitted to the Advances in Computational Intelligence - 14th International Work-Conference on Artificial Neural Networks, IWANN 2017, Cadiz, Spain, June 14-16*, pages 1–12, 2017.

[Sim96] Herbert A. Simon. *The Sciences of the Artificial (3rd Ed.).* MIT Press, Cambridge, MA, USA, 1996.

[SJN+17] Konstantin Selyunin, Stefan Jaksic, Thang Nguyen, Christian Reidl, Udo Hafner, Ezio Bartocci, Dejan Nickovic, and Radu Grosu. Runtime monitoring with recovery of the sent communication protocol. In *Submitted, Under Review*, pages 1–17, 2017.

[SKK+13] S. Srinivas, R. Kermani, K. Kim, Y. Kobayashi, and G. Fainekos. A graphical language for ltl motion and mission planning. In *2013 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 704–709, Dec 2013.

[SLB+08] Sergio Saponara, Nicola E. L'Insalata, Tony Bacchillone, Esa Petri, Iacopo Del Corona, and Luca Fanucci. Hardware/software fpga-based network emulator for high-speed on-board communications. In *11th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, DSD 2008, Parma, Italy, September 3-5, 2008*, pages 353–359, 2008.

[SMN11] Rodrigo Martins da Silva, Luiza de Macedo Mourelle, and Nadia Nedjah. Compact yet efficient hardware architecture for multilayer-perceptron neural networks. *SBA: Controle & Automacao Sociedade Brasileira de Automatica*, 22:647 – 663, 12 2011.

[SMR15] Johann Schumann, Patrick Moosbrugger, and Kristin Y. Rozier. *R2U2: Monitoring and Diagnosis of Security Threats for Unmanned Aerial Systems*, pages 233–249. Springer International Publishing, Cham, 2015.

[SNB+16a] Konstantin Selyunin, Thang Nguyen, Ezio Bartocci, Dejan Nickovic, and Radu Grosu. Monitoring of MTL Specifications With IBM's Spiking-Neuron Model. In *The 19th Design, Automation and Test in Europe Conference and Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016, Proceedings*, 2016.

[SNB+16b] Konstantin Selyunin, Thang Nguyen, Andrei Daniel Basa, Ezio Bartocci, Dejan Nickovic, and Radu Grosu. Applying High-Level Synthesis for Synthesizing Hardware Runtime STL Monitors of Mission-Critical Properties. In *Electronic Proc. of the 13th Design and Verification Conference and Exhibition (DVCon 2016), San Jose, CA, USA, February 28- March 3*, pages 1–8, 2016.

[SNBG16] Konstantin Selyunin, Thang Nguyen, Ezio Bartocci, and Radu Grosu. *Applying Runtime Monitoring for Automotive Electronic Development*, pages 462–469. Springer International Publishing, 2016.

[sou] Source Code. https://github.com/selyunin/truenorth_cpp_hls.git.

[SP89] Almut Schüz and Günther Palm. Density of neurons and synapses in the cerebral cortex of the mouse. *The Journal of Comparative Neurology*, 286(4):442–455, 1989.

[SRB+15] Konstantin Selyunin, Denise Ratasich, Ezio Bartocci, Md. Ariful Islam, Scott A. Smolka, and Radu Grosu. Neural programming: Towards adaptive control in cyber-physical systems. In *54th IEEE Conference on Decision and Control, CDC 2015, Osaka, Japan, December 15-18, 2015*, pages 6978–6985, 2015.

[SWM92] Rahul Sarpeshkar, Lloyd Watts, and Carver Mead. Refractory neuron circuits. *Caltech Authors*, 1992.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[TBF06] S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics.* MIT Press, Cambridge, 2006.

[TCK$^+$02] Joshua T Trachtenberg, Brian E Chen, Graham W Knott, Guoping Feng, Joshua R Sanes, Egbert Welker, and Karel Svoboda. Long-term in vivo imaging of experience-dependent synaptic plasticity in adult cortex. *Nature*, 420(6917):788–794, 2002.

[TRV12] Deian Tabakov, Kristin Y. Rozier, and Moshe Y. Vardi. Optimized temporal monitors for systemc. *Formal Methods in System Design*, 41(3):236–268, 2012.

[Van01] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions.* Department of operations and research and financial engineering, Princeton university, 2001.

[VBG06] Sergiy A. Vilkomir, Jonathan P. Bowen, and Aditya K. Ghose. Formalization and assessment of regulatory requirements for safety-critical software. *Innovations in Systems and Software Engineering*, 2(3):165–178, Dec 2006.

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

[vL09] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications.* Wiley Publishing, 1st edition, 2009.

[VR14] Srikanth Vijayaraghavan and Meyyappan Ramanathan. *A Practical Guide for SystemVerilog Assertions.* Springer Publishing Company, Incorporated, 2014.

[Vre03] Jilles Vreeken. Spiking neural networks, an introduction. Technical report, 2003.

[VVAE94] Carl Van Vreeswijk, LF Abbott, and G Bard Ermentrout. When inhibition not excitation synchronizes neural firing. *Journal of computational neuroscience*, 1(4):313–321, 1994.

[WH08] 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.

[WHTvS14] R. Wang, T. J. Hamilton, J. Tapson, and A. van Schaik. An fpga design framework for large-scale spiking neural networks. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 457–460, June 2014.

[WRR96] S. R. Wicks, C. J. Roehrig, and C. H. Rankin. A dynamic network simulation of the nematode tap withdrawal circuit - Predictions concerning synaptic function using behavioral criteria. *Journal of Neuroscience*, 16:4017–4031, 1996.

[WVdM00] E.A. Wan and R. Van der Merwe. The Unscented Kalman Filter for Nonlinear Estimation. In *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*, pages 153–158, 2000.

[YPA06] Jun Yuan, Carl Pixley, and Adnan Aziz. *Constraint-based verification.* Springer, 2006.

Appendix: SENT/SPC Monitoring

This manual gives a detailed introduction to runtime monitoring of the requirements of the SENT and the SPC communication protocols.

Installation Instructions

The current implementation of the monitoring framework is based on the COSIDE tool. The COSIDE include an eclipse-like environment together with a suite of pre-installed tools (e.g. `gcc`, `SystemC`, `python`, `svn`-client, etc.).

Install COSIDE

Unzip the COSIDE archive into your home directory (i.e. `C:\Users\yourlastname\`). It is important to use the provided archive, since it includes pre-installed plotting libraries (i.e. matplotlib), this way the dependencies for the GUI will be met. As the result of this step, a folder `'coside-2.1.1_mingw32_ifx'` is present in the home directory.

Copy Source Code

If your home directory (i.e. `C:\Users\yourlastname\`) has **NO** folder with the name `workspace`, then unzip the provided 'workspace' archive (keep in mind that paths in windows are case-insensitive i.e. for the OS `WORKSPACE` and `WorkSpace` and `workspace` are all the same). If there is a folder `workspace` in your home directory, then create a folder `'harmonia_ws'`, and copy `'coside-2.1.1_mingw32_ifx'` to `'harmonia_ws'` and unzip the `workspace` archive inside the `'harmonia_ws'` folder.

As the result of this step, one should have the following structure (see Fig. 1):

COSIDE First launch

Important: Before proceeding with the first launch, we need to change the installation path of COSIDE to the path on your PC. In the `'coside-2.1.1_mingw32_ifx'` directory the following files needs to be modified:

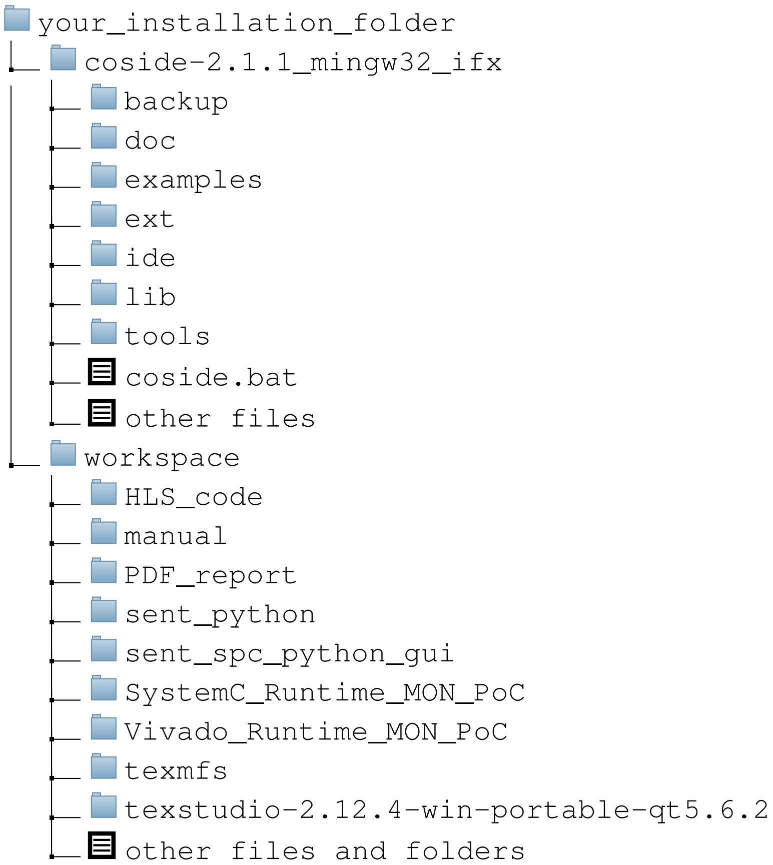
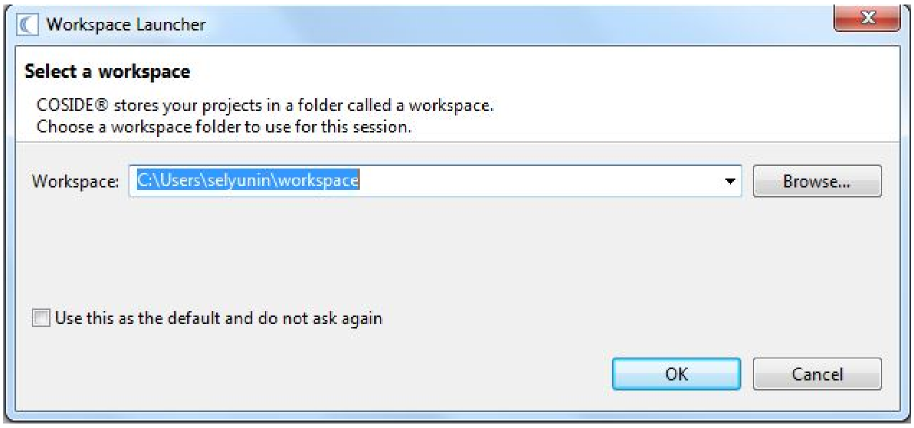


Figure 1: Installation Directory Structure

- `coside.bat`
- `coside_bash.bat`
- `coside_compile.bat`
- `coside_gen_hw.bat`
- `coside_gen_report.bat`
- `coside_gen_report_tcsh.bat`
- `coside_shell.bat`
- `coside_simulate.bat`

In each of these files, the line **21**:
“`set COSIDE_INSTALL_PATH=C:/Users/selyunin/COSIDE~1.1_M`”
should be changed to your installation path i.e.
“`set COSIDE_INSTALL_PATH=C:/Users/yourlastname/COSIDE~1.1_M`”
or
“`set COSIDE_INSTALL_PATH=C:/path/to/your/coside/COSIDE~1.1_M`”.
This is important, otherwise the COSIDE will not launch

Go to `'coside-2.1.1_mingw32_ifx'` folder and double-click 'coside.bat' file, the workspace launcher should pop-up (see Fig. 2):



Select a workspace

COSIDE® stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

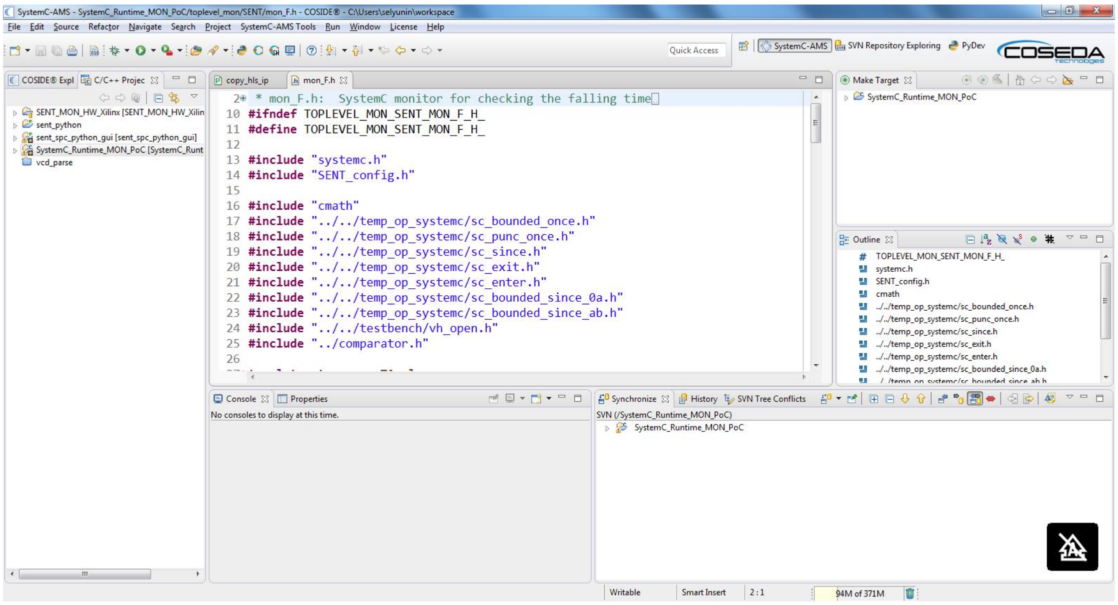
Workspace: C:\Users\selyunin\workspace

☐ Use this as the default and do not ask again

Figure 2: Workspace launcher

In the `Workspace` field (Fig. 2) point to the `workspace` directory from Section 9.4.

As the result of this step, COSIDE working environment should open (see Fig. 3).


SystemC-AMS - SystemC_Runtime_MON_PoC/toplevel_mon/SENT/mon_F.h - COSIDE® - C:\Users\selyunin\workspace
File Edit Source Refactor Navigate Search Project SystemC-AMS Tools Run Window License Help
Quick Access
SystemC-AMS
SVN Repository Exploring
PyDev
COSEDA Technologies
COSIDE® Expl
C/C++ Projec
SENT_MON_HW_Xilinx [SENT_MON_HW_Xilin
sent_python
sent_spc_python_gui [sent_spc_python_gui]
SystemC_Runtime_MON_PoC [SystemC_Runt
vcd_parse
copy_hls_ip
mon_F.h
Make Target
SystemC_Runtime_MON_PoC
Outline
TOPLEVEL_MON_SENT_MON_F_H_
systemc.h
SENT_config.h
cmath
../../temp_op_systemc/sc_bounded_once.h
../../temp_op_systemc/sc_punc_once.h
../../temp_op_systemc/sc_since.h
../../temp_op_systemc/sc_exit.h
../../temp_op_systemc/sc_enter.h
../../temp_op_systemc/sc_bounded_since_0a.h
Console
Properties
Synchronize
History
SVN Tree Conflicts
SVN (/SystemC_Runtime_MON_PoC)
SystemC_Runtime_MON_PoC
Writable
Smart Insert
2 : 1
94M of 371M

```
 * mon_F.h:  SystemC monitor for checking the falling time
#ifndef TOPLEVEL_MON_SENT_MON_F_H_
#define TOPLEVEL_MON_SENT_MON_F_H_

#include "systemc.h"
#include "SENT_config.h"

#include "cmath"
#include "../../temp_op_systemc/sc_bounded_once.h"
#include "../../temp_op_systemc/sc_punc_once.h"
#include "../../temp_op_systemc/sc_since.h"
#include "../../temp_op_systemc/sc_exit.h"
#include "../../temp_op_systemc/sc_enter.h"
#include "../../temp_op_systemc/sc_bounded_since_0a.h"
#include "../../temp_op_systemc/sc_bounded_since_ab.h"
#include "../../testbench/vh_open.h"
#include "../comparator.h"

```

No consoles to display at this time.

Figure 3: COSIDE Environment

Xilinx Tools

For offline SENT/SPC monitoring **no** installation of Xilinx tools is required. In order to generate RTL code (VHDL or Verilog) for online hardware monitoring, Vivado HLS is necessary: one needs to download and install Xilinx Vivado 2016.2 from the Xilinx archive website. Vivado HLS will be installed together with Vivado. The default installation path of the Xilinx tools is: `C:\Xilinx`.

Tutorial

General

This section provides a step-by-step tutorial to get familiar with the SENT/SPC monitoring. We first perform off-line monitoring of the SENT and SPC protocols, and then discuss available options/capabilities/limitations.

SENT Tutorial

In this short tutorial we perform requirements monitoring of the pre-recorded data for the SENT protocol. Prerequisite for this tutorial is preparing the environment as described in Sections 9.4-9.4.

1. Launch the COSIDE tool. (Go to the `'coside-2.1.1_mingw32_ifx'` folder and double click the 'coside.bat' file). The COSIDE GUI will be launched (see Fig. 4)

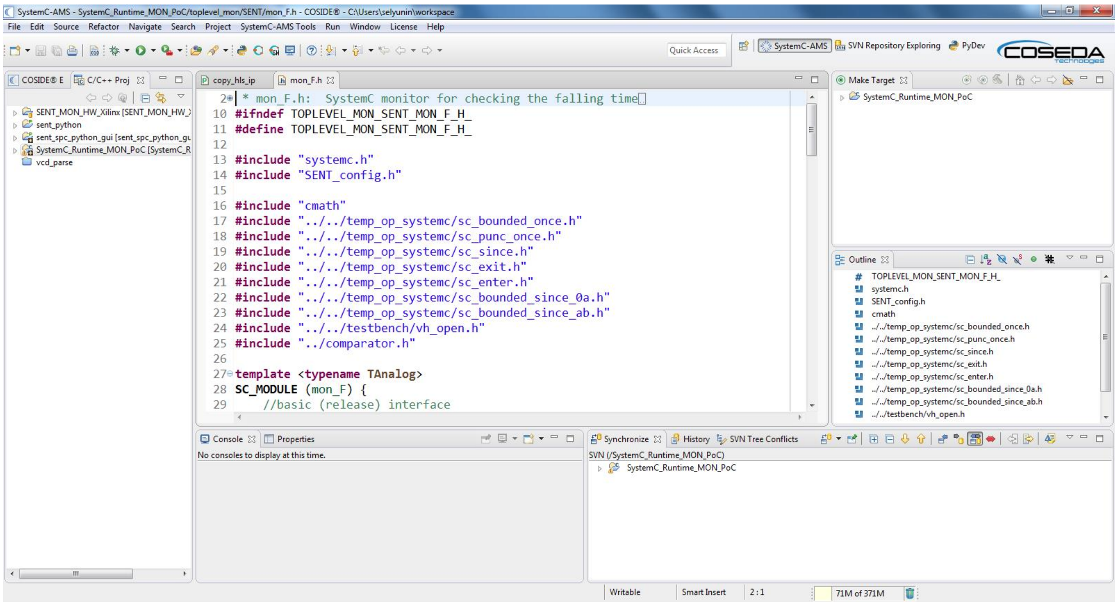
2. Expand `'sent_spc_python_gui'` project (projects `SENT_MON_HW_Xilinx`, `sent_python`, `sent_spc_python_gui`, `SystemC_Runtime_MON_PoC`, etc. are located on the left panel, Fig. 5).

3. Run `'gui_main.py'` file (i.e. right-click on the `'gui_main.py'` file, and select *Run As → 1. Python Run* see Fig. 6).

   As the result of this step, the GUI window for the SENT/SPC protocol monitoring should appear (see Fig. 7). The SENT/SPC Monitor GUI is an interface to the underlying SystemC code, and consists of: (i) the left panel, where parameters of the protocol can be set; (ii) the central plotting panel for visualizing the results; (iii) the bottom panel for configuring electrical parameters and thresholds, (iv) the right panel for configuring and executing the simulation and hardware generation.

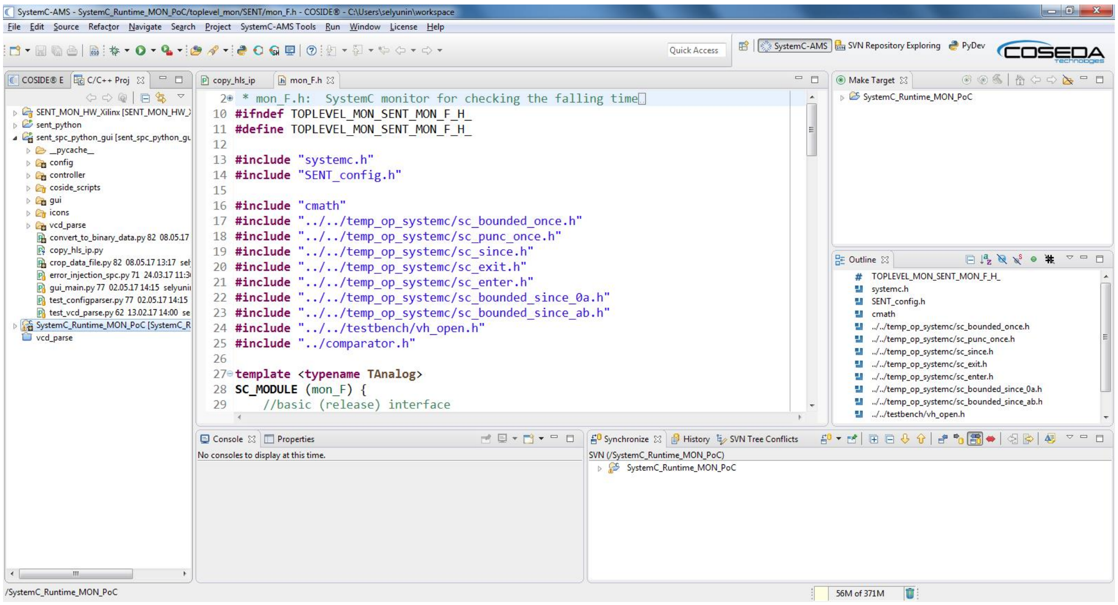
4. Load `'sent_config_Vdd5500_DUT2.ini'` configuration: In the GUI click *File → Open*, select `'sent_config_Vdd5500_DUT2.ini'` file and click *Open*. As the result of this step, the GUI change configuration as in Fig. 8.

   SENT and SPC protocols allow various configurations (e.g. number of nibbles, unit time may vary, and also simulation settings). All these parameters can be configured


SystemC-AMS - SystemC_Runtime_MON_PoC/toplevel_mon/SENT/mon_F.h - COSIDE® - C:\Users\selyunin\workspace
File Edit Source Refactor Navigate Search Project SystemC-AMS Tools Run Window License Help
Quick Access
SystemC-AMS
SVN Repository Exploring
PyDev
COSEDA Technologies
COSIDE® E
C/C++ Proj
SENT_MON_HW_Xilinx [SENT_MON_HW_]
sent_python
sent_spc_python_gui [sent_spc_python_gu
SystemC_Runtime_MON_PoC [SystemC_R
vcd_parse
copy_hls_ip
mon_F.h
Make Target
SystemC_Runtime_MON_PoC
Outline
TOPLEVEL_MON_SENT_MON_F_H_
systemc.h
SENT_config.h
cmath
../../temp_op_systemc/sc_bounded_once.h
../../temp_op_systemc/sc_punc_once.h
../../temp_op_systemc/sc_since.h
../../temp_op_systemc/sc_exit.h
../../temp_op_systemc/sc_enter.h
../../temp_op_systemc/sc_bounded_since_0a.h
../../temp_op_systemc/sc_bounded_since_ab.h
../../testbench/vh_open.h
Console
Properties
No consoles to display at this time.
Synchronize
History
SVN Tree Conflicts
SVN (/SystemC_Runtime_MON_PoC)
SystemC_Runtime_MON_PoC
Writable
Smart Insert
2 : 1
71M of 371M

```
 * mon_F.h:  SystemC monitor for checking the falling time
#ifndef TOPLEVEL_MON_SENT_MON_F_H_
#define TOPLEVEL_MON_SENT_MON_F_H_

#include "systemc.h"
#include "SENT_config.h"

#include "cmath"
#include "../../temp_op_systemc/sc_bounded_once.h"
#include "../../temp_op_systemc/sc_punc_once.h"
#include "../../temp_op_systemc/sc_since.h"
#include "../../temp_op_systemc/sc_exit.h"
#include "../../temp_op_systemc/sc_enter.h"
#include "../../temp_op_systemc/sc_bounded_since_0a.h"
#include "../../temp_op_systemc/sc_bounded_since_ab.h"
#include "../../testbench/vh_open.h"
#include "../comparator.h"

template <typename TAnalog>
SC_MODULE (mon_F) {
    //basic (release) interface
```

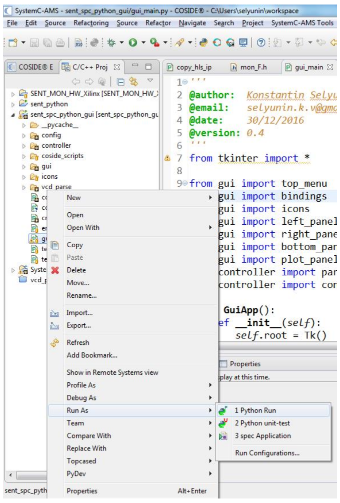
Figure 4: COSIDE GUI


SystemC-AMS - SystemC_Runtime_MON_PoC/toplevel_mon/SENT/mon_F.h - COSIDE® - C:\Users\selyunin\workspace
File Edit Source Refactor Navigate Search Project SystemC-AMS Tools Run Window License Help
Quick Access
SystemC-AMS
SVN Repository Exploring
PyDev
COSEDA Technologies
COSIDE® E
C/C++ Proj
SENT_MON_HW_Xilinx [SENT_MON_HW_]
sent_python
sent_spc_python_gui [sent_spc_python_gu
__pycache__
config
controller
coside_scripts
gui
icons
vcd_parse
convert_to_binary_data.py 82 08.05.17
copy_hls_ip.py
crop_data_file.py 82 08.05.17 13:17 sel
error_injection_spc.py 71 24.03.17 11:3
gui_main.py 77 02.05.17 14:15 selyuni
test_configparser.py 77 02.05.17 14:15
test_vcd_parse.py 62 13.02.17 14:00 se
SystemC_Runtime_MON_PoC [SystemC_R
vcd_parse
copy_hls_ip
mon_F.h
Make Target
SystemC_Runtime_MON_PoC
Outline
TOPLEVEL_MON_SENT_MON_F_H_
systemc.h
SENT_config.h
cmath
../../temp_op_systemc/sc_bounded_once.h
../../temp_op_systemc/sc_punc_once.h
../../temp_op_systemc/sc_since.h
../../temp_op_systemc/sc_exit.h
../../temp_op_systemc/sc_enter.h
../../temp_op_systemc/sc_bounded_since_0a.h
../../temp_op_systemc/sc_bounded_since_ab.h
../../testbench/vh_open.h
Console
Properties
No consoles to display at this time.
Synchronize
History
SVN Tree Conflicts
SVN (/SystemC_Runtime_MON_PoC)
SystemC_Runtime_MON_PoC
/SystemC_Runtime_MON_PoC
56M of 371M

```
 * mon_F.h:  SystemC monitor for checking the falling time
#ifndef TOPLEVEL_MON_SENT_MON_F_H_
#define TOPLEVEL_MON_SENT_MON_F_H_

#include "systemc.h"
#include "SENT_config.h"

#include "cmath"
#include "../../temp_op_systemc/sc_bounded_once.h"
#include "../../temp_op_systemc/sc_punc_once.h"
#include "../../temp_op_systemc/sc_since.h"
#include "../../temp_op_systemc/sc_exit.h"
#include "../../temp_op_systemc/sc_enter.h"
#include "../../temp_op_systemc/sc_bounded_since_0a.h"
#include "../../temp_op_systemc/sc_bounded_since_ab.h"
#include "../../testbench/vh_open.h"
#include "../comparator.h"

template <typename TAnalog>
SC_MODULE (mon_F) {
    //basic (release) interface
```

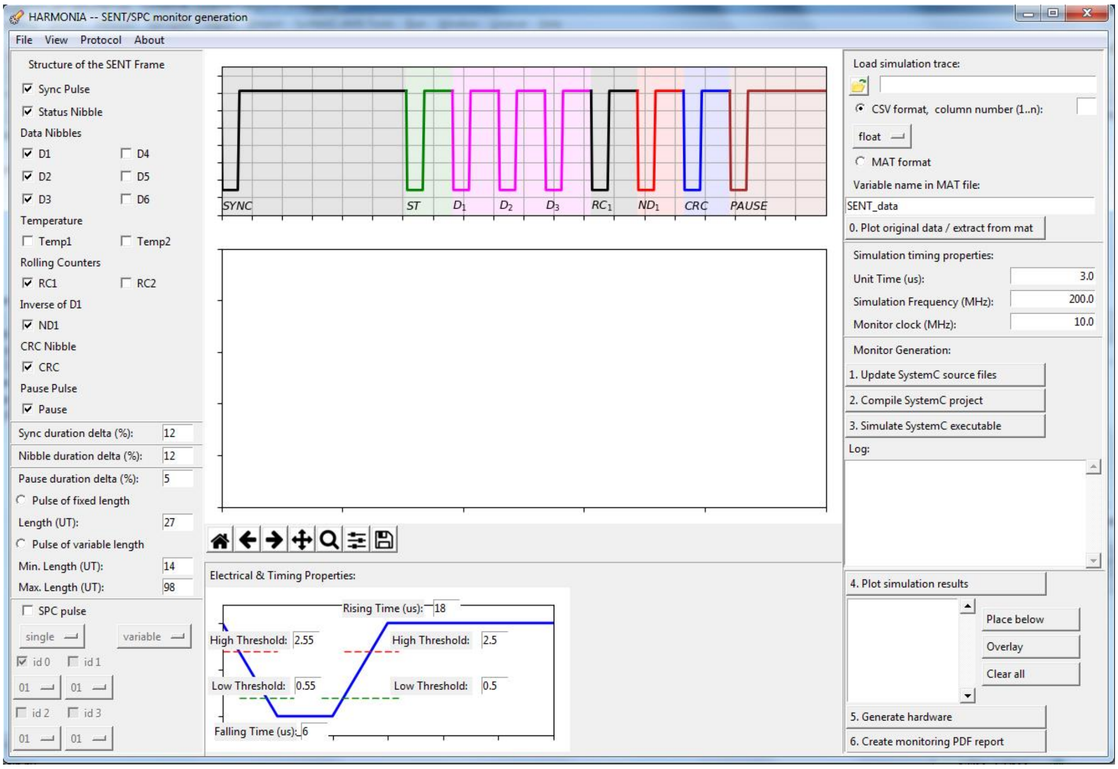
Figure 5: Expanded '`sent_spc_python_gui`' project

in the GUI, but in order to save time, one can preset configuration of the sensor and simulation parameters in a special "`*.ini`" file, which is a human-friendly plain text format. Whenever opened, these files are read and parsed, and the



```
'''
@author:   Konstantin Selyu
@email:    selyunin.k.v@gmc
@date:     30/12/2016
@version: 0.4
'''
from tkinter import *

from gui import top_menu
gui import bindings
gui import icons
gui import left_panel
gui import right_pane
gui import bottom_par
gui import plot_panel
controller import par
controller import cor

GuiApp():
ef __init__(self):
    self.root = Tk()
```

Figure 6: Launching the SENT/SPC Monitor GUI


HARMONIA -- SENT/SPC monitor generation
File
View
Protocol
About
SYNC
ST
D_1
D_2
D_3
RC_1
ND_1
CRC
PAUSE
Rising Time (us):
High Threshold:
Low Threshold:
Falling Time (us):
0. Plot original data / extract from mat
1. Update SystemC source files
2. Compile SystemC project
3. Simulate SystemC executable
4. Plot simulation results
Place below
Overlay
Clear all
5. Generate hardware
6. Create monitoring PDF report

Structure of the SENT Frame

☑ Sync Pulse
☑ Status Nibble

Data Nibbles
☑ D1 ☐ D4
☑ D2 ☐ D5
☑ D3 ☐ D6

Temperature
☐ Temp1 ☐ Temp2

Rolling Counters
☑ RC1 ☐ RC2

Inverse of D1
☑ ND1

CRC Nibble
☑ CRC

Pause Pulse
☑ Pause

Sync duration delta (%): 12
Nibble duration delta (%): 12
Pause duration delta (%): 5
○ Pulse of fixed length
Length (UT): 27
○ Pulse of variable length
Min. Length (UT): 14
Max. Length (UT): 98

☐ SPC pulse
single variable
☑ id 0 ☐ id 1
01 01
☐ id 2 ☐ id 3
01 01

Electrical & Timing Properties:
Rising Time (us): 18
High Threshold: 2.55
Low Threshold: 0.55
High Threshold: 2.5
Low Threshold: 0.5
Falling Time (us): 6

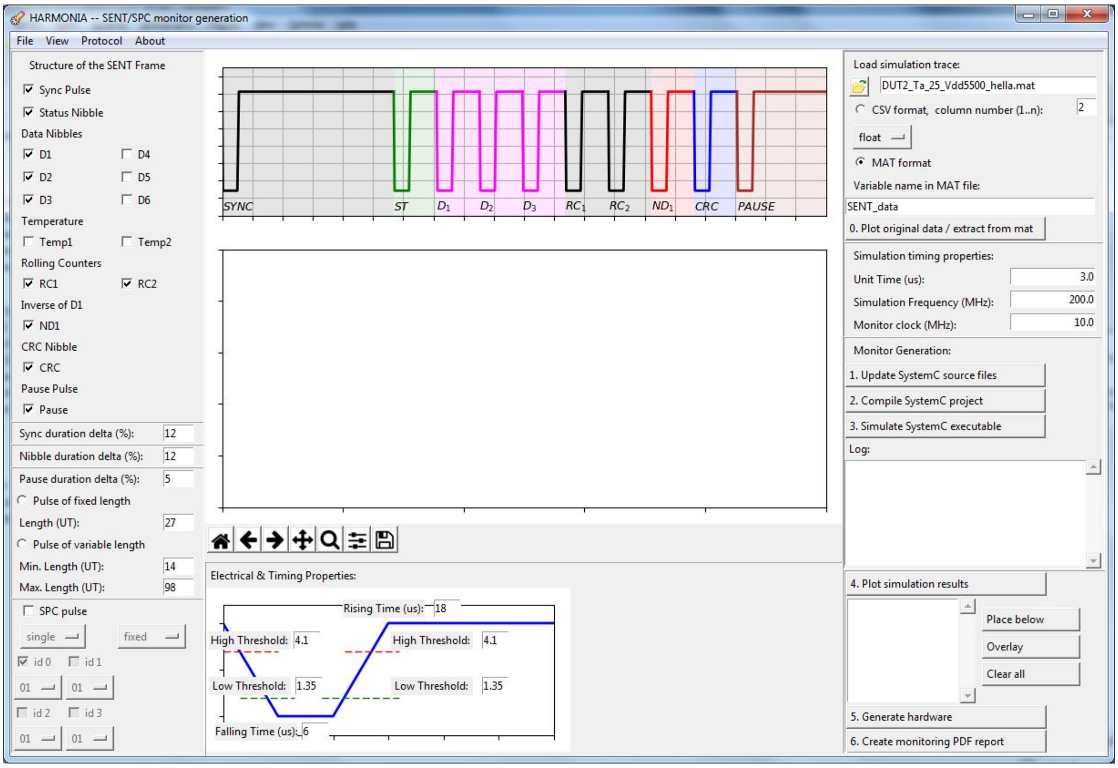
Load simulation trace:
● CSV format, column number (1..n):
float
○ MAT format
Variable name in MAT file: SENT_data

Simulation timing properties:
Unit Time (us): 3.0
Simulation Frequency (MHz): 200.0
Monitor clock (MHz): 10.0

Monitor Generation:

Log:

Figure 7: SENT/SPC Monitor GUI


HARMONIA -- SENT/SPC monitor generation
File View Protocol About
Structure of the SENT Frame
SYNC
ST
D_1
D_2
D_3
RC_1
RC_2
ND_1
CRC
PAUSE
Electrical & Timing Properties:
0. Plot original data / extract from mat
Simulation timing properties:
Monitor Generation:
1. Update SystemC source files
2. Compile SystemC project
3. Simulate SystemC executable
Log:
4. Plot simulation results
Place below
Overlay
Clear all
5. Generate hardware
6. Create monitoring PDF report

☑ Sync Pulse
☑ Status Nibble
Data Nibbles
☑ D1 ☐ D4
☑ D2 ☐ D5
☑ D3 ☐ D6
Temperature
☐ Temp1 ☐ Temp2
Rolling Counters
☑ RC1 ☑ RC2
Inverse of D1
☑ ND1
CRC Nibble
☑ CRC
Pause Pulse
☑ Pause
Sync duration delta (%): 12
Nibble duration delta (%): 12
Pause duration delta (%): 5
○ Pulse of fixed length
Length (UT): 27
○ Pulse of variable length
Min. Length (UT): 14
Max. Length (UT): 98
☐ SPC pulse
single fixed
☑ id 0 ☐ id 1
01 01
☐ id 2 ☐ id 3
01 01

Rising Time (us): 18
High Threshold: 4.1
Low Threshold: 1.35
High Threshold: 4.1
Low Threshold: 1.35
Falling Time (us): 6

Load simulation trace: DUT2_Ta_25_Vdd5500_hella.mat
○ CSV format, column number (1..n): 2
float
● MAT format
Variable name in MAT file: SENT_data
Unit Time (us): 3.0
Simulation Frequency (MHz): 200.0
Monitor clock (MHz): 10.0

Figure 8: Applying `'sent_config_Vdd5500_DUT2.ini'` configuration

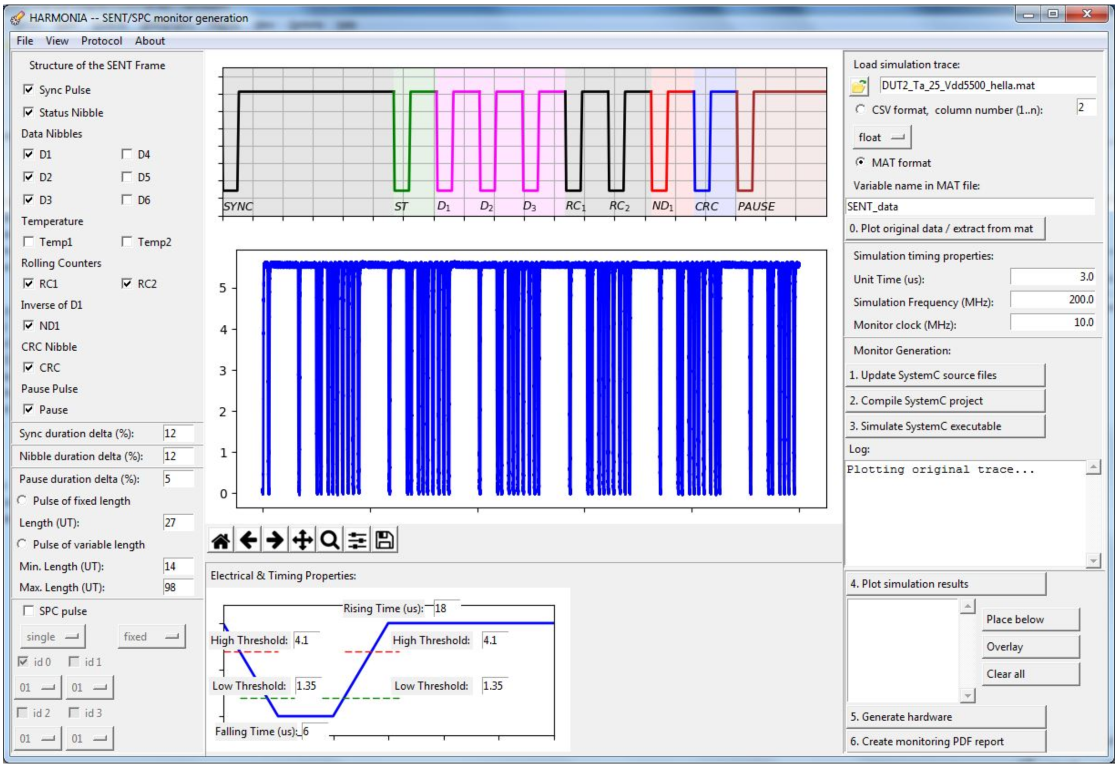
configuration from these files is applied to the GUI. In similar way, one can also save a current configuration of the GUI to a configuration file (i.e. *File* → *Save* or *File* → *Save as*).

5. Plot the recorded signal from the data file: Click the button *"0. Plot original data / extract from mat"* on the right panel of the GUI. As the result, the recorded SENT signal is printed in the central plot of the GUI, see Fig. 9.

   Currently supported data formats are MATLAB (`*.mat`) and CSV (comma separated format). For a MATLAB file (`*.mat`), it is necessary to specify a variable name (e.g. `SENT_data` in our case) that holds data values. If a CSV file is used as a simulation data source, it is necessary to specify the column number for the data to plot. Plotting original data is **necessary**, and should be performed each time whenever a data file is changed (since this way the input data files are evaluated).

6. Update monitor parameters & simulation settings of the SystemC project: Click the button *"1. Update SystemC source files"* on the right panel of the GUI.

   During this step configurations of monitors (e.g. internal buffer sizes, slope times, frame structure, etc) will be applied to the SystemC project.


HARMONIA -- SENT/SPC monitor generation
File
View
Protocol
About
SYNC
ST
D_1
D_2
D_3
RC_1
RC_2
ND_1
CRC
PAUSE
0
1
2
3
4
5
0. Plot original data / extract from mat
1. Update SystemC source files
2. Compile SystemC project
3. Simulate SystemC executable
4. Plot simulation results
Place below
Overlay
Clear all
5. Generate hardware
6. Create monitoring PDF report

Structure of the SENT Frame
☑ Sync Pulse
☑ Status Nibble
Data Nibbles
☑ D1 ☐ D4
☑ D2 ☐ D5
☑ D3 ☐ D6
Temperature
☐ Temp1 ☐ Temp2
Rolling Counters
☑ RC1 ☑ RC2
Inverse of D1
☑ ND1
CRC Nibble
☑ CRC
Pause Pulse
☑ Pause
Sync duration delta (%): 12
Nibble duration delta (%): 12
Pause duration delta (%): 5
○ Pulse of fixed length
Length (UT): 27
○ Pulse of variable length
Min. Length (UT): 14
Max. Length (UT): 98
☐ SPC pulse
single fixed
☑ id 0 ☐ id 1
01 01
☐ id 2 ☐ id 3
01 01

Electrical & Timing Properties:
Rising Time (us): 18
High Threshold: 4.1
Low Threshold: 1.35
High Threshold: 4.1
Low Threshold: 1.35
Falling Time (us): 6

Load simulation trace:
DUT2_Ta_25_Vdd5500_hella.mat
○ CSV format, column number (1..n): 2
float
● MAT format
Variable name in MAT file:
SENT_data

Simulation timing properties:
Unit Time (us): 3.0
Simulation Frequency (MHz): 200.0
Monitor clock (MHz): 10.0

Monitor Generation:

Log:

```
Plotting original trace...
```

Figure 9: Plotting original sensor data

7. Compile SystemC project: Click the button *"2. Compile SystemC project"* on the right panel of the GUI. A terminal window "`COSIDE TCSH`" will pop-up, and the necessary compilation steps will be performed. Do not close the terminal window, it will disappear after the compilation is finished.

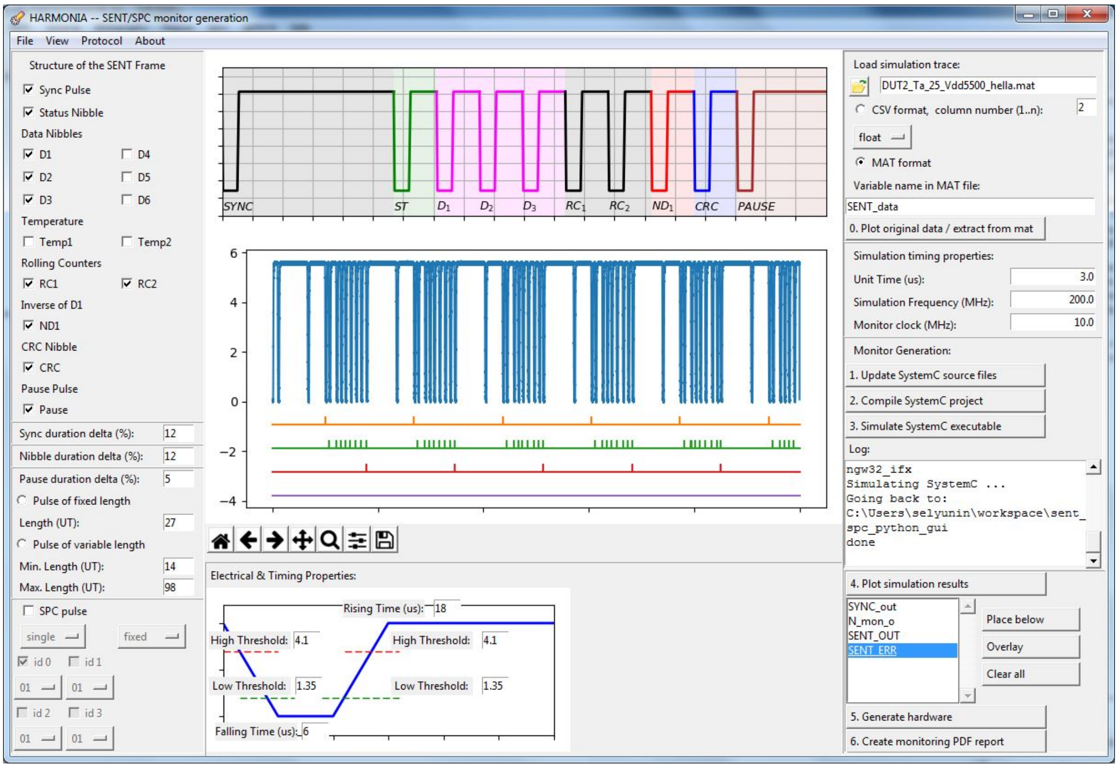
   In this step we compile a SystemC testbench and a runtime monitor, which will check the requirements of the SENT protocol.

8. Run the SystemC simulation: Click the button *"3. Simulate SystemC executable"* on the right panel of the GUI. A terminal window "`COSIDE TCSH`" will pop-up, and the simulation will be launched. Do not close the terminal window, it will disappear after the simulation is finished.

   In this step we run SystemC executable, play the recorded sensor data to the monitor, and save monitoring results in the $\star$`.vcd` file: `monitor_dump.vcd`. We also create log file `analysis_log.txt`, store decoded values in `analysis_short.csv`. All the files are created in
   `\workspace\SystemC_Runtime_MON_PoC\DEBUG\testbench\`

9. Plot simulation results: Click the button *"4. Plot simulation results"* on the right

panel of the GUI. Plotting might take some time, please be patient, During this step the GUI parses the `monitor_dump.vcd` file and it might take a while.



HARMONIA -- SENT/SPC monitor generation
File View Protocol About
Structure of the SENT Frame
SYNC
ST
D_1
D_2
D_3
RC_1
RC_2
ND_1
CRC
PAUSE
6
4
2
0
−2
−4
Electrical & Timing Properties:
0. Plot original data / extract from mat
Simulation timing properties:
Monitor Generation:
1. Update SystemC source files
2. Compile SystemC project
3. Simulate SystemC executable
4. Plot simulation results
Place below
Overlay
Clear all
5. Generate hardware
6. Create monitoring PDF report

☑ Sync Pulse
☑ Status Nibble
Data Nibbles
☑ D1 ☐ D4
☑ D2 ☐ D5
☑ D3 ☐ D6
Temperature
☐ Temp1 ☐ Temp2
Rolling Counters
☑ RC1 ☑ RC2
Inverse of D1
☑ ND1
CRC Nibble
☑ CRC
Pause Pulse
☑ Pause
Sync duration delta (%): 12
Nibble duration delta (%): 12
Pause duration delta (%): 5
○ Pulse of fixed length
Length (UT): 27
○ Pulse of variable length
Min. Length (UT): 14
Max. Length (UT): 98
☐ SPC pulse
single fixed
☑ id 0 ☐ id 1
01 01
☐ id 2 ☐ id 3
01 01

Rising Time (us): 18
High Threshold: 4.1
Low Threshold: 1.35
High Threshold: 4.1
Low Threshold: 1.35
Falling Time (us): 6

Load simulation trace: DUT2_Ta_25_Vdd5500_hella.mat
○ CSV format, column number (1..n): 2
float
● MAT format
Variable name in MAT file: SENT_data
Unit Time (us): 3.0
Simulation Frequency (MHz): 200.0
Monitor clock (MHz): 10.0

Log:

```
ngw32_ifx
Simulating SystemC ...
Going back to:
C:\Users\selyunin\workspace\sent_
spc_python_gui
done
```

SYNC_out
N_mon_o
SENT_OUT
SENT_ERR

Figure 10: Plotting signals from the `monitor_dump.vcd`

As the result of this step, one should see the data signal (but in a slightly different color), and a list of signals in the field below the button *"4. Plot simulation results"* (i.e. the signals `SYNC_out`, `N_mon_o`, `SENT_out`, `SENT_err`).

We then plot these signals in the GUI: Select `'SYNC_out'` signal from the list, and click `"Place below"` button. The signal will be plotted under the sensor data. We then repeat this procedure for other signals: Fig. 10 shows the expected result.

`SYNC_out`, `N_mon_o`, `SENT_out` are binary signals with the following meaning: `SYNC_out` – synchronization pulse, `N_mon_o` – nibble pulse, `SENT_out` – SENT frame. "1" on each on these signals means that the correct pulse (or frame) has been detected and all the requirements for this pulse (or frame) are met.

10. Examine log files: Go to the directory `\workspace\SystemC_Runtime_MON_PoC\DEBUG\testbench\`, and open in a text editor (e.g. `Notepad++`) files `'analysis_log.txt'`, and `'analysis_short.csv'`.

11. Generate PDF report: Click the button *“6. Create monitoring PDF report”* on the right panel of the GUI. A terminal window `"COSIDE TCSH"` will pop-up, and the report generation will be launched. Do not close the terminal window, it will disappear after the creation of the PDF report. The PDF report is located under: `workspace\PDF_report\report.pdf`.

    Important: **Always** close the file `report.pdf` (i.e. exit Adobe Viewer) before generating new PDF report. As report generation results in `report.pdf`, the file cannot be updated, if it is opened in another program. This also means, that in order to keep the report for the later, it should be copied in a separate location.

This is the end of the short SENT tutorial. We have seen the general flow of the off-line monitoring, and performed the required steps. We used GUI as an interface to the underlying SystemC monitor code, we checked previously recorded signal, specified the data file, configured the monitor and updated the parameters in the SystemC project, compiled and simulated the SystemC project. After simulation finished we checked the monitor outputs and summarized the monitoring results in a report.

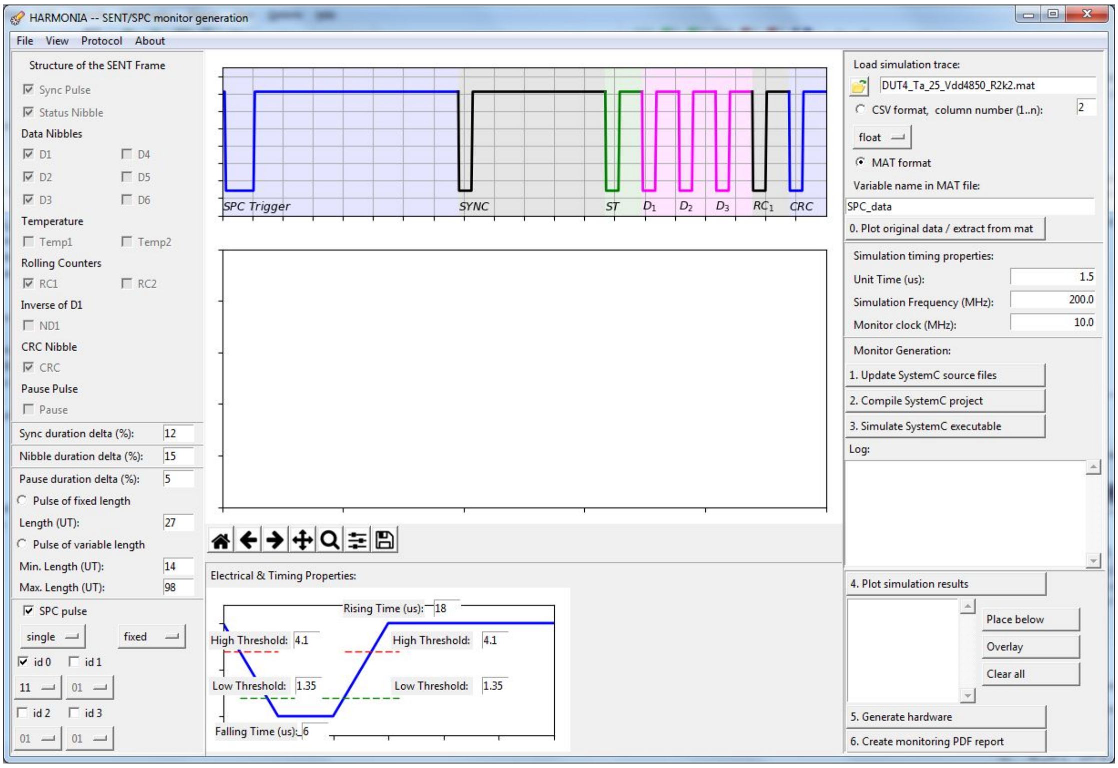
SPC Tutorial

SENT offline monitoring (Section 9.4) is a prerequisite for this tutorial. Section 9.4 introduces the key concepts and outlines the off-line monitoring procedure, hence we try to reduce repetitions as much as possible. To start with this tutorial, it is necessary to perform the steps 1, 2, 3 from the SENT tutorial. As a result, the SENT/SPC Monitoring GUI is launched (see Fig. 7).

1. Load configuration `'spc_config_Vdd4850_fix_DUT4.ini'`: In the GUI click *File → Open*, select `'spc_config_Vdd4850_fix_DUT4.ini'` file and click *Open*. As the result of this step, the GUI change configuration as in Fig. 11.

   As seen from Fig. 11, SPC trigger pulse is now active. According to the specification, the pulse length can be either *fixed* or *variable*. In this example we consider configuration with the pulse of fixed length. Since the SPC communication is initiated by a microcontroller, one or more (up to four) sensors can be on the bus. The mode of operation can be selected: either *single* or *bus*. An SPC response frame can have different configurations (e.g. temperature nibbles, rolling counters may be present or absent). At the bottom of the left GUI panel, one can select a sensor configuration (`01 - 16`). For reference, the frame structure for each configuration are summarized under the top menu *Protocol → SPC*.

2. Plot the recorded signal from the data file `'DUT4_Ta_25_Vdd4850_R2k2.mat'`: Click the button *“0. Plot original data / extract from mat”* on the right panel of the GUI. As the result of this step, SPC data signal should appear on the plotting panel.


HARMONIA -- SENT/SPC monitor generation
File
View
Protocol
About
SPC Trigger
SYNC
ST
D_1
D_2
D_3
RC_1
CRC
Electrical & Timing Properties:
0. Plot original data / extract from mat
1. Update SystemC source files
2. Compile SystemC project
3. Simulate SystemC executable
4. Plot simulation results
Place below
Overlay
Clear all
5. Generate hardware
6. Create monitoring PDF report

Structure of the SENT Frame
☑ Sync Pulse
☑ Status Nibble
Data Nibbles
☑ D1 ☐ D4
☑ D2 ☐ D5
☑ D3 ☐ D6
Temperature
☐ Temp1 ☐ Temp2
Rolling Counters
☑ RC1 ☐ RC2
Inverse of D1
☐ ND1
CRC Nibble
☑ CRC
Pause Pulse
☐ Pause
Sync duration delta (%): 12
Nibble duration delta (%): 15
Pause duration delta (%): 5
○ Pulse of fixed length
Length (UT): 27
○ Pulse of variable length
Min. Length (UT): 14
Max. Length (UT): 98
☑ SPC pulse
single fixed
☑ id 0 ☐ id 1
11 01
☐ id 2 ☐ id 3
01 01

Rising Time (us): 18
High Threshold: 4.1
High Threshold: 4.1
Low Threshold: 1.35
Low Threshold: 1.35
Falling Time (us): 6

Load simulation trace:
DUT4_Ta_25_Vdd4850_R2k2.mat
○ CSV format, column number (1..n): 2
float
● MAT format
Variable name in MAT file:
SPC_data
Simulation timing properties:
Unit Time (us): 1.5
Simulation Frequency (MHz): 200.0
Monitor clock (MHz): 10.0
Monitor Generation:
Log:

Figure 11: Loading the SPC configuration

3. Update the SystemC project: Click the button *"1. Update SystemC source files"*.

4. Compile SystemC project: Click the button *"2. Compile SystemC project"*.
   Do **not** close "`COSIDE TCSH`" window.

5. Run the SystemC simulation: Click the button *"3. Simulate SystemC executable"*.
   Do **not** close "`COSIDE TCSH`" window.

6. Plot simulation results: Click the button *"4. Plot simulation results"*. It might take some time to plot the results, as the file `monitor_dump.vcd` needs to be parsed.

   Important: By default only a short list of signals is displayed in the GUI. To display all recorded signals, go to the top menu *View*, select *Display all signals*, and then press the button *"4. Plot simulation results"*.

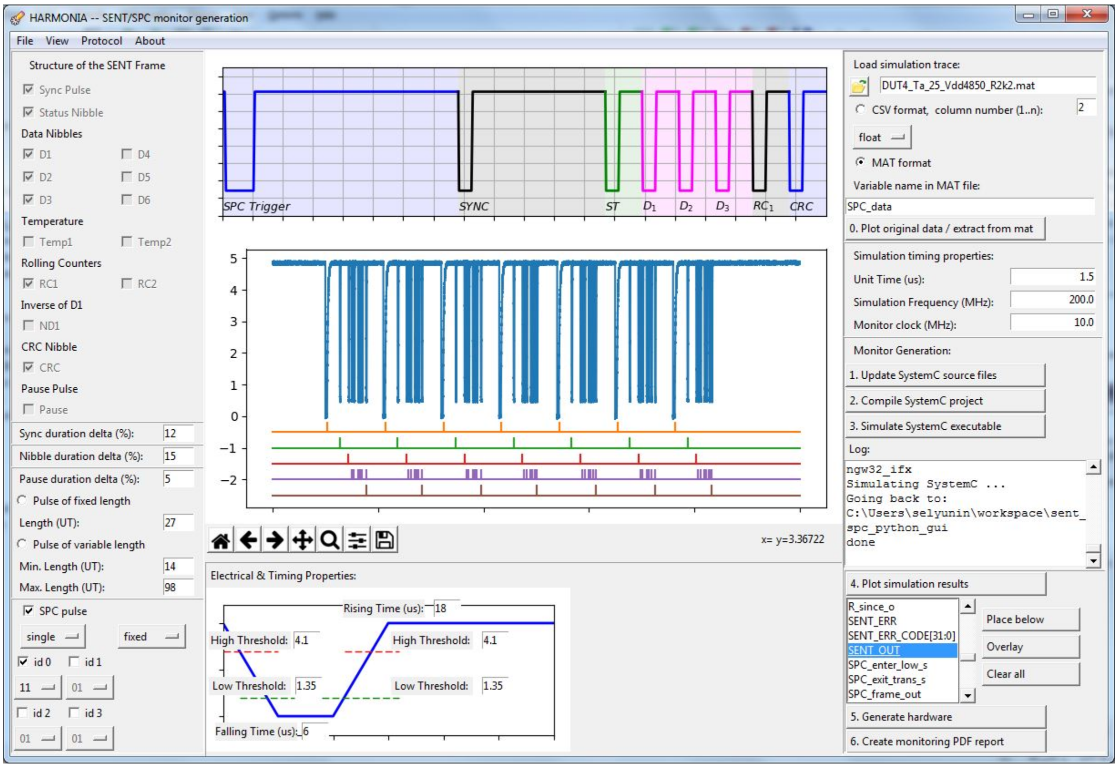
   Display all recorded signals as described above, and plot below the sensor data (by selecting the signal and clicking the button *Place below*) the following signals:

   a) `SPC_tmlow_s` – monitor output for SPC tmlow requirement;
   b) `SPC_mon_s` – monitor output for SPC trigger pulse;
   c) `SYNC_out` – monitor output for the SYNC pulse;

d) `N_mon_o` – monitor output for the NIBBLE pulse;

e) `SENT_out` – monitor output for the SENT response frame.

As a result of this step, the GUI window displays the sensor data together with the monitoring signals (see Fig. 12).


HARMONIA -- SENT/SPC monitor generation

Structure of the SENT Frame

☑ Sync Pulse
☑ Status Nibble
Data Nibbles: ☑ D1 ☐ D4 ☑ D2 ☐ D5 ☑ D3 ☐ D6
Temperature: ☐ Temp1 ☐ Temp2
Rolling Counters: ☑ RC1 ☐ RC2
Inverse of D1: ☐ ND1
CRC Nibble: ☑ CRC
Pause Pulse: ☐ Pause

Sync duration delta (%): 12
Nibble duration delta (%): 15
Pause duration delta (%): 5
○ Pulse of fixed length
Length (UT): 27
○ Pulse of variable length
Min. Length (UT): 14
Max. Length (UT): 98

☑ SPC pulse
single fixed
☑ id 0 ☐ id 1
11 01
☐ id 2 ☐ id 3
01 01

Electrical & Timing Properties:
Rising Time (us): 18
High Threshold: 4.1
Low Threshold: 1.35
High Threshold: 4.1
Low Threshold: 1.35
Falling Time (us): 6

Load simulation trace: DUT4_Ta_25_Vdd4850_R2k2.mat
○ CSV format, column number (1..n): 2
float
● MAT format
Variable name in MAT file: SPC_data

Simulation timing properties:
Unit Time (us): 1.5
Simulation Frequency (MHz): 200.0
Monitor clock (MHz): 10.0

Log:

```
ngw32_ifx
Simulating SystemC ...
Going back to:
C:\Users\selyunin\workspace\sent_
spc_python_gui
done
```

Figure 12: SPC Monitoring Results

7. Generate the PDF report: Click the button *"6. Create monitoring PDF report"*. Do **not** close "`COSIDE TCSH`" window.
   The PDF report is located under: `workspace\PDF_report\report.pdf`.

   One can also examine log files, as described in step 10 of the SENT tutorial.

This is the end of the SPC tutorial.

SENT/SPC Monitoring: Details

The monitoring framework aims to provide runtime monitors of formally defined properties. For a past-STL property a monitor can be constructed from *temporal operator Lib.* and simulated on a previously recorded data (either from an oscilloscope or from the

Chipscope). The *synthesizable* version contains `tcl`-scripts for generating HLD (VHDL or Verilog) code from the SystemC monitor using High-Level Synthesis (HLS).

The HLS imposes the restrictions on the SystemC source code, hence the current state is two versions of temporal operator libraries: *behavioral* & *synthesizable*. The *behavioral* one allows to prototype a solution with all the available SystemC/C++ features, and then use it as a reference for generating hardware monitors with *synthesizable* version.

The GUI is based on a cross-platform `tcl-tk` library and implemented in python. For interfacing with `tcl-tk` we use the `tkinter` package, and for plotting we rely on `matplotlib`. This design decision allows to fully leverage shipped software with the COSIDE and reduce to a minimum additional required packages.

Core Capabilities and Functionality

The main functionality of the SENT/SPC offline monitoring GUI is follows:

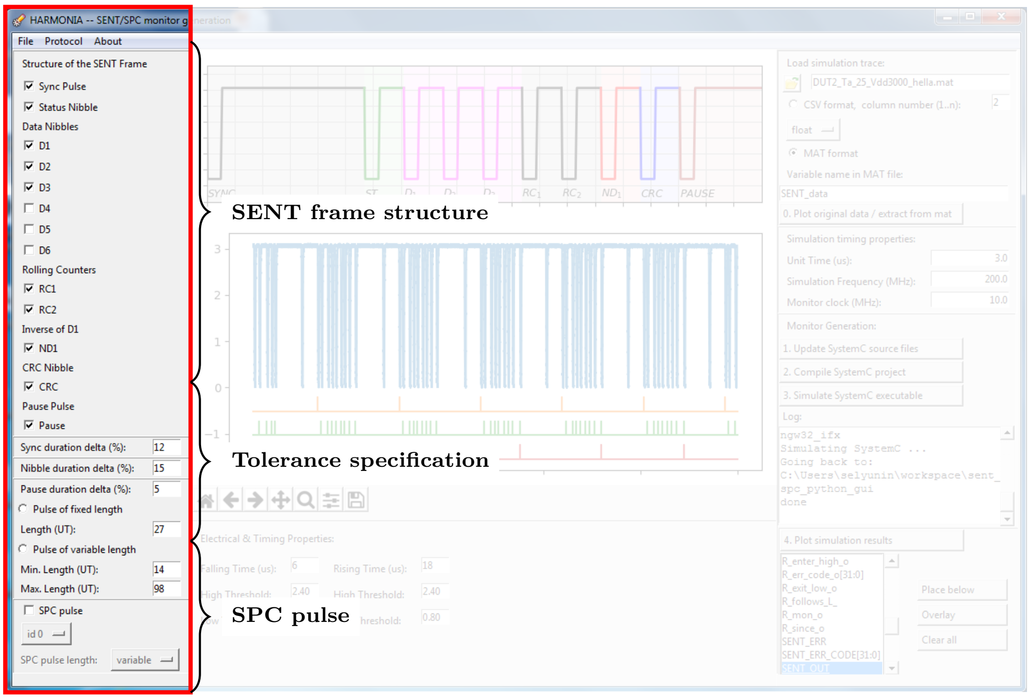
1. Runtime Monitor (in SystemC simulation runtime) SENT or SPC protocols;
2. Monitoring of electrical interface requirements and timing requirements for the synchronization pulse, nibble pulses and pause pulse;
3. Monitoring electrical interface requirements and timing requirements for the SPC trigger pulse (if SPC protocol is selected);
4. Reading the recorded traces in `.csv` or MATLAB `.mat` file formats in *float*, *integer* or *binary* (Chip Scope) representation;
5. Separate clocks for the recorded data and the monitor during the simulation runtime
6. Specification of the Unit Time (UT);
7. Compiling, simulating, and visualizing simulation results from the SystemC backend;
8. Loading and saving the pre-configured settings file (`.ini` format);
9. Data decoding of the SENT frames and generation of a monitoring report;

GUI: Detailed look

Left panel: Specification of the Frame

The SENT standard allows multiple configurations of the frame (i.e. the number of data nibbles may vary, rolling counter may be present/absent, and the pause pulse may be omitted. The shape of the SPC trigger pulse depends on the single/bus mode, a sensor id, and whether the SPC trigger pulses if of fixed or variable length. All these options for the protocols can be specified on the left panel (see Fig. 13).

As the original specifications give only absolute values for the nominal case, even a tiny deviation would result in the false positive. To remedy this effect, we introduce controllable tolerance intervals δ which the user can specify for nibbles, sync and pause pulses (Fig. 13).


HARMONIA -- SENT/SPC monitor generation
File
Protocol
About
SENT frame structure
Tolerance specification
SPC pulse
SYNC
ST
D₁
D₂
D₃
RC₁
RC₂
ND₁
CRC
PAUSE
3
2
1
0
−1
0. Plot original data / extract from mat
1. Update SystemC source files
2. Compile SystemC project
3. Simulate SystemC executable
4. Plot simulation results
Place below
Overlay
Clear all

Structure of the SENT Frame

☑ Sync Pulse
☑ Status Nibble

Data Nibbles

☑ D1
☑ D2
☑ D3
☐ D4
☐ D5
☐ D6

Rolling Counters

☑ RC1
☑ RC2

Inverse of D1

☑ ND1

CRC Nibble

☑ CRC

Pause Pulse

☑ Pause

Sync duration delta (%): 12
Nibble duration delta (%): 15
Pause duration delta (%): 5
○ Pulse of fixed length
Length (UT): 27
○ Pulse of variable length
Min. Length (UT): 14
Max. Length (UT): 98
☐ SPC pulse
id 0
SPC pulse length: variable

Electrical & Timing Properties:
Falling Time (us): 6
Rising Time (us): 18
High Threshold: 2.40
High Threshold: 2.40
Threshold: 0.80

Load simulation trace: DUT2_Ta_25_Vdd3000_hella.mat
○ CSV format, column number (1..n): 2
float
● MAT format
Variable name in MAT file: SENT_data

Simulation timing properties:
Unit Time (us): 3.0
Simulation Frequency (MHz): 200.0
Monitor clock (MHz): 10.0

Monitor Generation:

Log:

```
ngw32_ifx
Simulating SystemC ...
Going back to:
C:\Users\selyunin\workspace\sent_
spc_python_gui
done
```

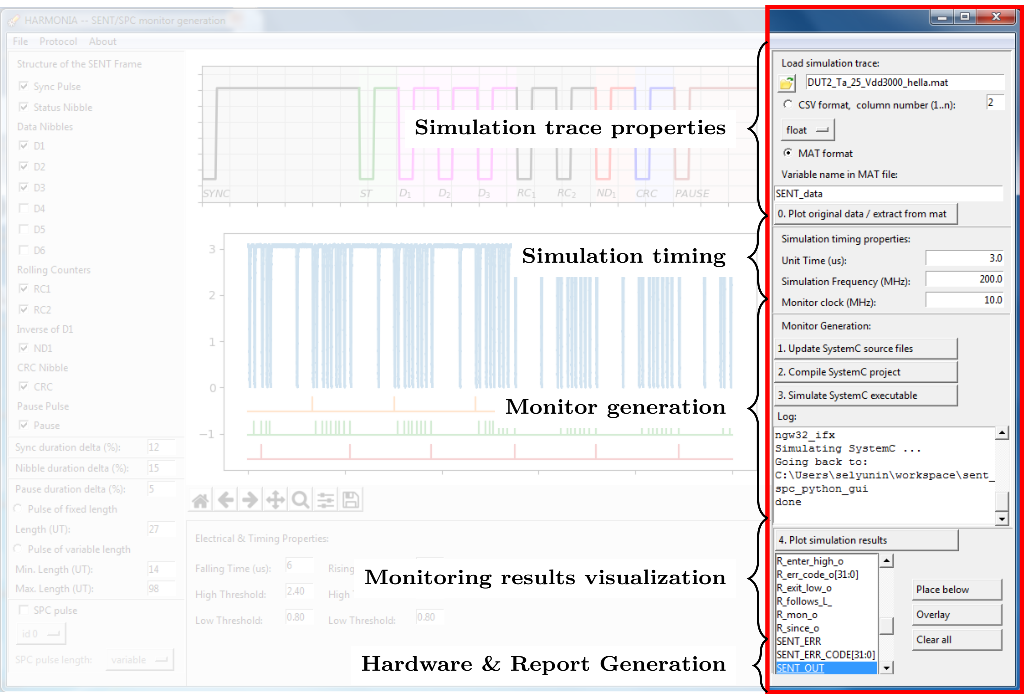
R_enter_high_o
R_err_code_o[31:0]
R_exit_low_o
R_follows_L_
R_mon_o
R_since_o
SENT_ERR
SENT_ERR_CODE[31:0]
SENT_OUT

Figure 13: Specifying the parameters of the protocol and defining the frame structure

Right Panel: Monitor Generation

The primary purpose of the GUI is to ease the interaction of the SystemC backend. The SystemC backend emulates hardware monitoring system, creates runtime monitor and necessary simulation infrastructure. In order to create a test bench, a user needs to define a signal to be monitored. The GUI supports both `.csv` and MATLAB `.mat` files. In the case of the `.csv` file a user needs to specify a number of the column that corresponds to the signal; in the case of the MATLAB file the user needs to specify a signal variable name in the `mat` file (see Fig. 14). For the monitoring to be correct, the user needs to specify simulation timing parameters (see Fig. 14) the clock frequency, at which the data was recorded, the clock frequency of the monitor and the unit time (protocol parameter).

The generation of the monitor and the offline monitoring is done in three steps: (i) updating the SystemC source files; (ii) compiling the SystemC monitor project; (iii) running the SystemC simulation of the monitor. After simulation has been finished, the user can visualize the results and plot the simulation signals (e.g. the signals that correspond to the detection of synchronization, nibble pulses, SENT frames, or error signals).



Simulation trace properties
Simulation timing
Monitor generation
Monitoring results visualization
Hardware & Report Generation
0. Plot original data / extract from mat
1. Update SystemC source files
2. Compile SystemC project
3. Simulate SystemC executable
4. Plot simulation results
Place below
Overlay
Clear all

Load simulation trace: DUT2_Ta_25_Vdd3000_hella.mat

○ CSV format, column number (1..n): 2

float

● MAT format

Variable name in MAT file: SENT_data

Simulation timing properties:

Unit Time (us): 3.0

Simulation Frequency (MHz): 200.0

Monitor clock (MHz): 10.0

Monitor Generation:

Log:

```
ngw32_ifx
Simulating SystemC ...
Going back to:
C:\Users\selyunin\workspace\sent_
spc_python_gui
done
```

R_enter_high_o
R_err_code_o[31:0]
R_exit_low_o
R_follows_L_
R_mon_o
R_since_o
SENT_ERR
SENT_ERR_CODE[31:0]
SENT_OUT

Figure 14: Right panel: elements for the monitor generation

Plot Panel: Vizualizations

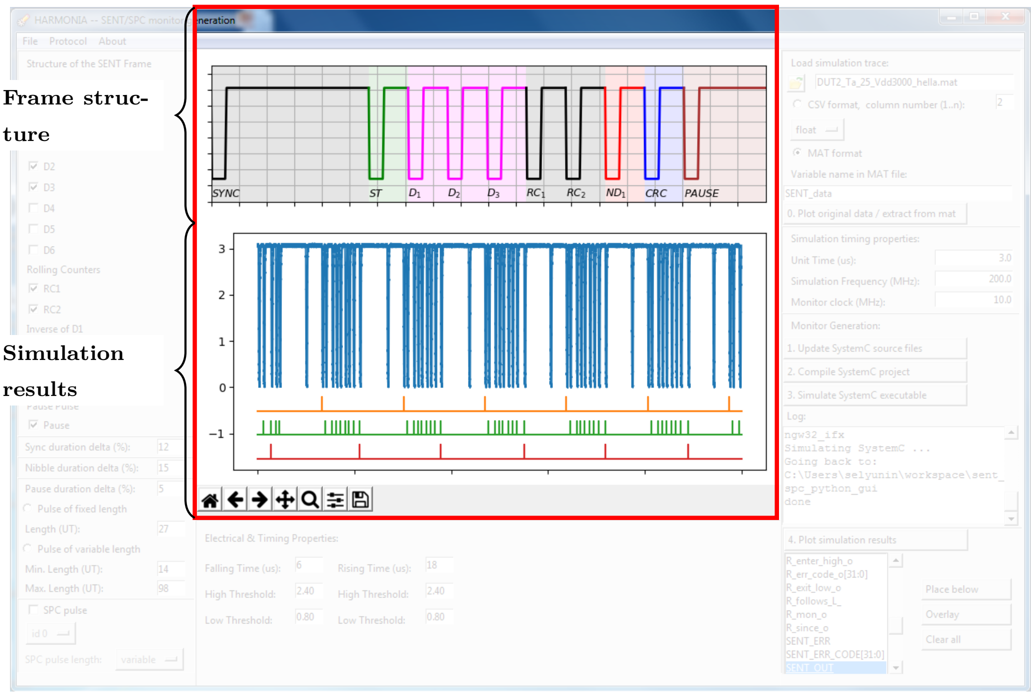
The plot panel (see Fig. 15) was developed to display three types of information: (i) structure of the protocol frame; (ii) the original recorded signal; (iii) the results of the SystemC simulation. The panel consist of two plots, the first plot represents the frame structure, and the second plot is either used to display the original signal or the simulation results. When the simulation is finished, the user can plot signals from the list of signals.

Bottom panel: Electrical interface requirements

At the bottom panel the user can specify electrical interface requirements, such as the falling and rising times, together with the corresponding comparator levels (see Fig. 16). The STL requirements for these formulae stay fixed, only the parameters in the source files are updated.

Data Decoding and Monitoring Report

After the SystemC simulation is finished, one can examine the monitoring report and the decoded data, that are available in a plain text format (see Fig. 17 and step 10 of the SENT tutorial).


HARMONIA -- SENT/SPC monitor generation
File
Protocol
About
Structure of the SENT Frame
Frame structure
Simulation results
SYNC
ST
D_1
D_2
D_3
RC_1
RC_2
ND_1
CRC
PAUSE
3
2
1
0
−1
Electrical & Timing Properties:
0. Plot original data / extract from mat
Simulation timing properties:
Monitor Generation:
1. Update SystemC source files
2. Compile SystemC project
3. Simulate SystemC executable
Log:
4. Plot simulation results
Place below
Overlay
Clear all

☑ D2
☑ D3
☐ D4
☐ D5
☐ D6
Rolling Counters
☑ RC1
☑ RC2
Inverse of D1
☑ Pause

Sync duration delta (%): 12
Nibble duration delta (%): 15
Pause duration delta (%): 5
○ Pulse of fixed length
Length (UT): 27
○ Pulse of variable length
Min. Length (UT): 14
Max. Length (UT): 98
☐ SPC pulse
id 0
SPC pulse length: variable

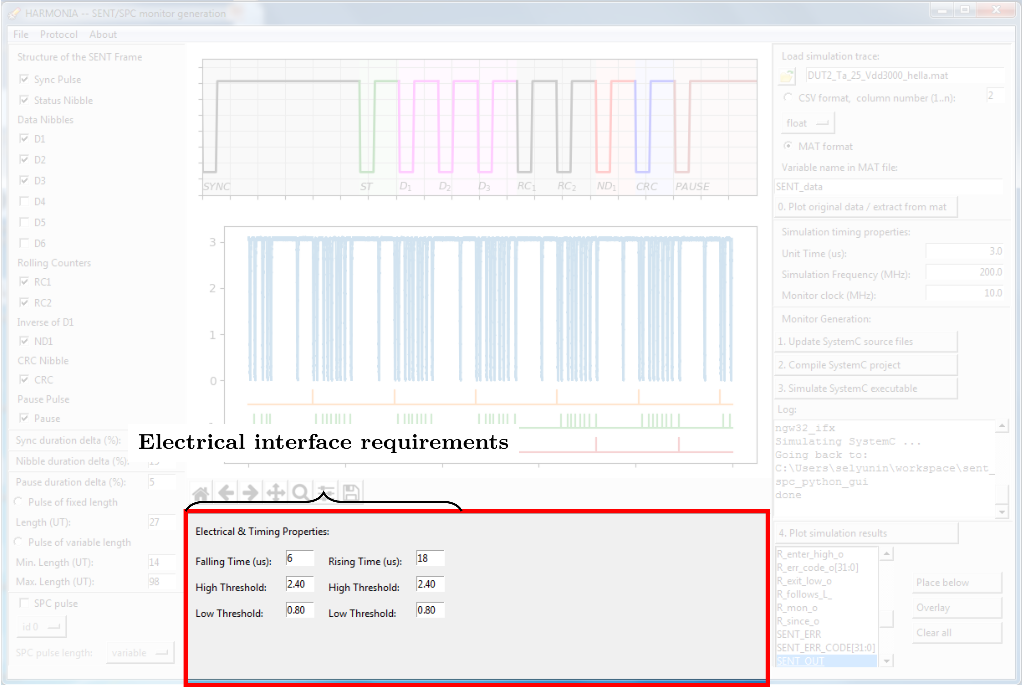
Falling Time (us): 6
Rising Time (us): 18
High Threshold: 2.40
High Threshold: 2.40
Low Threshold: 0.80
Low Threshold: 0.80

Load simulation trace: DUT2_Ta_25_Vdd3000_hella.mat
○ CSV format, column number (1..n): 2
float
● MAT format
Variable name in MAT file: SENT_data
Unit Time (us): 3.0
Simulation Frequency (MHz): 200.0
Monitor clock (MHz): 10.0

```
ngw32_ifx
Simulating SystemC ...
Going back to:
C:\Users\selyunin\workspace\sent_
spc_python_gui
done
```

R_enter_high_o
R_err_code_o[31:0]
R_exit_low_o
R_follows_L_
R_mon_o
R_since_o
SENT_ERR
SENT_ERR_CODE[31:0]
SENT_OUT

Figure 15: Plot panel


HARMONIA -- SENT/SPC monitor generation
File
Protocol
About
Structure of the SENT Frame
SYNC
ST
D_1
D_2
D_3
RC_1
RC_2
ND_1
CRC
PAUSE
3
2
1
0
Electrical interface requirements
Electrical & Timing Properties:
0. Plot original data / extract from mat
Simulation timing properties:
Monitor Generation:
1. Update SystemC source files
2. Compile SystemC project
3. Simulate SystemC executable
Log:
4. Plot simulation results
Place below
Overlay
Clear all

☑ Sync Pulse
☑ Status Nibble
Data Nibbles
☑ D1
☑ D2
☑ D3
☐ D4
☐ D5
☐ D6
Rolling Counters
☑ RC1
☑ RC2
Inverse of D1
☑ ND1
CRC Nibble
☑ CRC
Pause Pulse
☑ Pause

Sync duration delta (%):
Nibble duration delta (%):
Pause duration delta (%): 5
○ Pulse of fixed length
Length (UT): 27
○ Pulse of variable length
Min. Length (UT): 14
Max. Length (UT): 98
☐ SPC pulse
id 0
SPC pulse length: variable

Falling Time (us): 6
Rising Time (us): 18
High Threshold: 2.40
High Threshold: 2.40
Low Threshold: 0.80
Low Threshold: 0.80

Load simulation trace: DUT2_Ta_25_Vdd3000_hella.mat
○ CSV format, column number (1..n): 2
float
● MAT format
Variable name in MAT file: SENT_data
Unit Time (us): 3.0
Simulation Frequency (MHz): 200.0
Monitor clock (MHz): 10.0

```
ngw32_ifx
Simulating SystemC ...
Going back to:
C:\Users\selyunin\workspace\sent_
spc_python_gui
done
```

R_enter_high_o
R_err_code_o[31:0]
R_exit_low_o
R_follows_L_
R_mon_o
R_since_o
SENT_ERR
SENT_ERR_CODE[31:0]
SENT_OUT

Figure 16: Bottom panel

```
@ 1312.05 us: CRC nibble seen
Electrical & timing requirements for the NIBBLE pulse satisfied
CRC = 7
NIBBLE pulse length: 28.5 us (19 UT)
+++++++++++++++++++++++++++++++++++++++++++++++++++++++
@ 1312.15 us:  Correct frame seen
STATUS = 4; D1 = 6; D2 = 1; D3 = 0; RC1 = 1; CRC = 7;
Angle value: 1552
#######################################################
=======================================================
@ 1641.95 us: SPC trigger for sensor ID 0 seen
STL requirements for the SPC trigger pulse satisfied
SPC pulse length: 41.2 us (27.4667 UT)
```

Figure 17: Excerpt from the monitoring report

Configuration `*.ini` Files

Configuration files are structured human-friendly text files, that are used to save and restore the parameters of the monitoring framework. These files are split in sections, see Fig. 18 for a configuration file example. The default location of configuration files: `\workspace\sent_spc_python_gui\config\`.

If the configuration is not applied (e.g. `spc_protocol = no`), then the related parameters are ignored (the same is valid for MATLAB/CSV file parameters.)

Behavioral Code and Test data

Behavioral SystemC Code

The goal of this section is to sketch the architecture of the monitoring framework. The Figure 19 shows the directory tree structure for the behavioral code.

Test Data

The recorded traces are located: `\workspace\SystemC_Runtime_MON_PoC\meas_traces\`.

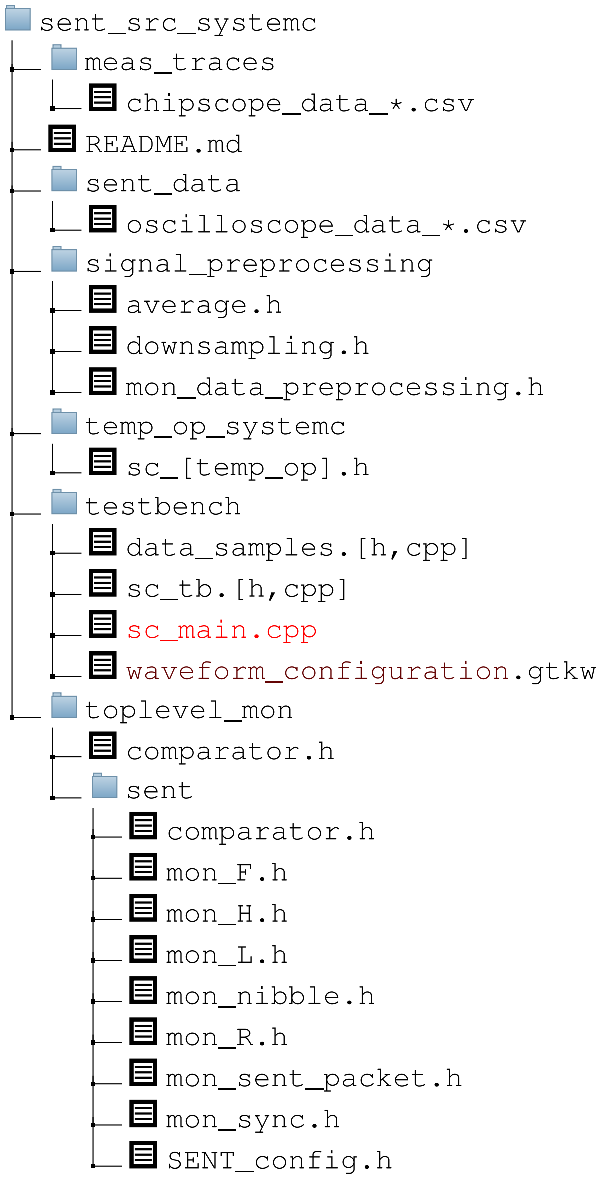
```
meas_traces
├── chipscope_data_*.csv
├── osci_data_*.csv
└── matlab_data_*.mat
```

Runtime monitoring framework can handle: MATLAB files, recordings from the oscilloscope, or recorded signals from the Chipscope (e.g. from the Line Emulizer).

Important: There should be **no** heading information in CSV files (i.e. the first line of the file should be data, and not ASCII names of the signals, frequency of recording, etc.). Floating point numbers should be separated with commas.

```
[SIMULATION_TRACE]
sim_filename_relative = workspace/SystemC_Runtime_MON_PoC/meas_traces/SENT/5014_single_SENT_default.csv
sim_filename_short = 5014_single_SENT_default.csv
sim_freq = 10.0
is_mat_file = no
is_csv_file = yes

[MATFILE]
mat_variable = SENT_data

[CSVFILE]
csv_column = 5
encoding = float

[VCDFILE]
vcd_filename = ../SystemC_Runtime_MON_PoC/DEBUG/testbench/monitor_dump.vcd
vcd_target_timescale = ns

[SENT.FRAME]
sync_present = yes
status_present = yes
D1_present = yes
D2_present = yes
D3_present = yes
D4_present = no
D5_present = no
D6_present = no
RC1_present = yes
RC2_present = yes
ND1_present = yes
CRC_present = yes
pause_present = yes
pause_fixed_len = yes

[SENT.TIMING]
mon_freq = 10.0
ut_time = 3.0
sync_tolerance_lim = 12
nibble_tolerance_lim = 12
pause_tolerance_lim = 5
fixed_pause_len = 27
variable_min_pause = 14
variable_max_pause = 98
falling_time_us = 6
rising_time_us = 18
falling_upper_threshold = 4.1
rising_upper_threshold = 4.1
falling_lower_threshold = 1.35
rising_lower_threshold = 1.35

[SPC]
spc_protocol = no
spc_sensor_id = 0
spc_is_fixed_length = no
spc_amplitude = 0.0
spc_pulse_tolerance_lim = 0.0
spc_bus_mode = no
spc_id0 = yes
spc_id1 = no
spc_id2 = no
spc_id3 = no
spc_id0_config = 01
spc_id1_config = 01
spc_id2_config = 01
spc_id3_config = 01
```

Figure 18: Example configuration file



```
sent_src_systemc
├── meas_traces
│   └── chipscope_data_⋆.csv
├── README.md
├── sent_data
│   └── oscilloscope_data_⋆.csv
├── signal_preprocessing
│   ├── average.h
│   ├── downsampling.h
│   └── mon_data_preprocessing.h
├── temp_op_systemc
│   └── sc_[temp_op].h
├── testbench
│   ├── data_samples.[h,cpp]
│   ├── sc_tb.[h,cpp]
│   ├── sc_main.cpp
│   └── waveform_configuration.gtkw
└── toplevel_mon
    ├── comparator.h
    └── sent
        ├── comparator.h
        ├── mon_F.h
        ├── mon_H.h
        ├── mon_L.h
        ├── mon_nibble.h
        ├── mon_R.h
        ├── mon_sent_packet.h
        ├── mon_sync.h
        └── SENT_config.h
```

Figure 19: Behavioral code structure

```
1,0,111101111111,111111111100,111111111100
2,0,111100001100,111111111100,111111111100
3,0,111100100100,111111111110,111111111110
```

```
-6.46982000e-004,1.01999994e-002
-6.46980000e-004,1.09999994e-002
-6.46978000e-004,1.09999994e-002
```

```
4.64000000
5.04000000
5.12000000
```

User Study

The developed software within the work on the applied part of the thesis was evaluated at Infineon by a verification engineer and a concept design engineer within the HARMONIA project (**Har**dware **moni**toring for **a**utomotive). Each user was asked to fill in the questionnaire and evaluate runtime monitoring of the SENT and SPC protocols w.r.t. the following criteria: (1) usefulness; (2) ease of use; (3) ease of learning; (4) satisfaction.

Questionnaire is based on two previously published works: (i) USE Questionnaire: Usefulness, Satisfaction, and Ease of use / Lund, A.M. (2001) Measuring Usability with the USE Questionnaire. STC Usability SIG Newsletter, 8:2; (ii) Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology / Davis, F. D. MIS Quarterly 1989 v.13 n.3 p.319-340. A user gives his response on a Likert scale, the results are summarized in Fig. 20.

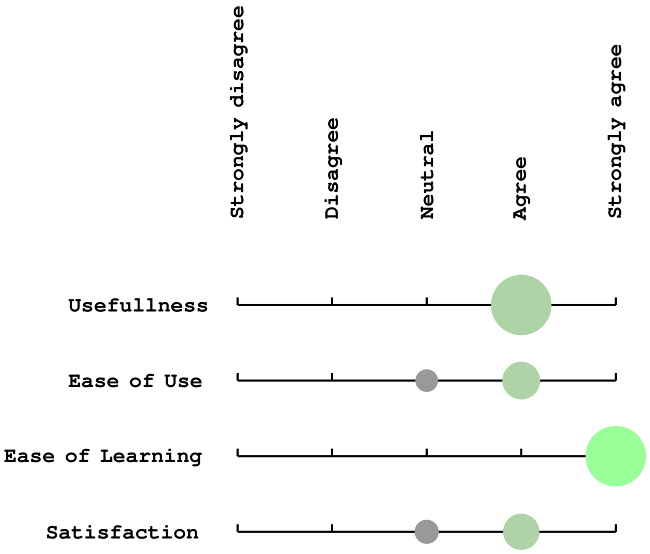


Figure 20: Summary of user study results

Curriculum vitæ

Konstantin Selyunin

Curriculum vitæ

Treitlstr. 3, CPS Group
1040 Vienna, Austria
T +43 (1) 58801 - 18225
M +43 (660) 4627 103
E kselyunin@acm.org
W www.selyunin.com

Education

PhD Candidate, *Vienna University of Technology*, Vienna, Austria. **2012–till Present**

- student of Vienna PhD School of Informatics
- 15 PhD & Master Courses (54 ECTS) successfully completed, `1.06` average (`"1"` corresponds to the excellent note)
- the research results presented at AVM'15, CDC'15, DATE'16, DVCON'16, MTCPS'16, DAC'16, RV'16, IWANN'17, CAV'17, group and project meetings

Dipl.Ing., *Omsk State Transport University*, Omsk, Russia, *with Distinction*. **2005–2010**

Speciality: "Automation remote control and communications on railway transport". Major "Microprocessor- and information control systems".

Experience

Project Assistant, *Vienna University of Technology*, Vienna, Austria. **2014–till Present**

Development of hardware runtime monitors for checking formal properties expressed in Signal Temporal Logic within the HARMONIA FFG project.

- Application of High-Level Synthesis of synthesizing hardware runtime monitors
  - A complete flow from natural language specification to hardware generation
  - Application IBM's TrueNorth model for creating MTL monitors
- Embedding SystemC runtime monitors in a chip concept
- Project Website administrator and maintainer `http://harmonia-project.com/`

University Assistant, *Omsk State Transport University*, Omsk, Russia. **2010–2012**

Teaching assistant for the "Foundations of Microprocessor Technology" course. Preparing course content, labs, supervising students.

Technical Skills

Programming languages: `C/C++, Cuda C, Python, Ruby, Bash, R, Scala, MATLAB/Simulink, Octave, Mathcad, VHDL`

Operating Systems: Linux (preferred), Windows

GNU tools: make, gcc, gdb

Robotic frameworks: ros

IDEs: Eclipse, Vivado

Text typesetting: `vim`, LaTeX, MS Office if required
Bugtracking systems: redmine (configuration & maintenance)
PAAS, IAAS: Heroku, Amazon EC2, S3

Publications

[NBN$^+$16] Thang Nguyen, Ezio Bartocci, Dejan Nickovic, Radu Grosu, Stefan Jaksic, and Konstantin Selyunin. The HARMONIA project: Hardware monitoring for automotive systems-of-systems. In *Proc. of Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications - 7th International Symposium, ISoLA 2016, Corfu, Greece, October 10-14*, pages 371–379, 2016.

[SHR$^+$17] Konstantin Selyunin, Ramin M. Hasani, Denise Ratasich, Ezio Bartocci, and Radu Grosu. Computing with biophysical and hardware-efficient neural models. In *Proc. of the 14th International Work-Conference on Artificial Neural Networks, IWANN 2017, Cadiz, Spain, June 14-16*, pages 535–547, 2017.

[SJN$^+$17] Konstantin Selyunin, Stefan Jaksic, Thang Nguyen, Christian Reidl, Udo Hafner, Ezio Bartocci, Dejan Nickovic, and Radu Grosu. Runtime monitoring with recovery of the sent communication protocol. In *Proc. of the 29th International Conference on Computer Aided Verification, (CAV 2017), Heidelberg, Germany, July 24-28*, pages 1–18, 2017.

[SNB$^+$16a] Konstantin Selyunin, Thang Nguyen, Ezio Bartocci, Dejan Nickovic, and Radu Grosu. Monitoring of MTL Specifications With IBM's Spiking-Neuron Model. In *Proc. of the 19th Design, Automation and Test in Europe Conference and Exhibition, DATE 2016, Dresden, Germany, March 14-18*, 2016.

[SNB$^+$16b] Konstantin Selyunin, Thang Nguyen, Andrei Daniel Basa, Ezio Bartocci, Dejan Nickovic, and Radu Grosu. Applying High-Level Synthesis for Synthesizing Hardware Runtime STL Monitors of Mission-Critical Properties. In *Electronic Proc. of the 13th Design and Verification Conference and Exhibition (DVCon 2016), San Jose, CA, USA, February 28- March 3*, pages 1–8, 2016.

[SNBG16] Konstantin Selyunin, Thang Nguyen, Ezio Bartocci, and Radu Grosu. Applying runtime monitoring for automotive electronic development. In *Proc. of the International Conference on Runtime Verification, RV 2016, Madrid, Spain, September 23-30, 2016*, pages 462–469, 2016.

[SRB$^+$15] Konstantin Selyunin, Denise Ratasich, Ezio Bartocci, Md. Ariful Islam, Scott A. Smolka, and Radu Grosu. Neural programming: Towards adaptive control in cyber-physical systems. In *Proc. of the 54th IEEE Conference on Decision and Control, CDC 2015, Osaka, Japan, December 15-18, 2015*, pages 6978–6985, 2015.

Conference Presentations

- 2017, The 29th International Conference on Computer Aided Verification, (CAV 2017), Heidelberg, Germany [SJN+17]
- 2017, The 14th International Work-Conference on Artificial Neural Networks (IWANN 2017), Cadiz, Spain [SHR+17]
- 2016, The 16th RV conference and exhibition (RV 2016), Madrid, Spain [SNBG16]
- 2016, The 53rd DAC conference and exhibition (DAC 2016), Austin, Texas, United States (Work-in-progress presentation)
- 2016, The 19th DATE conference and exhibition (DATE 2016), Dresden, Germany [SNB+16a]
- 2016, The 28th Design and Verification conference and exhibition (DV-Con 2016), San Jose, CA, United States [SNB+16b]
- 2015, The 54th IEEE Conference on Decision and Control (CDC 2015), Osaka, Japan [SRB+15]

Seminars, Meetings & Workshops

- 2016, Workshop on Monitoring and Testing of Cyber-Physical Systems (MTCPS), Vienna, Austria
- 2015, Automatic Verification and Analysis of Complex Systems 2nd AVACS Autumn School, Oldenburg, Germany
- 2015, Alpine Verification Meeting, Attersee, Austria
- 2013, Dagstuhl Seminar 14122 : Verification of Cyber-Physical Systems, Schloss Dagstuhl Leibniz-Zentrum für Informatik GmbH, Germany

Peer-reviewing conference papers

RV’2017, CMSB’2017, SPIN’2017, RTNS’2016, ICCP’2016, NFM’2016, DATE’2016, FORMATS’2015, ISORC’2015, ICFEM’2014, RV’2014.

Languages

English: Fluent	*Full professional proficiency*
German: Advanced	*C1 certificate*
Russian: Fluent	*Mother language*